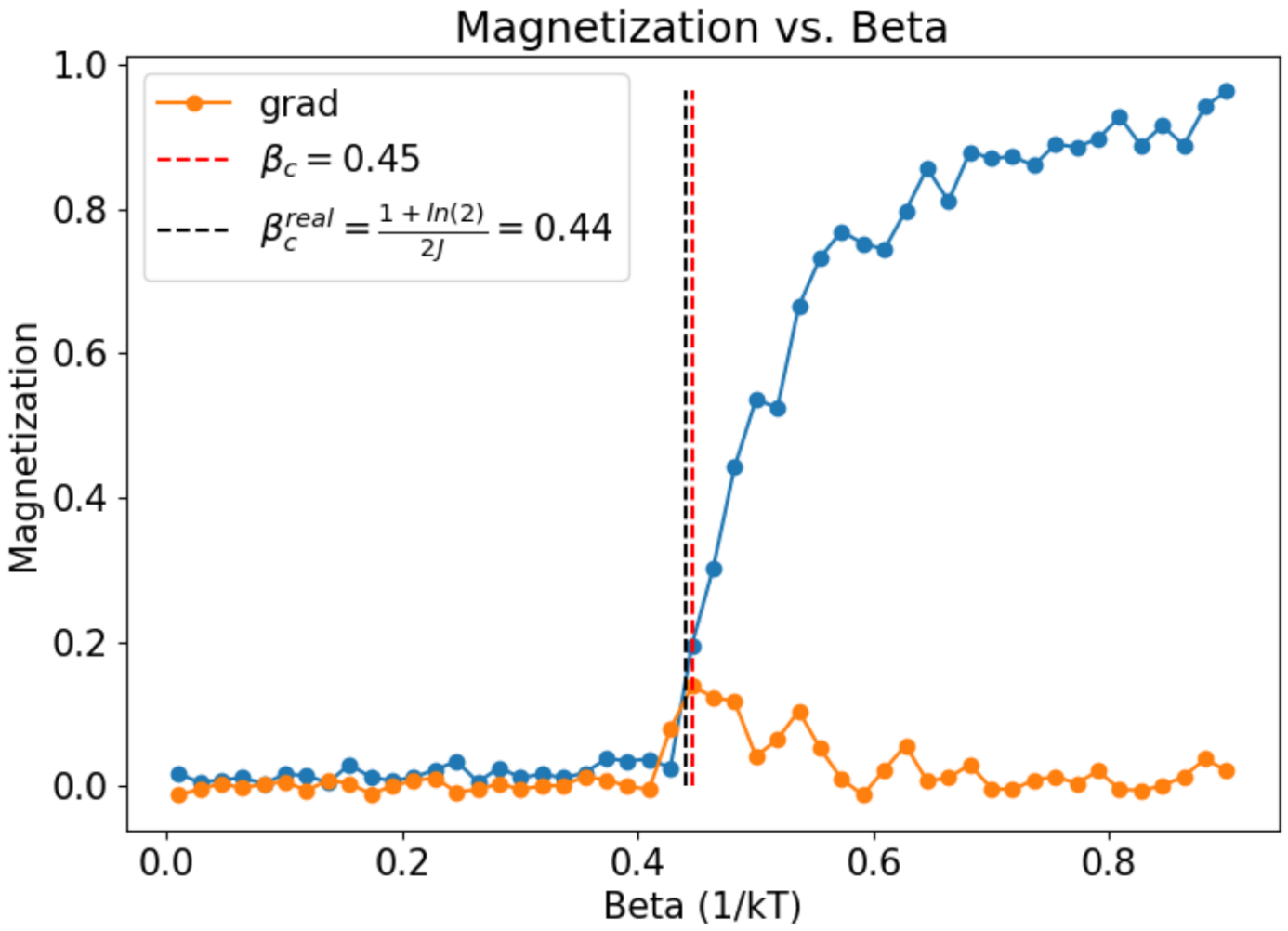


I found β_c by the maximum of the derivative

3) for anti-ferromagnet all that is required is to adjust the check. We want to flip spins around the random selected spin if they're the opposite sign.

```
if spin_lattice[i,j] == cluster_spin
if spin_lattice[i,j] == -cluster_spin
```

} anti-ferromagnet

and the prob. stays the same $p = 1 - e^{-\beta J}$

Where $J \to -J$ for the system gains energy when the spins are disaligned

Lets discuss the physics a bit. For an anti-ferro magnet without external magtic field there will be no net magnetization for $T=0$, for the two sub-lattices will cancel eachotherout. For $T \to \infty$ there can be tiny magnetization because the thermal fluctuations can produce islands of magnetization, but it is insignificant. Whith external magnetic field it is mor interesting. The interaction tries to disalign the spins but the fidld tries to align them. At $\beta = 0$ the thermal fluctuations will win and the field will not be able to fight it.

As β increses the system starts to order. Without the field, there will be no net mag. cha of J. With the field theres an inbalance in the field direction, leading to non-zero mag. but a small one. As $\beta \to \infty$ theres strong ordering due to the field and J.

Again, no $h \to$ no M $\quad h \longrightarrow$ small M that saturates.

