Ising Model

PHYS 481 -- Assignment 4

Patrick Harrison 30023631 & Craig Michie 30001523

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
import itertools
import timeit
from matplotlib.colors import ListedColormap
```

Introduction

Fundamentals of Conway's Game of Life are used to model physical systems such as the Ising Model. The Ising Model was sovled by Dr. Ernst Ising to model ferromagnetic and anti-ferromagnetic materials in 1 dimension. It using a simple 2 dimensional lattice structure contained of atoms which each is in a spin state. Using phsyical laws, the model can predict the second order phase transition occuring at the Curie temperatire for more than one dimension. Unlike the game of life however, the Ising Model is not completely deterministic. It has an element of randomness to it.

The Curie point/temperature, is a point where magnetic materials will undergo a sharp change in their magnetic properties. Below the Cuire point, ferromagnetic materials such as iron, will align each spin with their immediate domain (neighbours). These materials will have a coupling parameters $J > 0$. Anit-ferromagnetic materials will align themselves opposite to their neighbours and have a coupling parameter of $J < 0$.

Each atom can adopt two states $s = \{-1, 1\}$. The state a dipole will be in, depends on it's relative energy to it's new state.

$$p(\vec{x}) \propto \exp\left(-\frac{E(\vec{x})}{k_b T}\right)$$

We will start the process with a one dimensional model to exlpore the fundamentals of the Ising model and methods of simulating the model using python.

Write a function which will calculate all of the energy states and probabilities for an arbitrary number of independent dipoles. Tabulate the results for $N = 3$, $\beta = 0.1$ and $B = 2$

To find the energy of the independent dipoles, use:

$$E = -B \sum_k s_k$$

Here, E is the energy of the entire system, B is an external magnetic field, s_k is either a spin up or a spin down state. We will use the convention that:

$$\text{spin up} = 1$$
$$\text{spin down} = -1$$

For an arbitrary N, there will be 2^N unique permutations for the 1 dimensional lattice.

Since probability is a function of the total energy of the system we need to begin by definign a function to find the energy. The energy of a certian lattice will not depend on the interactions with it's self as we are assuming here that each dipole is independent of it's neighbours and that their respective spin states do not effect their neighbour's.

This model should only be phsyically correct looking at long time periods over many sweeps of the system.

In [2]:
```
def get_energy(lattice, B):
    '''
    calculates the energy states for the independent dipoles based on probability
        E=-B*sum[s_k]

    parameters:
        lattice -- lattice to calculate energy for
              B -- External Magnetic field
    Return:
        energy -- the energy of the system
    '''

    #count the number of spin up states in lattice
    spins = np.sum(lattice)

    #use the formula
    energy =  -1*B*spins

    return energy
```

In [3]:
```
#test the function
get_energy(lattice=[1,1,1],B=2)
```

Out[3]:
```
-6
```

We see that the energy takes on only certain values for a unique B. We can also see that using the convention for the spin states being $s = \{1, -1\}$ for up and down spin states respectivly, that the energy is dependent on the difference between the number of up and down states. Namely if we have all the same spins, then the magnitude of the energy will be higher.

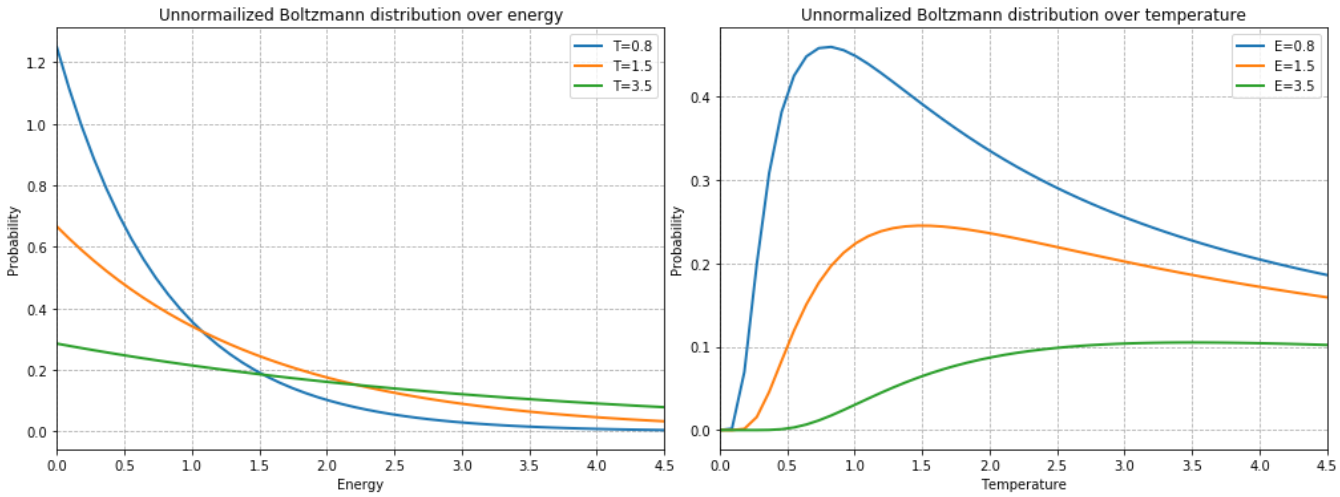
Now that we have the energy for the system, we can find the probability of that lattice's spin state occuring. The probability of a spin state occuring is given by the Boltzmann probability distribution.

$$p(\vec{x}) \propto \exp\left(-\frac{E(\vec{x})}{k_b T}\right)$$

This distribution depends on the energy of the system and the temperature for the system. We will define a varible $\beta \propto 1/T$. Just looking at the trends, we can drop the boltzmann constant k_B to make $\beta = 1/T$ for now.

We can see the general trend of the Boltzman Distribution below.

In [4]:
```
####
# Based From phys481_week04_isin_model-notes
####
f, (ax1, ax2) = plt.subplots(nrows=1,ncols=2,figsize=(14,5))
f.tight_layout()

ax1.set_xlabel("Energy")
ax1.set_ylabel("Probability")
ax1.set_title("Unnormailized Boltzmann distribution over energy")
ax1.set_xlim(0,4.5)
E = np.linspace(0, 4.5, num=50)

for i, T in enumerate([0.8, 1.5, 3.5]):
    ax1.plot(E, 1.0/T*np.exp(-E/T), "-", label="T={}".format(T), lw=2 )
ax1.legend()
ax1.grid(linestyle='--')



ax2.set_xlabel("Temperature")
ax2.set_ylabel("Probability")
ax2.set_title("Unnormalized Boltzmann distribution over temperature")
ax2.set_xlim(0,4.5)
T = np.linspace(0.000001, 4.5, num=50)

for i, E in enumerate([0.8, 1.5, 3.5]):
    ax2.plot(T, 1.0/T*np.exp(-E/T), "-", label="E={}".format(E), lw=2 )
ax2.legend()
plt.grid(linestyle='dashed')
#---------------------------------#
```



From the trends of the Boltzman Distributions, we see there will be a certain point at which the probability is maximized. Looking specifically at the Boltzman Distribution with temperature as the independent varible and setting energy as a constant, there is a point were at a certain temperature, the probability of finding that state is greatly enhanced before it tapers off. This is expecially prominant in the $E = 0.8$ case.

Now that we have energy, we can find the proabability following the boltzmann distribution for a certain state of the system. The partition function for this model is given by:

$$p_k = \frac{e^{-\beta E_k}}{Z}$$

In [5]:
```
def get_prob(lattice, beta, B):
    '''
    Calculates the un-normalized boltzmann probability of a spin state dedpendent on
    the lattice and beta

        p = [e^(-E*beta)]

    Parameters:
        lattice -- arbitrary sized binary state lattice (list)
           beta -- defined as 1/T for the system (float)
           B    -- external magnetic field as a (float)
    Returns:
        prob -- probability
    '''

    prob = (np.exp(-1*get_energy(lattice, B)*beta))

    return prob
```

In [6]:
```
#test the function
get_prob(lattice=[1,1,1], beta=0.1, B=2)
```

Out[6]:
```
1.822118800390509
```

Now that we have the probability, we also need to normailize it over all possible lattice's. We do this by dividing each probability by the sum of all the probabilities. getting the Z term in the partition function. more formally, we see $Z = \frac{1}{C}$ and

$$C = \sum_{i}^{N} \frac{1}{e^{\left(-\frac{E_i}{T}\right)}}$$

Where *i* is each possible lattice in the given length of lattice N

In [7]:
```
def normalize_prob(prob,prob_list):
    '''
    normalizes the probability by dividing by the sum of all probabilities

    Parameters:
        prob -- probability to normalize
        prob_list -- list of all probabilities

    Returns:
        prob/Z -- normalized probability
    '''

    Z=0
    for p in prob_list:
        Z += p
    return prob / Z
```

Putting everything we have done so far together, we are able to tabulate the results to look for comparisons and trends in the model.

In [8]:
```
def independent_state_analysis(N, beta, B):
    '''
    calculates energy and probabilites for for independent dipoles in 1 dimension of an
    arbitrary size

    parameters:
           N -- Arbitrary size for lattice
        beta -- defined as 1/T where T is the temp of the system
           B -- external magnetic field on system
    Returns:
            lattice_list -- all permutations of the arbitrary sized lattice
             energy_list -- list of the energy for each permutation
        probabiltiy_list -- list of the probabilities for each permutation
    '''

    #permutations
    lattice_list = list(itertools.product([1,-1],repeat=N))