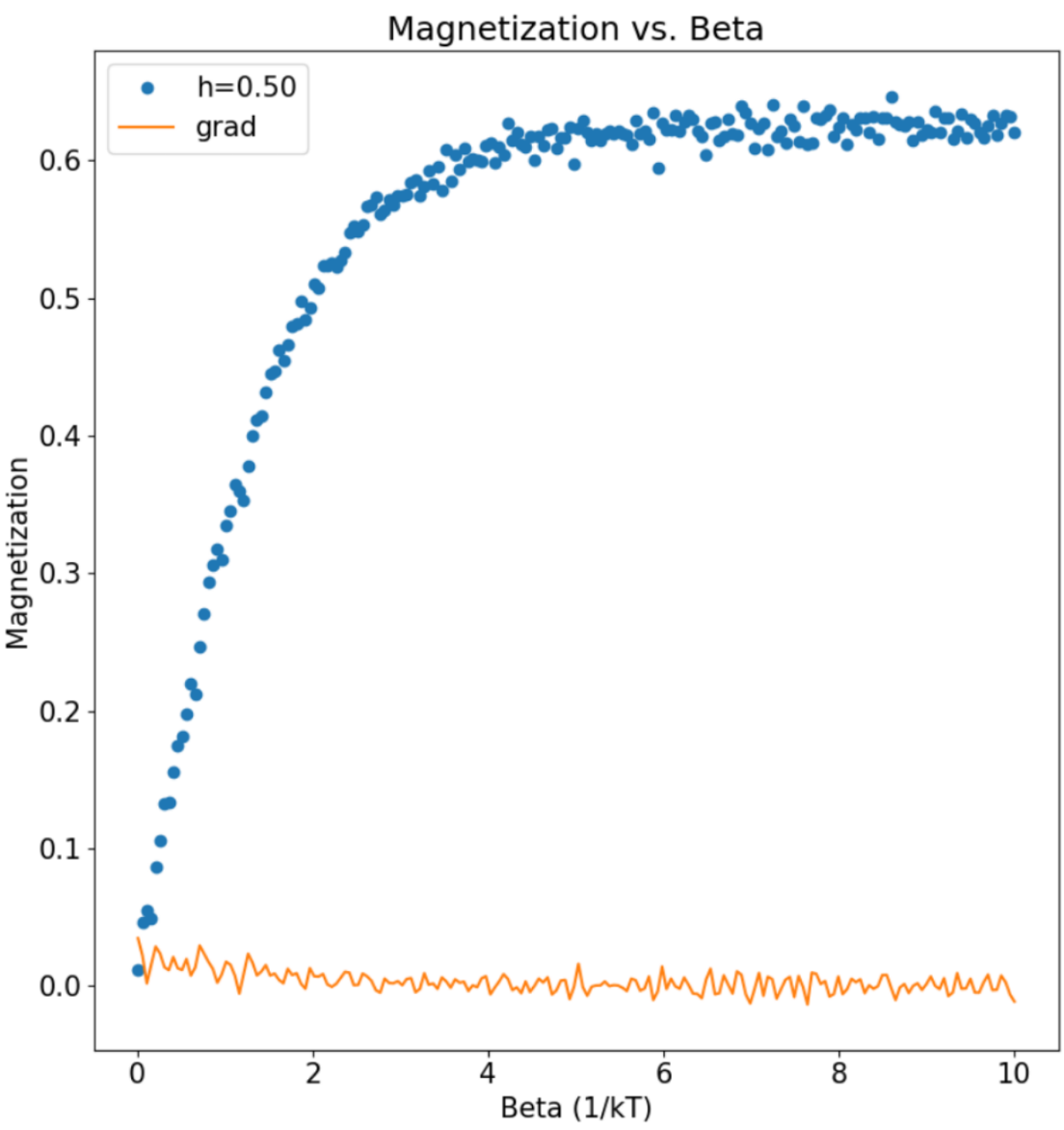
The point where M starts to rise, how fast it will rise, and at which value it will saturates, all depend on the ratio $\frac{\beta}{J}$.

4)

With magnetic field we have a small problem. We can only calculate the energy diff $E_A - E_B$ after we construct the cluster, because it depends on the size of the cluster. So we can construct the cluster as before and than add a check to see if the cluster should flip based on the external field. The energy contribution is $e^{2\beta h \sigma_i S} = E_h$ were σ_i is the cluster spin direction and S is the cluster size. and the probability will be exactly this factor. Why not $1 - E_h$?

Lets check: for same orientation $E_h > 1$ and the random numer generator get numbers between 0 and 1, so it will always fail the check. (wait flip)