    #initialize lists
    energy_list = []
    probability_list = []

    #calculate lists
    for lattice in lattice_list:
        energy_list.append(get_energy(lattice, B))
        probability_list.append(get_prob(lattice, beta, B))

        #normalize prob
        normalized = []
        for p in probability_list:
            normalized.append(normalize_prob(float(p),probability_list))


    return lattice_list, energy_list, normalized
```

This larger function is able to return the lattices used in the computation, the energy of each lattice, and the probability of that lattice appearing. Tabulating the results using a module pandas:

In [9]:
```
# this section of code creates a tabular form of the data
import pandas as pd

#Calculate need values
lattice_list, energy_list, probability_list = independent_state_analysis(N=3, beta=0.1, B=2)

#create data frame
df1 = pd.DataFrame({

    's1' : [state[0] for state in lattice_list],
    's2' : [state[1] for state in lattice_list],
    's3' : [state[2] for state in lattice_list],
    'energy' : [str(energy)+'*B' for energy in energy_list],
    'probability' : ['%.3f'%prob for prob in probability_list]

})
print(df1)
#-----------------------------------#
```

	s1	s2	s3	energy	probability
0	1	1	1	-6*B	0.215
1	1	1	-1	-2*B	0.144
2	1	-1	1	-2*B	0.144
3	1	-1	-1	2*B	0.096
4	-1	1	1	-2*B	0.144
5	-1	1	-1	2*B	0.096
6	-1	-1	1	2*B	0.096
7	-1	-1	-1	6*B	0.065

As stated above, we have seen explicitly now that the magnitude of the energy is directly porportional to the difference in spin up and spin down states. The probability however, is not directly porportional to the difference in spin states but is in fact dependent on the number of spin up states. This follows the model we are achiveing. The probabiltiy of a spin state in the up position should be high as we used a positive external magnetic field and the spins should want to align with the magnetic field.

As farther evidence, the probabilities and energys are the same if the number of spin up and spin down states in the permutaiton are the same. For the state $s = \{1, 1, -1\}$ it has a energy of $E = -2B$ and probabiltiy of $p = 0.144$ this is the same as if we had either states $s = \{-1, 1, 1\}$ or $s = \{1, -1, 1\}$. Again, this is only correct if the spin states in the lattice are not interacting with eachother.

Write a function which will return a Boltzmann random state for an arbitrary number N of independent dipoles. Plot the time required to produce the result as a function of N. Discuss your results.

Done above, the probability of having a state is modeled by the Boltzmann Distribution. We can now create a function that will pull a lattice structure with spin up and spin down states. The energy and position of these latticies will be calcualted also as above with a normalized proabability.

In [10]:
```
def rand_state(N,beta,B):
    '''
    returns a state based on the probability of it occuring.

    Parameters:
        N -- size of lattice
        beta -- defined as 1/T for the system
        B -- external magnetic field

    Returns:
        state -- a choosen state based on the probability of it being choosen
    '''

    #calculate needed values
    lattice_list, energy_list, probability_list = independent_state_analysis(N, beta, B)

    #Choose state
    index = np.random.choice(np.arange(2**N),p=probability_list)
    state = lattice_list[index]
    return state
```

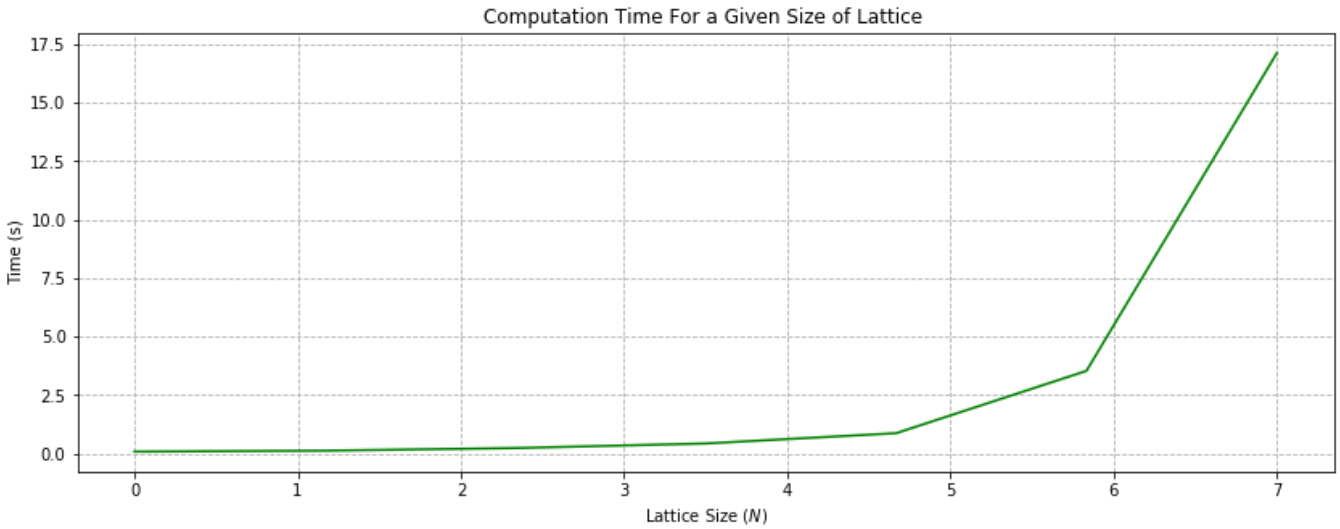
In [11]:
```
rand_state(N=4,beta=0.1,B=2)
```

Out[11]:
```
(1, 1, -1, 1)
```

We can now test to see how fast this function is for different N values. It is expected that as N gets larger, the time it takes to compute the function should increase exponentially.

In [12]:
```
time = []

trials=7
for trial in range(0,trials):
    s="rand_state(N="+str(trial)+",beta=0.1,B=2)"
    time.append(timeit.timeit(s, setup= 'from __main__ import rand_state',number=1000))

plt.figure(figsize=(14,5))
t = np.linspace(0,trials,trials)
plt.xlabel('Lattice Size ($N$)')
plt.ylabel('Time (s)')
plt.title('Computation Time For a Given Size of Lattice ')
plt.grid(linestyle='--')
plt.plot(t,time,color='g')
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x117cbfa90>]
```



As evident by the graph, this process will long be out of range of most computers after about a lattice above sizez 6 (long before avagadro's number 6.02×10^{23}). This is becuase of the computationally expensive permuations that have to be calculated. There will always be 2^N lattices to find the probabiltiy for and then choose based on the probabiltiy. Based on this, we will need to find a new solution if we want to be able to model a useful system.

Question: Use the metropolis algorithm to generate a sequence of states for a single dipole with $T = 300K$, $dE = 0.1, 1, 10eV$. Calculate the expected probabilities for each state and compare to the computed results for 1000 iterations.

As the Gibbs/Monte Carlo algorithm takes a very long time to compute for larger N values, we need to come up with a different solution. Unlike the previous algorithm which computed the next state like a cellular automata, This approach is bassed on Markov chains. we will not use the markov chains directly but use their solutions.

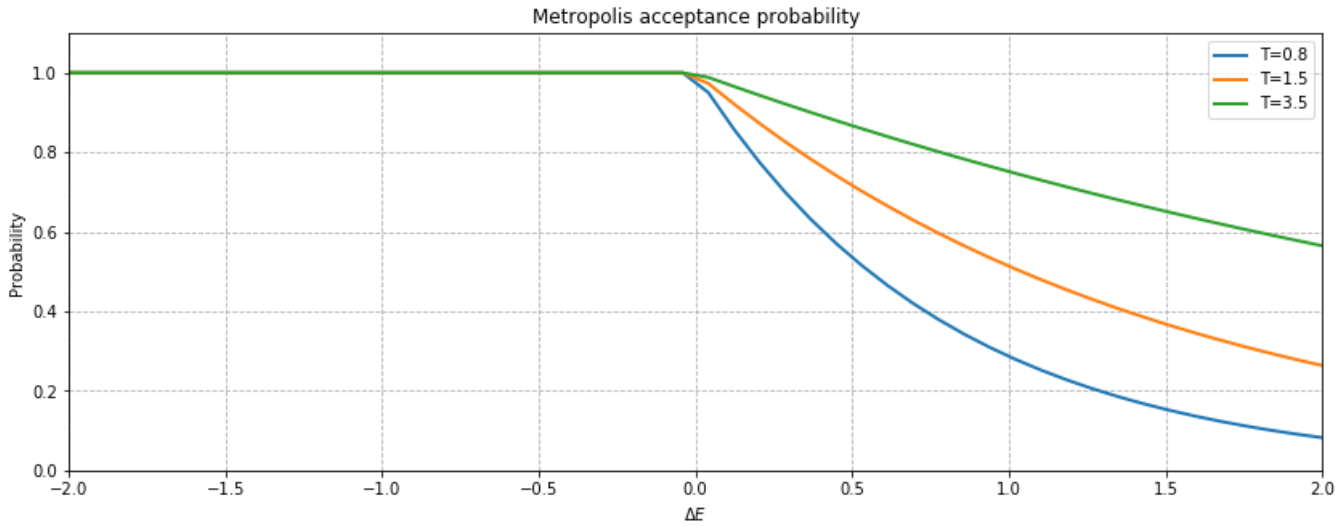
If the system is in a state with energy E_o and we flip the spin of a randomly selected element. This changes the energy of the system to E_1, the probability to accept the flip is given by:

$$p = min(1, exp(-\Delta E/T))$$

where $\Delta E = E_1 - E_o$

To visulaize the metropolis acceptance proabbility, create a plot for different system temperatures.

In [13]:
```
####
# Taken from the notes
####