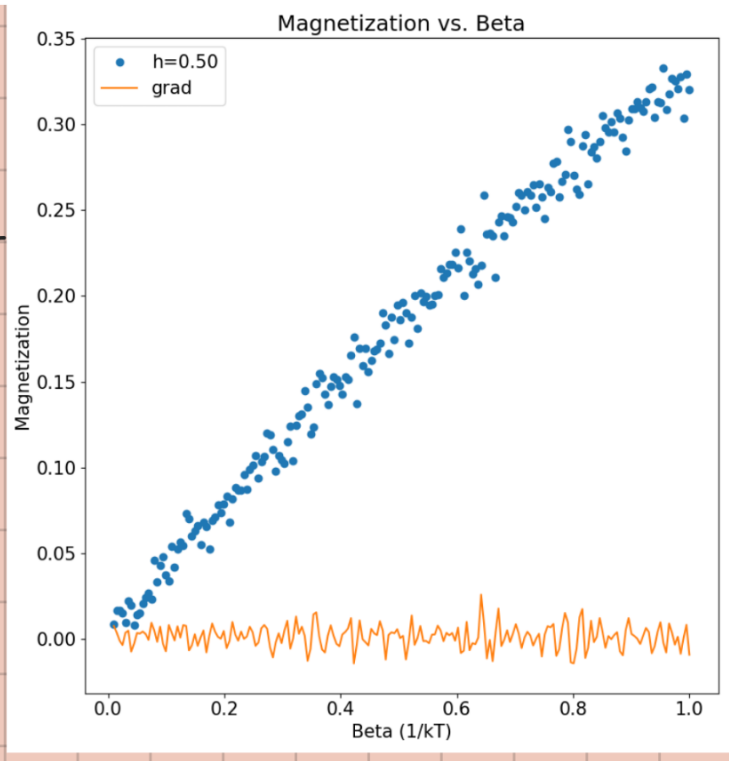
for opposite orientation $E_h < 1$ the check will only pass with regard to E_h whic is basically the Boltzman factor



(h

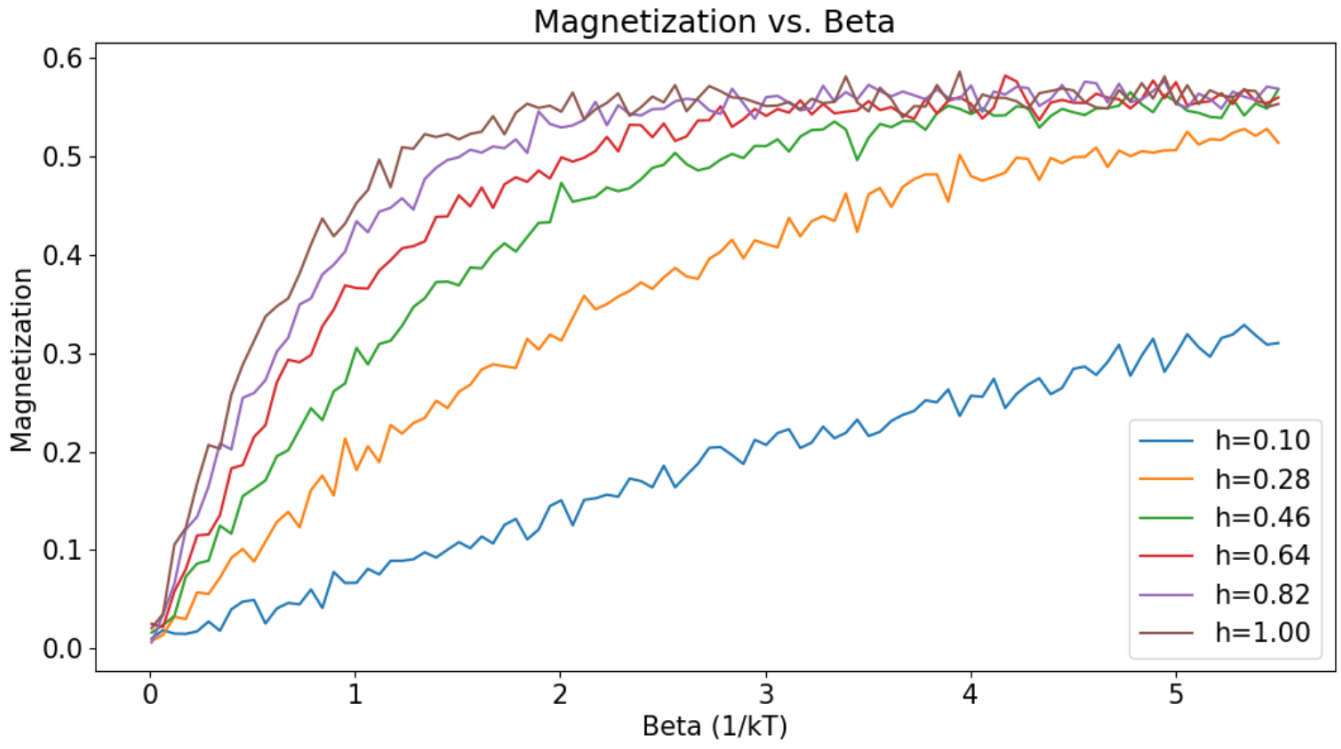
Anti-ferromagnet with external mag. field

Unfortunately I cant seem to be able to find β_c. Not numerically and not by eye.