plt.figure(figsize=(14,5))
ax = plt.subplot(xlabel="$\Delta E$", ylabel="Probability", ylim=(0,1.1), title="Metropolis acc
eptance probability")
ax.set_xlim(-2,2)
dE = np.linspace(-2, 2)

for i, T in enumerate([0.8, 1.5, 3.5]):
    ax.plot(dE, np.minimum(1, np.exp(-dE/T)), "-", label="T={}".format(T), lw=2)
ax.legend()
plt.grid(linestyle='--')
#----------------------#
```



It is clear from the graph above that the proabability of a switch occuring is $p\,(\Delta E < 0) = 1$ (new state E_1 is a lower energy than E_o) so the spin will always flip if the change of energy is negative (energy is released from system and total energy goes down). This agrees with our physical intuition that things will move from an excited state (high energy) to a lower state (low energy).

Now we can test this numerically by creating a function that chooses a state based on this probability. Since we are still assuming independent dipoles, we will only look at $N = 1$ then we can expand it over a loop later. we can set up a function to take the probabiltiy for the dipole.

The metropolis algorithm states that:

1. start with a random configuration of states
2. generate a new state by flipping 1 random state
3. calcualte the energy difference of the trial state then the initial state
4. if $E_1 \leq E_o$ accept flip with probability $p = 1$
5. if $E_1 > E_o$ accept flip with probability $p = \exp\left(-\frac{dE}{k_B T}\right)$

We still use the same energy equation as before.

$$E = -B\sum_k s_k$$

Since we are using one dipole, we will say the the energy is either

$$E = -B \qquad\qquad E = B$$
$$\text{for spin up state} \qquad \text{for spin down state}$$

From this we know that the spin up state is always of a lower energy as long as the magnetic field is positive (spins will align with the magentic field). Following the metropolis algorithm for a single dipole,

- The dipole will always be choosen as the trial flip so it will not matter what state we started in
- If the dipole is in a spin down state, it will always go to a spin up state
- if the dipole is in a spin up state, it will only flip based on the boltzmann distribution

The funciton below takes this into account and modifies the metropolis algorithm for a single non-interacting dipole.

In [14]:
```
def metropolis_single_dipole(T,dE):
    '''
    gives the proabbiltiy and lattice permuations for the metropolis algorithm on one dipole

     Parameters:
        T -- temperature
        dE -- change in energy if spin flips as a list of values

    Returns:
        lattice_list -- all permutations of the arbitrary sized lattice
        probabiltiy_list -- list of the probabilities for each permutation
    '''

    #initialize lists
    probability_list = []

    #calculate lists
    for n in range(1):
        probability_list.append(min(1, np.exp(-dE / (T)))) #takes 1 if dE<0
        probability_list.append(1-min(1, np.exp(-dE / T)))


    return [1,-1], probability_list
```

In [15]:
```
l_list, p_list = metropolis_single_dipole(T=300,dE=10)
print(l_list,p_list)
```

```
[1, -1] [0.9672161004820059, 0.0327838995179941]
```

we now have a function returning the single dipole state and their respective probabiltiy of occuring in the metropolis algorithm.

In [16]:
```
def metropolis_chooser(T,dE):
    '''
    returns a state based on the probability of it occuring.

    Parameters:
        T -- temperature
        dE -- change in energy if spin flips

    Returns:
        state -- a choosen state based on the probability of it being choosen
    '''

    #calculate needed values
    lattice_list, probability_list = metropolis_single_dipole(T, dE)

    #use an index to get the state out of the lattice list
    index = np.random.choice(np.arange(2),p=probability_list)
    state = lattice_list[index]

    return state
```

In [17]:
```
metropolis_chooser(T=300, dE=100)
```

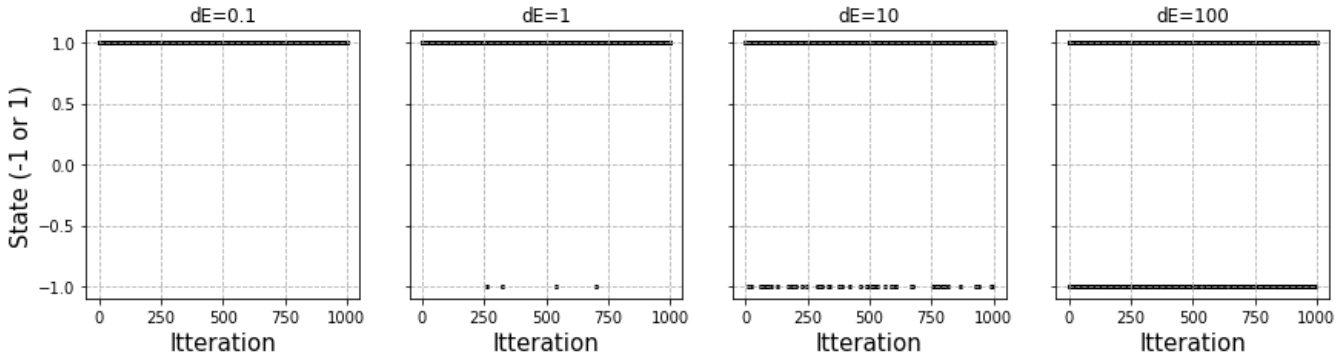
Out[17]:
```
-1
```

we can now define a function based on the independent single dipole functions to get the a state for $N = 1000$. Doing this would been very computationaly expensive based on the Gibbs/Montecarlo simulation. Using the metropolis simulation, it becomes very quick.

In [18]:
```
def get_state(T,dE,N=1000):
    '''
    returns a list of length 10000 for that number of itterations over the
    metropolis chooser function.
    '''
    dE_list=[]
    for itterations in range(N):
        dE_list.append(metropolis_chooser(T, dE))

    return dE_list
```

looking at different changes in energy dE we can look at the number of spin up and spin down states as below

In [19]:
```
#set up plots
f, axarr = plt.subplots(nrows=1,ncols=4,figsize=(12,3),sharey=True)
f.tight_layout()

xlabel='Itteration'
ylabel='State (-1 or 1)'

x=np.linspace(0,1000,1000)

#plot different dE's
for n, dE in enumerate([0.1,1,10,100]):
    title='dE='+str(dE)
    axarr[n].set_title(title)
    axarr[n].set_ylim(-1.1,1.1)
    axarr[n].set_xlabel(xlabel, fontsize=15)
    axarr[0].set_ylabel(ylabel, fontsize=15)
    axarr[n].grid(linestyle='--')
    axarr[n].scatter(x, get_state(T=300, dE=dE, N=1000), color='k',s=4)
```



The scatter plots above show the number of up $(\text{state} = 1)$ or down $(\text{state} = -1)$ for 1000 itterations. This is what is expected from the calculated probabilities. The Higher energy system has a higher chance in a spin up state for that singlular dipole. More quantitatlivly, we can say that the probability is the ratio of the empircal number of times the up state occured divided by the number of times it was tried.

$$\text{Probability} = \frac{\text{Number of up states}}{\text{Number of itterations}}$$

In [20]:
```
def empirical_prob(occurance,itter):
    '''
    uses that probabiltiy is
        probabiltiy=occurance/itterations
    '''
    return occurance/itter
```

In [21]:
```
theoretical=[]
for dE in [0.1,1,10]:
    theoretical.append(min(1, np.exp(-dE / 300)))



for n, dE in enumerate([0.1,1,10]):
    occurance = get_state(T=300, dE=dE).count(1)
    print('The empirical probability of a state appearing in a spin up for dE='+str(dE),'is:',e
mpirical_prob(occurance,10000))
    print('The theoretical probability of a state appearing in spin up for dE='+str(dE),'is:','
%.3f'%theoretical[n])
    print('The occurance of getting a spin down state was:',occurance)
    print()
```

```
The empirical probability of a state appearing in a spin up for dE=0.1 is: 0.1
The theoretical probability of a state appearing in spin up for dE=0.1 is: 1.000
The occurance of getting a spin down state was: 1000

The empirical probability of a state appearing in a spin up for dE=1 is: 0.0993
The theoretical probability of a state appearing in spin up for dE=1 is: 0.997
The occurance of getting a spin down state was: 993

The empirical probability of a state appearing in a spin up for dE=10 is: 0.0972
The theoretical probability of a state appearing in spin up for dE=10 is: 0.967
The occurance of getting a spin down state was: 972
```

These results are expected as the number of ones divided by the total number of itterations give the probability of getting an up state. These are just approxiamtions to the caluclated value as we can not run the simulation an infinite number of times. This means that a likely result of getting the emprical proabbility for $dE = 0.1$ is $p(dE = 0.1) = 0$ if this was run, we should see that the numbers equal to the caluclated proabbilities before.

Write python code to evolve a system of N dipoles in a ring (wrap-around boundary conditions) using the Metropolis algorithm for arbitrary N , B , T , J .

We will use an arbitrary constant J for the ferror/anti-ferromagnetic materials as well as an arbitrary external magnetic field B. We need to initialise the system by creating a grid of binary states.

Initialized with a probability of being an up spin of $p(\text{up}) = 0.5$. Physically, this number will change based on it's temperature and the energy in the system.

In [22]:
```
def create_lattice(row_size,col_size,p):
    '''
    creates an empty dimensional lattice structure as a numpy array

    parameters:
        row_size -- number of columns for the lattice structure
        col_size -- number of rows for the lattice structure
          p   -- probability that each element is in an up state.
    returns:
        lattice -- dimensional array of arbitrary size with -1 and 1 values
    '''

    lattice=np.random.choice(a=[1,-1],size=(row_size,col_size),p=[p,1-p])

    return lattice
```

In [23]:
```
#sample grid to test wiht p=0.5 for up and down spins
lattice=create_lattice(row_size=1, col_size=49, p=0.5)
print(lattice)
#show lattice as colour-coded grid, blue := spin down, #red := spin up
plt.figure(figsize=(12,5))
plt.axis('off')
plt.imshow(lattice,cmap='jet')
```