5) As h increases you need less thermal energy to overcome J. h increases → T_c drops

h decreses → T_c increses



Although I cant see β_c at all, no matter h.

ex3_5.py | ex3q3__.py | ex3q3_6.png

```python
def initialize_lattice(L):
    """Initialize the lattice with random spins (-1 or +1)."""
    return np.random.choice([-1, 1], size=(L, L))


def wolff_update(spin_lattice, beta, J, h):
    """Perform a single Wolff cluster update."""
    L = spin_lattice.shape[0]
    visited = np.zeros_like(spin_lattice, dtype=bool)

    # Pick a random seed spin
    x, y = np.random.randint(L), np.random.randint(L)
    cluster = [(x, y)]
    visited[x, y] = True
    cluster_spin = spin_lattice[x, y]

    # Define probability for adding a neighbor to the cluster
    add_prob = 1 - np.exp(- 2 * beta * J)

    # Grow the cluster
    i = 0
    while i < len(cluster):
        cx, cy = cluster[i]
        neighbors = [
            ((cx - 1) % L, cy),
            ((cx + 1) % L, cy),
            (cx, (cy - 1) % L),
            (cx, (cy + 1) % L),
        ]

        for nx, ny in neighbors:
            if not visited[nx, ny] and spin_lattice[nx, ny] == np.sign(J)*cluster_spin:
                if np.random.rand() < add_prob:
                    cluster.append((nx, ny))
                    visited[nx, ny] = True
        i += 1

    # Flip the entire cluster
    if np.random.rand() < np.exp(2 * beta * h * len(cluster) * cluster_spin):
        for cx, cy in cluster:
            spin_lattice[cx, cy] *= -1

    return spin_lattice
```

ex3_5.py | ex3q3__.py | ex3q3_6.png

```python
    return spin_lattice


def precompute_frames(L, beta, J, h, steps, method):
    """Precompute all frames for the simulation."""
    spin_lattice = initialize_lattice(L)
    highlight_cluster = np.zeros_like(spin_lattice, dtype=float)
    magnetizations = []
    frames = []

    for _ in range(steps):
        spin_lattice = wolff_update(spin_lattice, beta, J, h)
        frames.append((spin_lattice.copy(), highlight_cluster.copy()))
        magnetizations.append(np.abs(np.sum(spin_lattice)) / (L * L))
    return frames, magnetizations


def simulate(L, beta, J, steps, h=0, animate=True):
    """Simulate the Ising model on a 2D lattice."""
    frames, magnetizations = precompute_frames(L, beta, J, h, steps)

    if animate:
        fig, ax = plt.subplots()
        im = ax.imshow(frames[0][0], cmap="coolwarm", interpolation="nearest")
        cluster_overlay = ax.imshow(
            frames[0][1],
            cmap="Grays",
            alpha=0.2,
            interpolation="nearest",
            vmin=0,
            vmax=1,
        )

        def update(frame):
            nonlocal cluster_overlay, im
            spin_lattice, highlight_cluster = frames[frame]
            im.set_array(spin_lattice)
            cluster_overlay.set_array(highlight_cluster)
            return [im, cluster_overlay]

W       ani = FuncAnimation(fig, update, frames=len(
            frames), blit=True, repeat=False)
        plt.title("Spin Lattice Evolution")
        plt.colorbar(cluster_overlay, label="Spin")
        plt.show()
    else:
        return frames[-1][0], magnetizations


W def plot_magnetization_vs_beta(L, J, h, steps, betas, method="wolff", plt_tc=False):
    """Plot the magnetization as a function of beta."""
```

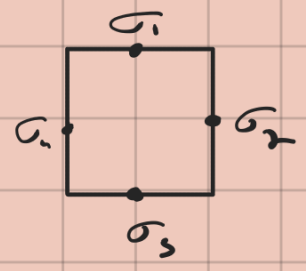
NORMAL | main | ⚠ 5 | ex3_5.py | utf-8 | python | 34% | 53:1

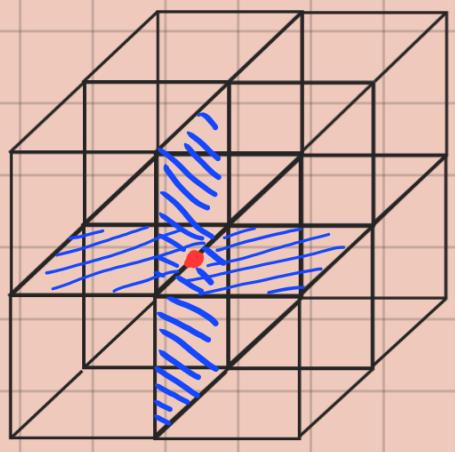
ex3_5.py | ex3q3__.py | ex3q3_6.png

```
        return frames[-1][0], magnetizations


def plot_magnetization_vs_beta(L, J, h, steps, betas, method="wolff", plt_tc=False):
    """Plot the magnetization as a function of beta."""
    magnetizations = []

    for beta in tqdm(betas, desc=f"Calculating Magnetization ({method})"):
        _, mags = precompute_frames(L, beta, J, h, steps, method)
        magnetizations.append(np.mean(mags))

    plt.plot(betas, magnetizations, marker="o", ls="None", label=f"h={h:.2f}")
    plt.plot(betas, np.gradient(magnetizations), label="grad")
    if plt_tc:
        imax = np.gradient(magnetizations).argmax()
        betac = betas[imax]
        plt.vlines(
            x=betac,
            ymin=0,
            ymax=max(magnetizations),
            colors="r",
            ls="dashed",
            label=rf"$\beta_c={betac:.2f}$",
        )
        plt.vlines(
            x=np.log(1 + np.sqrt(2)) / (2 * np.abs(J)),
            ymin=0,
            ymax=max(magnetizations),
            colors="k",
            ls="dashed",
            label=r"$\beta_c^{real}=\frac{1+ln(2)}{2J}=$"
            + rf"${np.log(1+np.sqrt(2))/(2*np.abs(J)):.2f}$",
        )
    plt.title("Magnetization vs. Beta")
    plt.xlabel("Beta (1/kT)")
    plt.ylabel("Magnetization")
    plt.legend()


if __name__ == "__main__":
    # Parameters
    L = 64  # Lattice size
    J = -1  # Interaction strength
    h = 0.5
    steps = 10000  # Number of Monte Carlo steps

    # Simulate with animation
    beta = 0.3
    # simulate(L, beta, J, steps, method="wolff", animate=True)

    # Plot magnetization vs beta for Wolff and Metropolis
    betas = np.linspace(0.01, 10, 200)
    hs = np.linspace(0.1, 1, 6)
    #for h in hs:
    plot_magnetization_vs_beta(L, J, h, steps, betas, plt_tc=0)
    plt.show()
    # betas = np.linspace(0.1, 10, 50)
    # plot_magnetization_vs_beta(L, -J, 1000, betas, method="metropolis", h=0.5)
```