```
[[-1  1 -1 -1  1 -1 -1  1 -1 -1  1 -1 -1 -1 -1  1  1  1  1  1  1  1  1 -1
  -1 -1  1 -1  1 -1  1 -1  1 -1  1 -1 -1 -1 -1  1  1  1  1  1 -1  1 -1  1
   1]]
```

Out[23]: <matplotlib.image.AxesImage at 0x11837bba8>

Now that we have an arbitrary shaped lattice structure filled with assigned 1 and -1 values we can define those values

$$1 := \text{dipole in up sate}$$

$$\text{-1} := \text{dipole in down sate}$$

We can then create a function which will flip the values for a probability given by the botlzmann distribution. The energy of the system is:

$$E(S_1, \ldots, S_N) = -J \sum_{i,j} S_i S_j - B \sum_k S_k$$

We know that probabiltiy is boltzmann distributed.

$$p(\mathbf{x}) \propto \exp(-\frac{E(\mathbf{x})}{kT})$$

and the relative probability of two different states is

$$r = \frac{p_f}{p_i} = e^{(E_f - Ei)/kT}$$

If we only consider the effect of a single flip S_n, then

$$r = e^{-2S_n(Jf+B)/kt}$$

where all the information about neighbors is gathered into a term

$$f = S_{i+1,j} + S_{i-1,j} + S_{i,j+1} + S_{i,j-1}$$

with only five possible values $(0, \pm 2, \pm 4)$. With two possible values of S_n there are only 10 possible values for r, so we could compute these once at the beginning of the program in order to avoid re-calculating the same computationally expensive exponentials over and over.

Here we only have a 1 dimensional case so

$$f = S_{i+1} + S_{i-1}$$

this we only have three possible values for f as $(0, \pm 2)$ and six total possible values.

In [24]:
```
def get_r(nearspins, J, B, K, T):
    '''computes the relative probabiltiy of two different states for a single flip

    Paramters:
        J -- coupling constant (flaot)
        B -- external magnetic field (float)
        K -- boltzmann constant  (float)
        T -- temperature (float)
    returns:
        r -- list of possible values for the relative probability
    '''
    r={}
    for state in [1,-1]:
        r_prime={}
        for nearspin in nearspins:
            r_prime[nearspin]=(np.exp(-2*state*(J*nearspin+B)/(K*T)))
        r[state]=r_prime

    return r
```

In [25]:
```
#test the function for arbitrary parameters
r=get_r(nearspins=[0,2,-2], J=1, B=5, K=8.62E-5, T=300) ;print(r)
```

```
{1: {0: 1.1464762961501898e-168, 2: 7.6403123189321485e-236, -2: 1.7203588580762082e-101}, -1:
{0: 8.722378328779672e+167, 2: 1.3088470186252353e+235, -2: 5.812740727351794e+100}}
```

In [26]:
```
def flip_spin(state, r_catalog, nearspins, J, B, K, T):
    '''
    given spins around a cell for certain phsyical parameters and will compute based on a chanc
e
    whether the spin will flip or not.

    parameters:
        state -- either spin up or spin down (1 or -1)
        nearspins -- the number spins around the state we are looking at
        J -- coupling strength
        B -- external magnetic field
        T -- temperature

    Returns:
        state -- new state of spin
        energy -- energy caluclated for the spin off near spins and temperature
    '''
    dE= (-J*nearspins+B*state)-(-J*nearspins-B*state) #new energy - initial energy
    if dE < 0:
        state *= -1

    elif np.random.rand() < r_catalog[state[0]][nearspins[0]]:
        state *= -1


    return state , dE
```

This is the requiered funciton for calculating whether a given spin will flip or not. we can now put this in a function that will choose a random atom/cell and count the nearspins then call the function above to see if

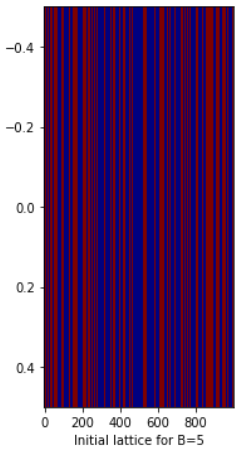
In [27]:
```
def simple_step_1D(grid, r_catalog, nsites=1, J=1.0, B=0.0, K=1, T=0.5):
    '''
    Metropolis algorithm step.

    Function will step the model based on the immidiate domain of the cell and flip if needed.
 Chooses
    random position using numpy's random.rand() function which can specify the number of sites
 to choose
    multiplying it by the grid size in that dimension lets it be the right size.

    To count the neighbouring spins, use the remainder to wrap around edges of the grid. Only l
ooking
    at the cells directly above,below,to the left and to the right.

    parameters:
        grid -- initial grid of truth values for the up and down spin states
        nsites -- number of sites to choose a move from for each step
        J -- coupling parameter (J>0 for ferromagnetic , J<0 for anti-ferromagnetic)
        B -- external magnetic field
        T -- temperature

    Returns:
        grid.copy() -- changed grid after doing step for above parameters as a shallow copy so
 we do not
                       lose the intial grid.
    '''

    for site in range(nsites):
        #get shape of grid
        x_size, y_size = grid.shape

        #Pick random sites (number specified in paramters)
        x_pos = np.random.randint(0,x_size,nsites)
        y_pos = np.random.randint(0,y_size,nsites)

        # count neighboring spins, wrap around the edges

        nearspins = grid[(x_pos+1)%x_size,y_pos] + grid[x_pos-1,y_pos]
        nearspins
        #computes the flip of spin using nested function
        grid[x_pos,y_pos], energy = flip_spin(grid[x_pos,y_pos],r_catalog, nearspins, J, B, K,
T)


        #fraction of cell that are in up state
        frac = np.count_nonzero(grid == 1)/(x_size*y_size)

    #returns a shallow copy of the grid so we do not lose initial grid
    return grid.copy(), energy[0], frac
```

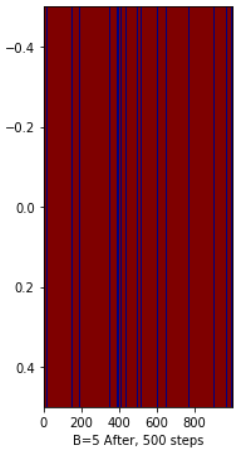
We can now evolve the system using the simple_step function over a number of steps and plot it.

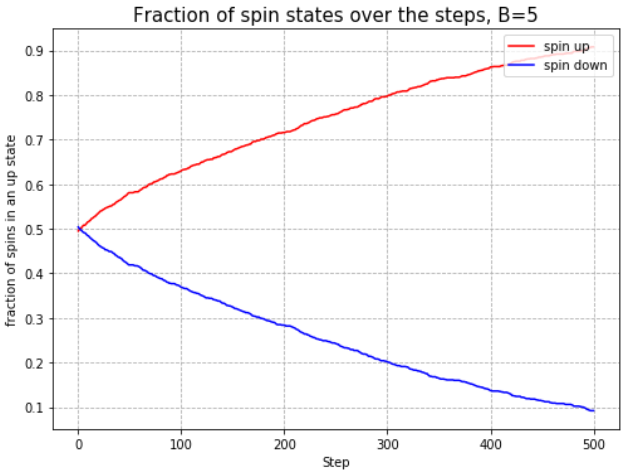
In [28]:
```
def evolve_ring(lattice, spins, B, T, J, K, steps):
    '''
    evolves the 1D ring using the metropolis algorithm for a number of steps

    Parametes:
        lattice -- intial lattice to evolve over
        B -- given B field to compute the Ising model over
        T -- tempertaure of the system as a float value
        J -- coupling constant
        K -- porportinality constant (boltzmann constant) as a float value
        steps -- number of steps
    '''


    f, axarr = plt.subplots(nrows=1, ncols=3, figsize=(15,5),gridspec_kw={'width_ratios':[1, 1,
3]})
    f.tight_layout()
    f.subplots_adjust(wspace=0.55,hspace=0.25)


    #initialize fraction lists
    frac_up=[]
    frac_down=[]

    #r catalog
    r_catalog = get_r(spins, J=J, B=B, K=K, T=T)



    #create and show initial lattice
    title = 'Initial lattice for B='+str(B)
    axarr[0].set_xlabel(title,fontsize=10)
    axarr[0].imshow(lattice,cmap='jet',aspect='auto')

    #evolve
    for i in range(steps):
        lattice = simple_step_1D(lattice, r_catalog=r_catalog, nsites=1, J=J, B=B, T=T)[0]
        frac_up.append(simple_step_1D(lattice, r_catalog=r_catalog, nsites=1, J=J, B=B, T=T)[2
])
        frac_down.append(1-simple_step_1D(lattice, r_catalog=r_catalog, nsites=1, J=J, B=B, T=T
)[2])


    #plot the lattice after total steps
    title = 'B='+str(B)+ ' After, '+str(steps)+' steps'
    axarr[1].set_xlabel(title,fontsize= 10)
    axarr[1].imshow(lattice,cmap='jet',aspect='auto')


    #plot fraction of cells over steps
    axarr[2].set_title('Fraction of spin states over the steps, B='+str(B), fontsize=15)
    axarr[2].set_xlabel('Step')
    axarr[2].set_ylabel('fraction of spins in an up state')
    axarr[2].plot(np.linspace(0,steps,steps),frac_up,color='r',label='spin up')
    axarr[2].plot(np.linspace(0,steps,steps),frac_down,color='b',label='spin down')
    axarr[2].legend(loc=1)
    axarr[2].grid(linestyle='--')
```

In [29]:
```
#create lattice
lattice = create_lattice(1,1000,0.5)

evolve_ring(lattice=lattice, spins=[0,2,-2],B=5, T=100, J=0, K=8.62E-5, steps=500)
```

```
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:16: RuntimeWarning: overflow enco
untered in exp
  app.launch_new_instance()
```







As a bug in the code, if the image plot of the matrix will go solid blue in the event of all atoms in the lattice being in the same state, even that state is spin up (red).

Write python code to implement the Ising model on a 51x49 grid for arbitrary J and B. Use 200 sweeps to thermalize and assume that the grid is randomized after 20 sweeps.

Now we can expand to a 2 dimensional grid. This means we have 10 values for the relative energy rather than 6.

$$f = S_{i+1,j} + S_{i-1,j} + S_{i,j+1} + S_{i,j-1}$$

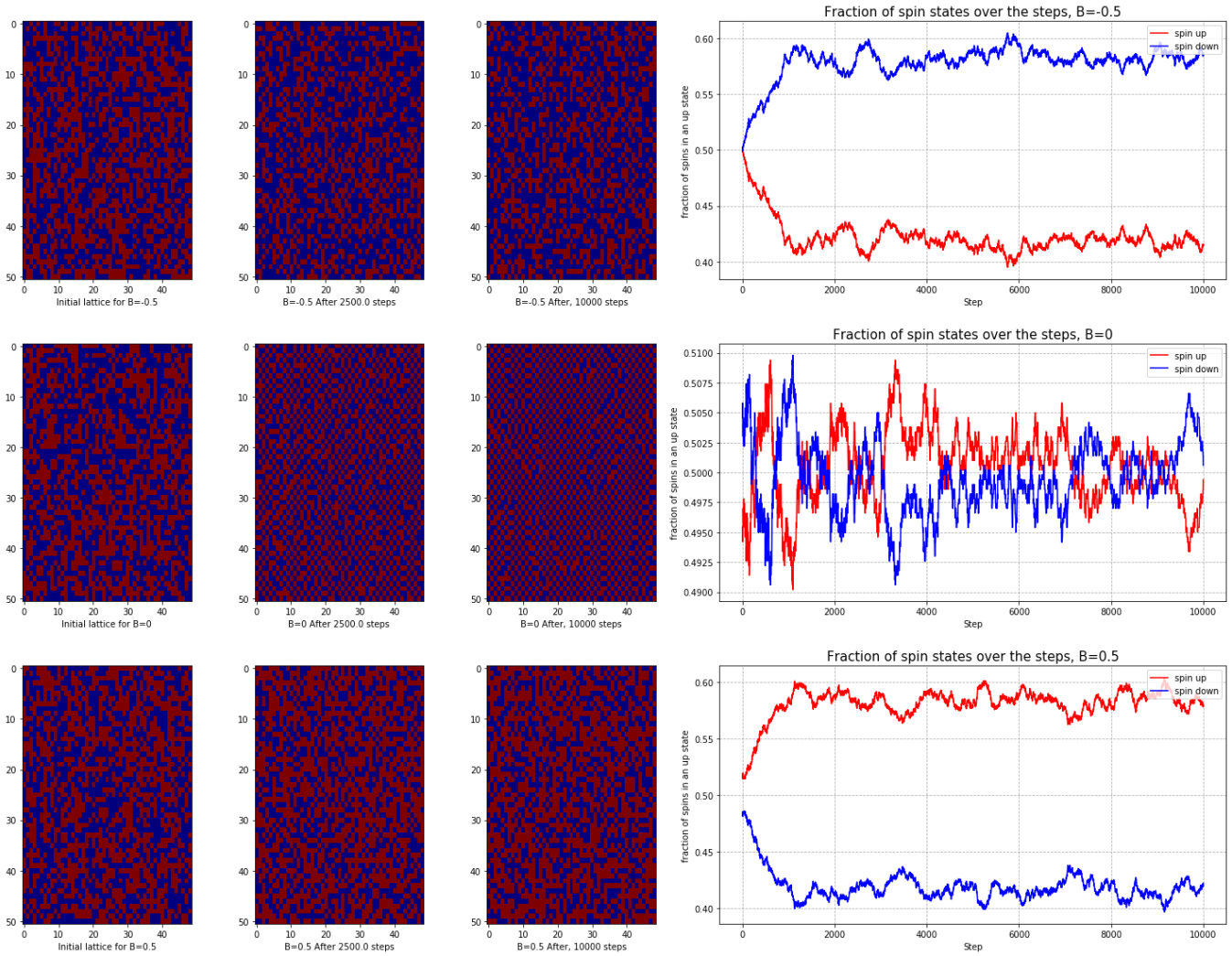
we will redefine the simple_step_2D to take all nearspins not just neaspins in the x-direction.

In [30]:
```
def simple_step_2D(grid, r_catalog, nsites=1, J=1.0, B=0.0, K=1, T=0.5):
    '''
    Metropolis algorithm step in 2 dimensions.

    Function will step the model based on the immidiate domain of the cell and flip if needed.
 Chooses
    random position using numpy's random.rand() function which can specify the number of sites
 to choose
    multiplying it by the grid size in that dimension lets it be the right size.

    To count the neighbouring spins, use the remainder to wrap around edges of the grid. Only l
ooking
    at the cells directly above,below,to the left and to the right.

    parameters:
        grid -- initial grid of truth values for the up and down spin states
        nsites -- number of sites to choose a move from for each step
        J -- coupling parameter (J>0 for ferromagnetic , J<0 for anti-ferromagnetic)
        B -- external magnetic field
        T -- temperature

    Returns:
        grid.copy() -- changed grid after doing step for above parameters as a shallow copy so
 we do not
                       lose the intial grid.
    '''

    for site in range(nsites):
        #get shape of grid
        x_size, y_size = grid.shape

        #Pick random sites (number specified in paramters)
        x_pos = np.random.randint(0,x_size,nsites)
        y_pos = np.random.randint(0,y_size,nsites)

        # count neighboring spins, wrap around the edges

        nearspins = grid[x_pos,(y_pos+1)%y_size] + grid[x_pos,y_pos-1] + grid[(x_pos+1)%x_size,
y_pos] + grid[x_pos-1,y_pos]

        #computes the flip of spin using nested function
        grid[x_pos,y_pos], energy = flip_spin(grid[x_pos,y_pos],r_catalog, nearspins, J, B, K,
T)

        #fraction of cell that are in up state
        frac = np.count_nonzero(grid == 1)/(x_size*y_size)

    #returns a shallow copy of the grid so we do not lose initial grid
    return grid.copy(), energy[0], frac
```

In [31]:
```
def plot_it_2D(xsize, ysize, spins, B_fields, prob, temp, K, J, steps):
    '''
    2D plotting:
    calculates and plots the ising model over a given number of steps. Shows the Ising model wh
en done 1/4 steps
    and once done all steps along with the intial grid. It will also plot the fraction of spin
 up and spin down cells
    over the steps

    parameters:
        xsize -- x dimension of the lattice as an int
        ysize -- y dimension of the lattice as an int
        spins -- list of possible sum of nearpsins
        B_fields -- given B fields to compute the Ising model over as a list
        prob -- inital probability of each atom in the lattice being in the up state fractin be
tween 1 and 0
        temp -- tempertaure of the system as a float value
        k -- porportinality constant as a float value
        steps -- number of steps (int, must be divisible by 4 evenly)

    Returns:
        none
        plots using matplotlib
    '''

    #set up plots
    f, axarr = plt.subplots(nrows=len(B_fields),ncols=4,figsize=(20,len(B_fields)*5), \
                            gridspec_kw = {'width_ratios':[1, 1, 1, 3]})
    f.tight_layout()
    f.subplots_adjust(wspace=0.25,hspace=0.25)

    #for each B in B_fields plot it
    for n, B in enumerate(B_fields):

        #create and show initial lattice
        lattice = create_lattice(xsize,ysize,prob)
        title = 'Initial lattice for B='+str(B)
        axarr[n][0].set_xlabel(title,fontsize=10)
        axarr[n][0].imshow(lattice,cmap='jet',aspect='auto')

        #initialize fraction lists
        frac_up=[]
        frac_down=[]

        #r catalog
        r_catalog = get_r(spins, J=J, B=B, K=K, T=temp)

        #calculate for the first quater steps
        for i in range (int(steps/4)):
            lattice = simple_step_2D(lattice,r_catalog=r_catalog, nsites=1, J=J, B=B, K=K, T=te
mp)[0]
            frac_up.append(simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=J, B=B, K=K
, T=temp)[2])
            frac_down.append(1-simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=J, B=B,
K=K, T=temp)[2])