NORMAL main 5 ex3_5.py utf-8 python 80% 124:1

(a) $H = -J \sum \sigma_1 \sigma_2 \sigma_3 \sigma_4$





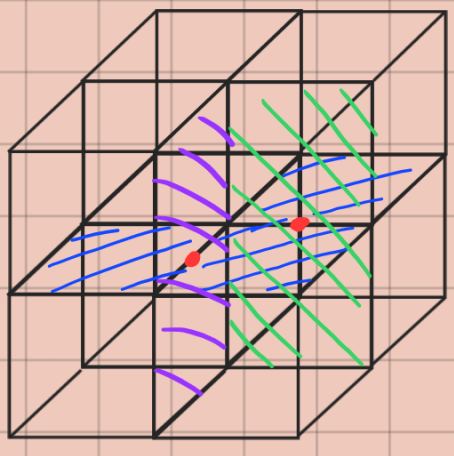
1st excitation:

4 plaquettes $\to t^4$

degeneracy: N sites $\Rightarrow$ 6N edges

But each edge is shared between 2 sites, so we divide by 2 to avoid double counting

$\Rightarrow \frac{6N}{2} = \underline{\underline{3N}}$



2nd excitation: two edges on the same plaquette

7 Plaquettes, but the shared plaquette doesn't change sign $\to t^{7-2} = t^6$

degeneracy: 3N edges, next, there are 12 edges to pick from, but we don't care which is which so we divide by 2: $\frac{3N \cdot 12}{2} = 18N$

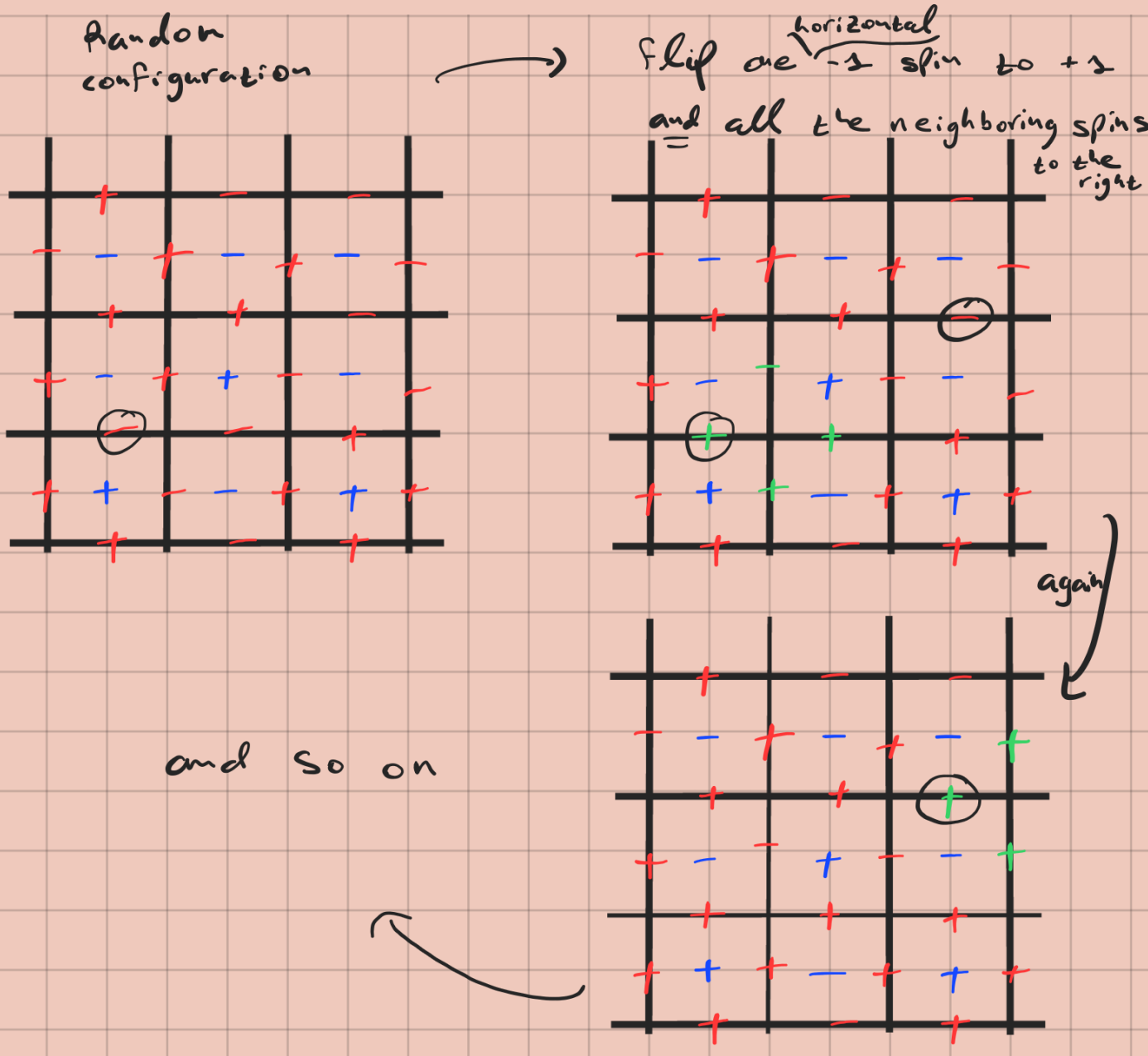
overall: $2^N e^{4N_p} (1 + 3N t^4 + 18N t^6 + \dots)$ where N_p = number of plaquettes