        #plot the lattice after 1/4 total steps
        title = 'B='+str(B)+' After '+str(steps/4)+' steps'
        axarr[n][1].set_xlabel(title, fontsize=10)
        axarr[n][1].imshow(lattice,cmap='jet',aspect='auto')


        #calculate for the last 3 quater steps
        for i in range(int(3*steps/4)):
            lattice = simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=J, B=B, T=temp)[
0]
            frac_up.append(simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=J, B=B, T=t
emp)[2])
            frac_down.append(1-simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=J, B=B,
T=temp)[2])

        #plot the lattice after total steps
        title = 'B='+str(B)+ ' After, '+str(steps)+' steps'
        axarr[n][2].set_xlabel(title,fontsize= 10)
        axarr[n][2].imshow(lattice,cmap='jet',aspect='auto')
```

```
        #plot fraction of cells over steps
        title = 'Fraction of spin states over the steps, B='+str(B)
        axarr[n][3].set_title(title, fontsize=15)
        axarr[n][3].set_xlabel('Step')
        axarr[n][3].set_ylabel('fraction of spins in an up state')
        axarr[n][3].plot(np.linspace(0,steps,steps),frac_up,color='r',label='spin up')
        axarr[n][3].plot(np.linspace(0,steps,steps),frac_down,color='b',label='spin down')
        axarr[n][3].legend(loc=1)
        axarr[n][3].grid(linestyle='--')
```

In [32]:
```
temp=900000
K=8.617E-5
J=-100
plot_it_2D(xsize=51, ysize=49, spins=(0,2,-2,4,-4), B_fields=[-0.5,0,0.5], prob=0.5, temp=temp,
K=K, J=J, steps=10000)
```



We see some intresting results for the different magnetic fields. for a negative J value (neighbouring spins will oppose eachother) and no external magnetic field, we get a checkerboard pattern. There is nothing for the spins to want to align with up or down externally so they oppose eachother. Recall this is still omitting the diagonal neighbours. for non-zero values of B and a negative J value, we get a moe apparent random distribution of spin up and down state but will have more spins in up or down depending on the positive or negative magentic field respectivly.

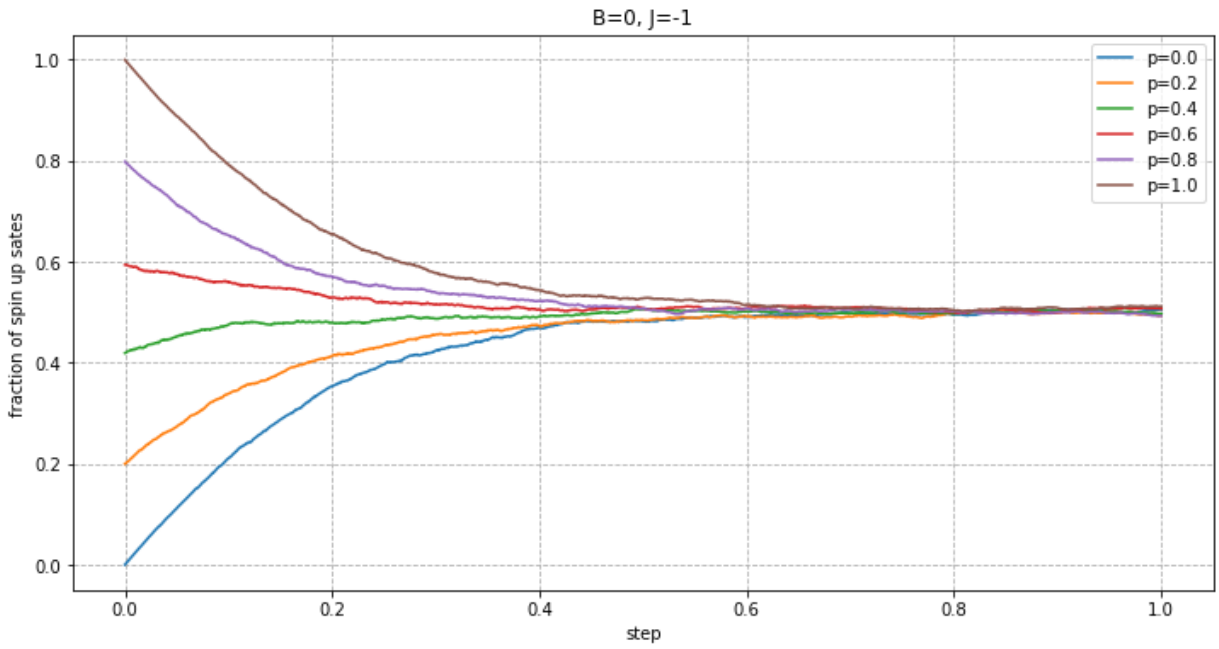
The fact that the external magentic fields do have a total effect to push the spins into the same direction is a product of the hystesis curve. the fraction of spin states flattens out before it gets 60%.

It is unfortunate how long the code was for this function. Having more time, I may have been able to find a way to cut it down.

Seen below, it wont matter the starting porability of the grid. The fraction of spin up and spin down cells for no external magnetic field and a negative coupling constant J, will always end up with half the spins in an up state and half in a down state.

In [33]:
```
r_catalog = get_r([0,2,-2,4,-4], J=-1, B=0, K=8.62E-5, T=300)
plt.figure(figsize=(12,6))

for prob in range(0,12,2):
    frac_up=[]
    lattice = create_lattice(51,49,prob/10)
    for i in range(3000):
        lattice = simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=-1, B=0, K=8.61E-5,
T=300)[0]
        frac_up.append(simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=-1, B=0, K=8.61
E-5, T=300)[2])
    plt.title('B='+str(0)+', J='+str(-1))
    plt.ylabel('fraction of spin up sates')
    plt.xlabel('step')
    plt.grid(linestyle='--')
    plt.plot(np.linspace(0,1,3000),frac_up,label='p='+str(prob/10))
plt.legend()
```

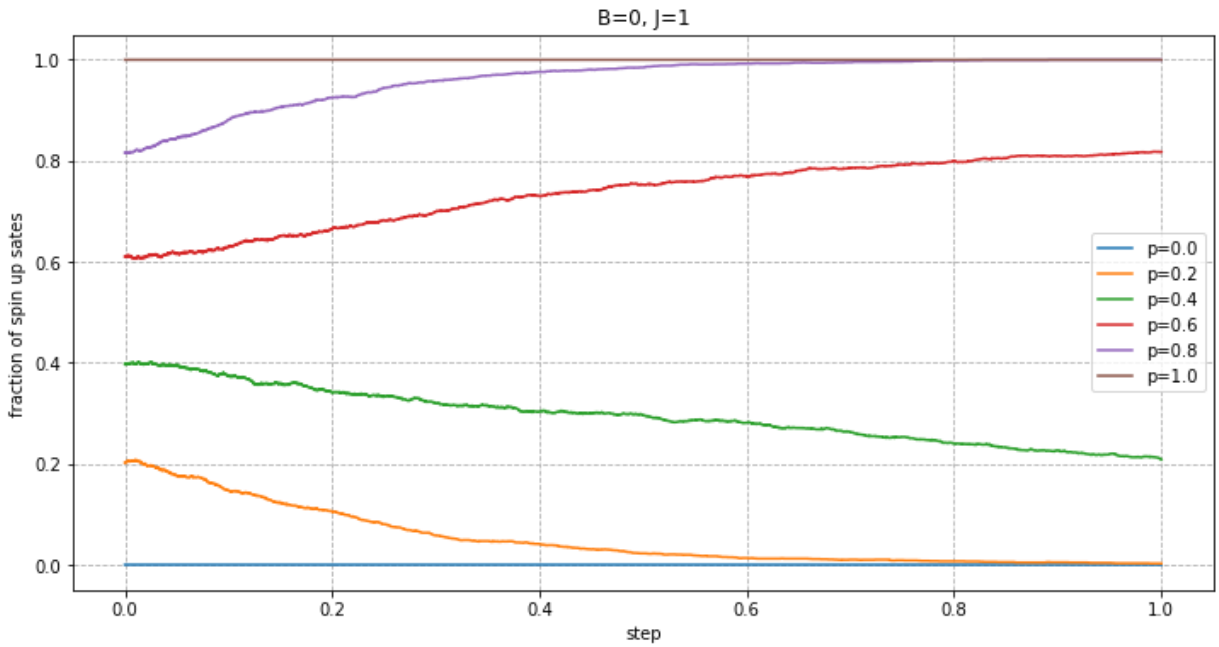
Out[33]:
```
<matplotlib.legend.Legend at 0x1186e3b38>
```



For $J > 0$ the spin states will trend towards the majority. Either spin up or spin down will take over.

In [34]:
```
r_catalog = get_r([0,2,-2,4,-4], J=1, B=0, K=8.62E-5, T=300)
plt.figure(figsize=(12,6))