And what we saw is dual: $2^N \cosh^{3N} 4K (1 + 3N t^4 + 18N t^6 + \dots)$

$\Rightarrow$ $Z_{\text{low temp}}(N, K) = Z_{\text{high temp}}(N, \tilde{K})\, e^{\tilde{K} N_p} \cosh^{-3N}(\tilde{K})$

It is Dual!

2)

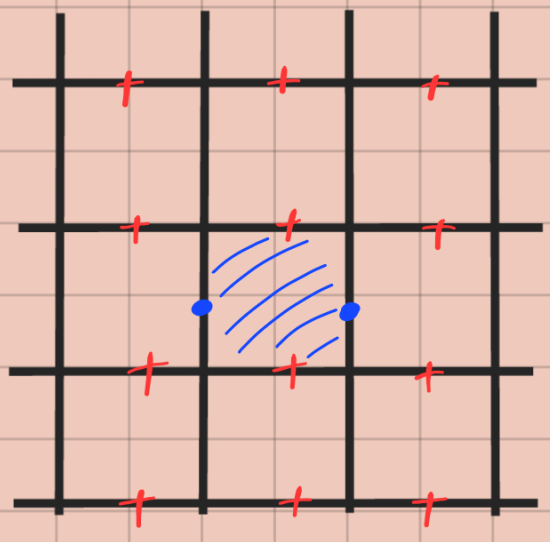


Why this works? because when you flip the 4 edges coming out of a vertex, the 4 plaquettes that might change value dont. Why? Because you flip 3 spins for each plaquette which doesn't change its value. And for the $\sigma_{Horizontal} = 1$ you dont need to flip at all so its obviously doesn't change the value of the plaquette.

With this algorithm we got a guage trans. such that every micro state is equivalent to a different micro state but with all $\sigma_H = 1$:

- visit every σ_H
  - if $\sigma_H = +1$
    - flip σ_H and 3 of its neighbors to the right

3)



The contribution of each plaquette is determined by 4 spins.

$$\text{So, } H = -J\sum_p \sigma_1^p \sigma_2^p \sigma_3^p \sigma_4^p$$

$$= -J \sum_p \sigma_1^p \sigma_3^p$$

We get that each row of plaquettes is completely decoupled from it neighbours.

Lets assume M rows and N cols.

$$\Rightarrow Z = \prod_M Z_i \quad ; \quad Z_i = \sum_{\{\sigma_{ij}\}} e^{\beta J \sum_{\langle ij \rangle} \sigma_{ij}\sigma_{ij}} = \lambda_+^N + \lambda_-^N = 2^N(\cosh^N \beta J + \sinh^N \beta J)$$

$$\begin{pmatrix} e^{\beta J} & e^{\beta J} \\ e^{-\beta J} & e^{\beta J} \end{pmatrix} \rightarrow \lambda_\pm = e^{\beta J} \pm e^{-\beta J}$$

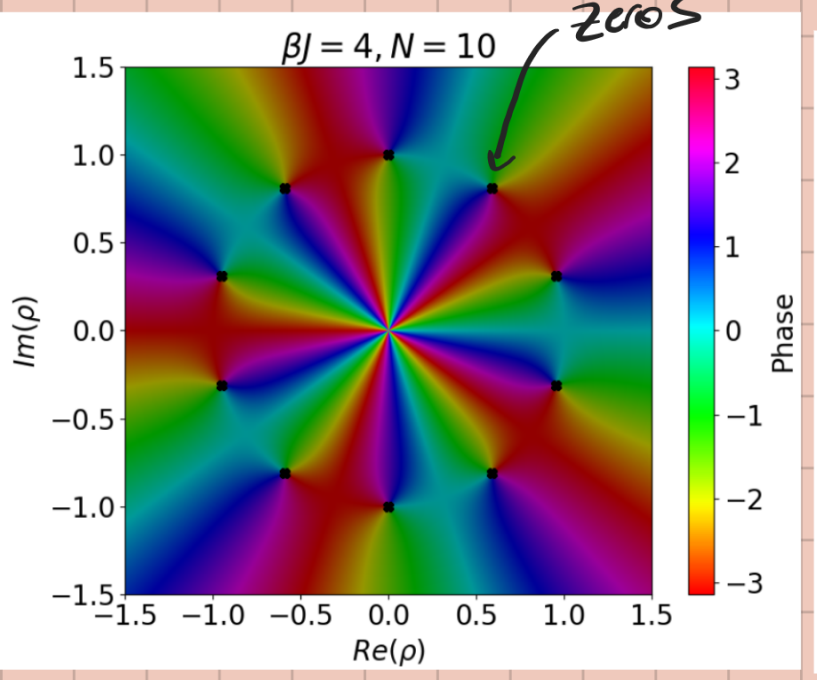
$$\Rightarrow Z = 2^{N+M}(\cosh^N \beta J + \sinh^N \beta J)^M$$