for prob in range(0,12,2):
    frac_up=[]
    lattice = create_lattice(51,49,prob/10)
    for i in range(10000):
        lattice = simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=1, B=0, K=8.61E-5, T
=300)[0]
        frac_up.append(simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=1, B=0, K=8.61E
-5, T=300)[2])
    plt.title('B='+str(0)+', J='+str(1))
    plt.ylabel('fraction of spin up sates')
    plt.xlabel('step')
    plt.grid(linestyle='--')
    plt.plot(np.linspace(0,1,10000),frac_up,label='p='+str(prob/10))
plt.legend()
```

Out[34]:
```
<matplotlib.legend.Legend at 0x119ee7128>
```



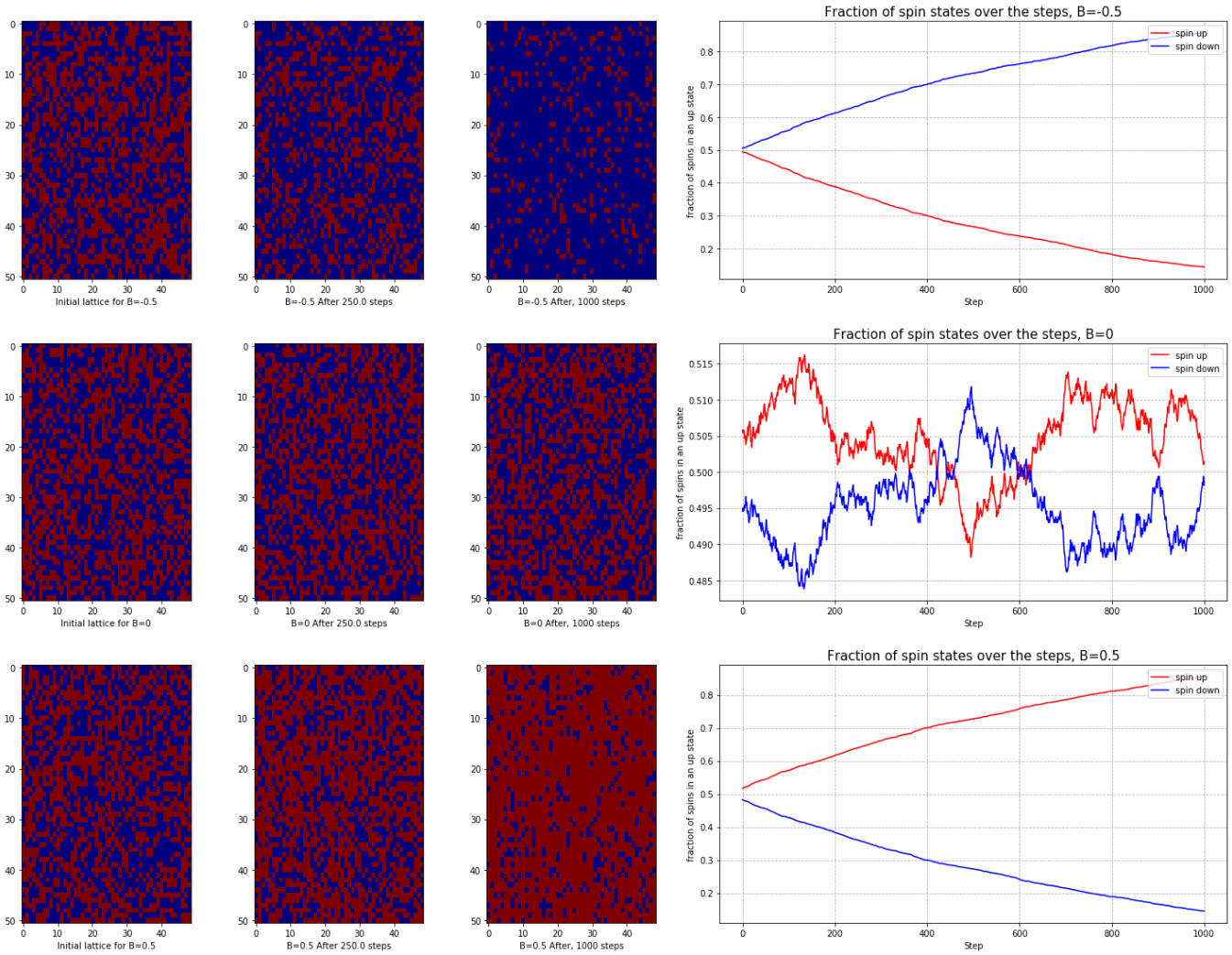
Produce plots for J=(kB T), B=0, B=+0.5, B=-0.5

In [36]:
```
def plot_it_2D_J(xsize, ysize, spins, B_fields, prob, temp, K, steps):
    '''
    2D plotting:
    calculates and plots the ising model over a given number of steps. Shows the Ising model wh
en done 1/4 steps
    and once done all steps along with the intial grid. It will also plot the fraction of spin
 up and spin down cells
    over the steps

    parameters:
        xsize -- x dimension of the lattice as an int
        ysize -- y dimension of the lattice as an int
        spins -- list of possible sum of nearpsins
        B_fields -- given B fields to compute the Ising model over as a list
        prob -- inital probability of each atom in the lattice being in the up state fractin be
tween 1 and 0
        temp -- tempertaure of the system as a float value
        k -- porportinality constant as a float value
        steps -- number of steps (int, must be divisible by 4 evenly)

    Returns:
        none
        plots using matplotlib
    '''

    #set up plots
    f, axarr = plt.subplots(nrows=len(B_fields),ncols=4,figsize=(20,len(B_fields)*5), \
                            gridspec_kw = {'width_ratios':[1, 1, 1, 3]})
    f.tight_layout()
    f.subplots_adjust(wspace=0.25,hspace=0.25)

    #for each B in B_fields plot it
    for n, B in enumerate(B_fields):

        #create and show initial lattice
        lattice = create_lattice(xsize,ysize,prob)
        title = 'Initial lattice for B='+str(B)
        axarr[n][0].set_xlabel(title,fontsize=10)
        axarr[n][0].imshow(lattice,cmap='jet',aspect='auto')

        #initialize fraction lists
        frac_up=[]
        frac_down=[]

        #r catalog
        r_catalog = get_r(spins, J=K*T*B, B=B, K=K, T=temp)

        #calculate for the first quater steps
        for i in range (int(steps/4)):
            lattice = simple_step_2D(lattice,r_catalog=r_catalog, nsites=1, J=K*T, B=B, K=K, T=
temp)[0]
            frac_up.append(simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=K*T, B=B, K
=K, T=temp)[2])
            frac_down.append(1-simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=K*T, B=
B, K=K, T=temp)[2])


        #plot the lattice after 1/4 total steps
        title = 'B='+str(B)+' After '+str(steps/4)+' steps'
        axarr[n][1].set_xlabel(title, fontsize=10)
        axarr[n][1].imshow(lattice,cmap='jet',aspect='auto')


        #calculate for the last 3 quater steps
        for i in range(int(3*steps/4)):
            lattice = simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=K*T, B=B, T=temp
)[0]
            frac_up.append(simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=K*T, B=B, T
=temp)[2])
            frac_down.append(1-simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=K*T, B=
B, T=temp)[2])

        #plot the lattice after total steps
        title = 'B='+str(B)+ ' After, '+str(steps)+' steps'
        axarr[n][2].set_xlabel(title,fontsize= 10)
        axarr[n][2].imshow(lattice,cmap='jet',aspect='auto')
```

```
        #plot fraction of cells over steps
        title = 'Fraction of spin states over the steps, B='+str(B)
        axarr[n][3].set_title(title, fontsize=15)
        axarr[n][3].set_xlabel('Step')
        axarr[n][3].set_ylabel('fraction of spins in an up state')
        axarr[n][3].plot(np.linspace(0,steps,steps),frac_up,color='r',label='spin up')
        axarr[n][3].plot(np.linspace(0,steps,steps),frac_down,color='b',label='spin down')
        axarr[n][3].legend(loc=1)
        axarr[n][3].grid(linestyle='--')
```

In [37]:
```
temp=1
K=8.617E-5
plot_it_2D_J(xsize=51, ysize=49, spins=(0,2,-2,4,-4), B_fields=[-0.5,0,0.5], prob=0.5, temp=tem
p, K=K, steps=1000)
```

```
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:16: RuntimeWarning: overflow enco
untered in exp
  app.launch_new_instance()
```



For B=0 and J/(T kB)=range(0.0, 0.6, 21), calculate and plot the average magnetization M versus J/TkB.

The magnitization is given by

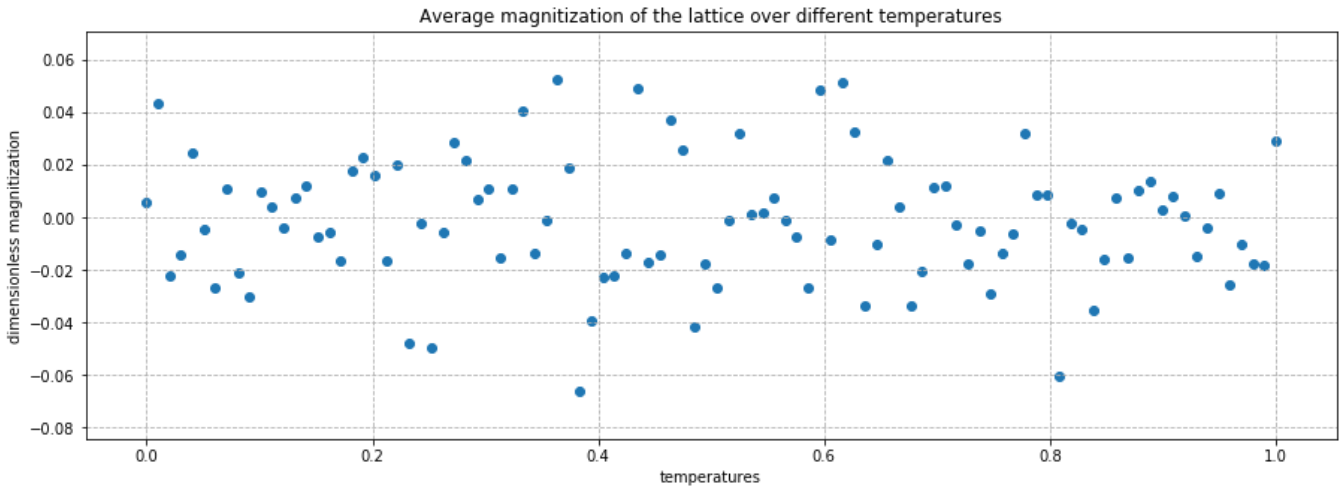
$$M = \sum_{i}^{N} s_i$$

Use

$$k_B T = 2.3J$$

In [38]:
```
r_catalog = get_r([0,2,-2,4,-4], J=8.62E-5*5/2.3, B=0, K=8.62E-5, T=5)