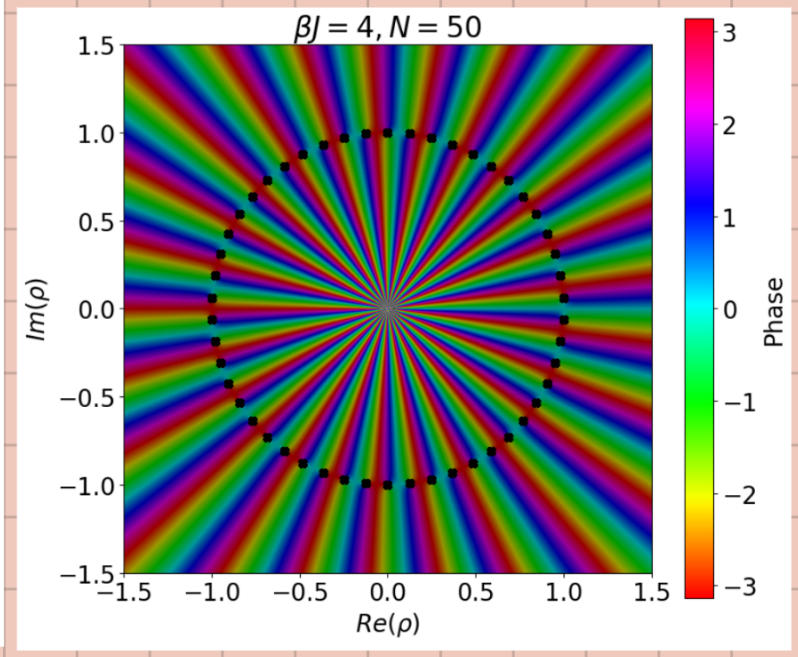
(3) $\lambda_\pm = e^{\beta J}\cosh\beta B \pm \sqrt{e^{2\beta J}\cosh^2\beta B - 2\sinh 2\beta J}$ $\quad e^{\beta B}+e^{-\beta B}$

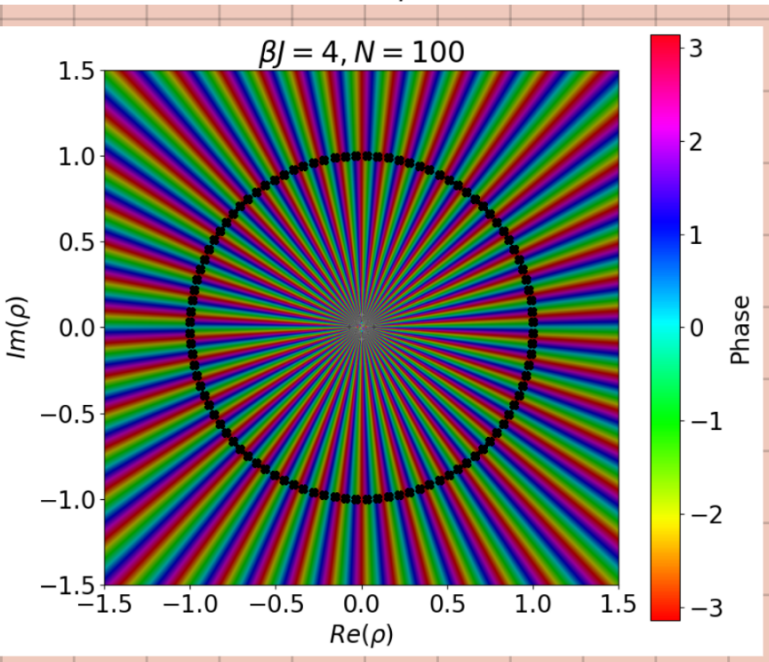
$\rho = e^{-2\beta B} \qquad z = e^{-2\beta J}$

$$\sqrt{e^{2\beta J}\cosh^2\beta B} \pm \sqrt{e^{2\beta J}\cosh^2\beta B - 2\sinh 2\beta J}$$

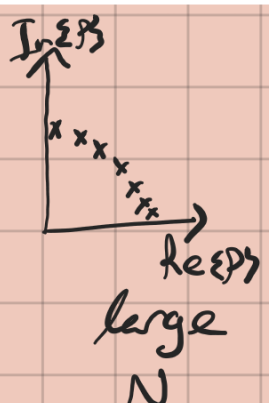
$$\sqrt{\frac{1}{4z}\left(\rho + 2 + \frac{1}{\rho}\right)} \pm \sqrt{\frac{1}{4z}\left(\rho + 2 + \frac{1}{\rho}\right) - \left(\frac{1}{2} - z\right)}$$







small N



large N

As N grows the zeroes come closer and closer to the real line. In the thermodynamic limit $N \to \infty$ there will be two solutions on the real line $\rho = \pm 1$ $\Leftrightarrow$ $B = 0$

And only then the system has two ground states and F is Non analytic in $\beta \to \infty$ $(T = 0)$

ex3_5.py | ex3q3__.py | ex3q3_6.png | ex3q3_.py

```python
N = 14
bJ = 4
tau = np.exp(-2 * bJ)


def lambda_minus(rho):
    # return 1 / (2 * np.sqrt(tau * rho)) * (1 + rho - np.sqrt(rho**2 + 2 * rho + tau**2))
    a = (1/rho + 2 + rho) / (4*tau)
    return np.sqrt(a) - np.sqrt(a - 1/tau + tau)
    #return (1 / np.sqrt(rho) + np.sqrt(rho))/(2*np.sqrt(tau)) - np.sqrt((1 / np.sqrt(rho) + np.sqrt(rho))**2 / (4*tau) - tau - 1/tau)


def lambda_plus(rho):
    # return 1 / (2 * np.sqrt(tau * rho)) * (1 + rho + np.sqrt(rho**2 + 2 * rho + tau**2))
    a = (1 / rho + 2 + rho) / (4 * tau)
    return np.sqrt(a) + np.sqrt(a - 1 / tau + tau)
    #return (1 / np.sqrt(rho) + np.sqrt(rho))/(2*np.sqrt(tau)) + np.sqrt((1 / np.sqrt(rho) + np.sqrt(rho))**2 / (4*tau) - tau - 1/tau)


def part_f(z):
    return lambda_plus(z) ** N + lambda_minus(z) ** N


# Set up the range for the complex plane
x = np.linspace(-1.5, 1.5, 1000)  # Real part range
y = np.linspace(-1.5, 1.5, 1000)  # Imaginary part range
X, Y = np.meshgrid(x, y)  # Create a grid
Z = X + 1j * Y  # Combine to create complex numbers


# Compute the function values
z = part_f(Z)

# Compute magnitude and phase for visualization
magnitude = np.abs(z)
phase = np.angle(z)

# Normalize magnitude for color intensity
# Use log scale for better contrast
magnitude_normalized = np.log(1 + magnitude)

# Create a color representation
hsv = np.zeros(Z.shape + (3,))
hsv[..., 0] = (phase + np.pi) / (2 * np.pi)  # Hue (normalized phase)
hsv[..., 1] = 1  # Saturation
hsv[..., 2] = magnitude_normalized / np.max(magnitude_normalized)  # Value

# Convert HSV to RGB for plotting
rgb = hsv_to_rgb(hsv)
```

NORMAL | main | ⚠ 9 | ex3q3_.py | utf-8 | python | 47% | 55:1

ex3_5.py | ex3q3__.py | ex3q3_6.png | ex3q3_.py

```
  rgb = hsv_to_rgb(hsv)


  # Find zeros of the function
  def real_imag_to_complex(v):
W     """Helper function to convert real/imaginary input to complex output for root-finding."""
      return v[0] + 1j * v[1]


  def complex_to_real_imag(c):
W     """Helper function to convert complex input to real/imaginary for root-finding."""
      return [c.real, c.imag]


  def objective(v):
      """Objective function for root-finding."""
      z = real_imag_to_complex(v)
      f_val = part_f(z)
      return [f_val.real, f_val.imag]


  ix = np.linspace(-1.2, 1.2, 100)
  iy = np.linspace(-1.2, 1.2, 100)
  # Search for zeros in the grid
  initial_guesses = []
  for _x in ix:
      for _y in iy:
W         initial_guesses.append([_x,_y])
  zeros = []

  for guess in initial_guesses:
      res = root(
          objective, guess, method="hybr", tol=1e-8
      )  # Use 'hybr' method for multidimensional root finding
      if res.success:
          zero = real_imag_to_complex(res.x)
          if zero not in zeros:  # Avoid duplicates
              zeros.append(zero)

  # Plot the result
  fig, ax = plt.subplots(figsize=(8, 6))
  img = ax.imshow(rgb, extent=[x.min(), x.max(),
                  y.min(), y.max()], origin="lower")
  ax.set_xlabel(r"$Re(\rho)$")
  ax.set_ylabel(r"$Im(\rho)$")
  ax.set_title(rf"$\beta J = {bJ}, N = {N}$")

  # Add a colorbar for the phase
  norm = Normalize(vmin=-np.pi, vmax=np.pi)
  sm = ScalarMappable(cmap="hsv", norm=norm)
  sm.set_array([])
  cbar = plt.colorbar(sm, ax=ax, orientation="vertical", label="Phase")

  # Overlay zeros
  zeros_real = [z.real for z in zeros]
  zeros_imag = [z.imag for z in zeros]
  ax.plot(zeros_real, zeros_imag, ms=7, marker="X", c="k", ls="None")
  plt.show()
```

NORMAL main 9 ex3q3_.py utf-8 python 99% 114:1