M_list=[]
t_list = np.linspace(0.0,1,100)
for t in t_list:
    lattice = create_lattice(51,49,0.5)
    M_value=0
    for i in range(2000):
        lattice = simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=1, B=0, T=T)[0]
        M_value += np.sum(lattice)/np.size(lattice)
    M_list.append(M_value/2000)


plt.figure(figsize=(15,5))
plt.title('Average magnitization of the lattice over different temperatures')
plt.xlabel('temperatures')
plt.ylabel('dimensionless magnitization')
plt.grid(linestyle='--')
plt.scatter(t_list,M_list)
```

Out[38]:
```
<matplotlib.collections.PathCollection at 0x11a3d10b8>
```



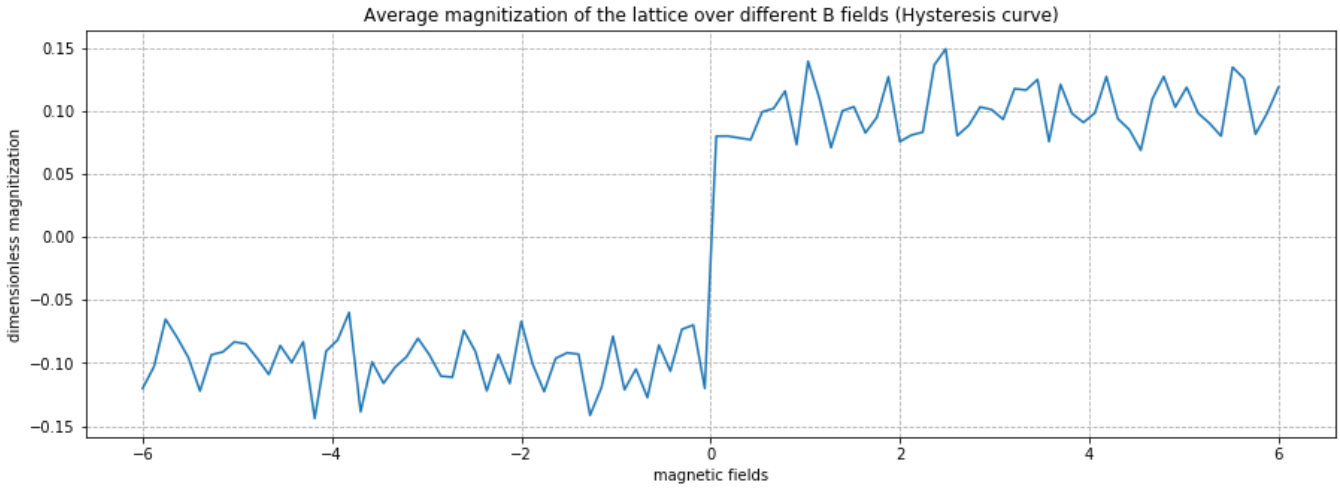
The plot fluctuates around zero so we can say that the lattice of atoms does not have a magnitization of it's own.

Produce a hysteresis curve over an appropriate range of external magnetic field B values.

The Hysteresis curves plots the magnitization over a range of external magnetic fields B. Here we will look at the lattice over 2000 steps and external magnetic fields from -6 to 6 with 100 points

In [39]:
```
r_catalog = get_r([0,2,-2,4,-4], J=8.62E-5*5/2.3, B=0, K=8.62E-5, T=5)


M_list=[]
B_list = np.linspace(-6.0,6.0,100)
for B in B_list:
    lattice = create_lattice(51,49,0.5)
    M_value=0
    for i in range(2000):
        lattice = simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=1, B=B, T=5)[0]
        M_value += np.sum(lattice)/np.size(lattice)
    M_list.append(M_value/2000)


plt.figure(figsize=(15,5))
plt.title('Average magnitization of the lattice over different B fields (Hysteresis curve)')
plt.xlabel('magnetic fields')
plt.ylabel('dimensionless magnitization')
plt.grid(linestyle='--')
plt.plot(B_list,M_list)
```

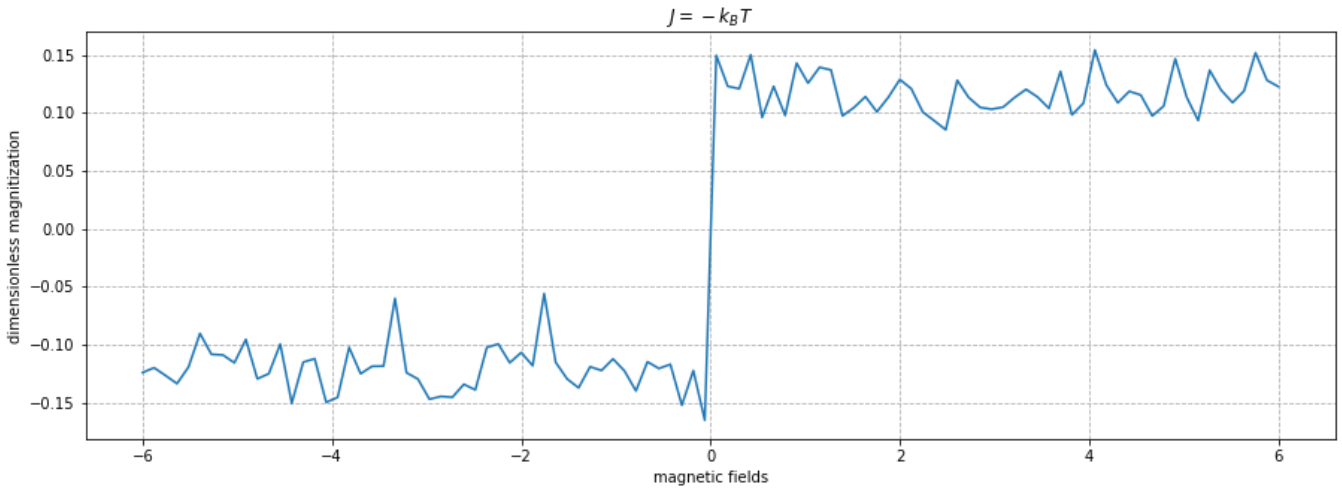
Out[39]:
```
[<matplotlib.lines.Line2D at 0x11b0d46a0>]
```



Produce a plot for J=-kB T.

In [40]:
```
r_catalog = get_r([0,2,-2,4,-4], J=8.62E-5*5/2.3, B=0, K=8.62E-5, T=T)
M_list=[]
B_list = np.linspace(-6.0,6.0,100)
for B in B_list:
    lattice = create_lattice(51,49,0.5)
    M_value=0
    for i in range(2000):
        lattice = simple_step_2D(lattice, r_catalog=r_catalog, nsites=1, J=8.62E-5*5, B=B, T=5)
[0]
        M_value += np.sum(lattice)/np.size(lattice)
    M_list.append(M_value/2000)


plt.figure(figsize=(15,5))
plt.title('$J=-k_B T$')
plt.xlabel('magnetic fields')
plt.ylabel('dimensionless magnitization')
plt.grid(linestyle='--')
plt.plot(B_list,M_list,label=str(T))
```

Out[40]:
```
[<matplotlib.lines.Line2D at 0x11868c400>]
```



Briefly discuss all of your results.

The fundamentals for an evolution of state was deterministic in the game of life. In the Ising model there is an element of randomness to the result of model. We followed the same process as the game of life, starting in solving in 1D the expanding into the 2 dimensional case.

The first way to compute the model was way to computationally expensive to be a useful solution. This is becuase you had to compute all the permutations of the system and look at all the energies for those systems. The solution was to use the metropolis algorithm. for an independent dipole, there was onyl two possible states and so it was easy to compute use those reultes to populate the larger lattice.

We saw that the larger the energy difference the more chance of a spin staying put and not changing. Running the model choosing the states based on their probability then looking at the results, we saw that the states were choosen sufficiently often to follow the theortetical probabilities.

For the non-independent states, we used the realtive proabiblites to find the probability of the state switching. the 1 dimensional case only had 6 possible values, we could compute them seperatly and call them when needed instead of doing expensive exponentials. Expanding then to 2 dimensions, there are 10 possible values so we can use the same trick to make computation faster. If we were to expand even farther to use the diagonal neighbours then we would need more values. A more phsyically correct model would have a gradual fall off of influence due to the nature of the magnetic field.

We saw that steping over a lattice with B=0 and $J < 0$ we got a checkerboard pattern as the only influence that the spins ahve is to be opposite to their neighbours. If there is a B field non zero, then the spins will tend towards that field direction. we also saw that if there is no B field and a $J < 0$ then we will always reach a steady state solution with approximatly half the cells in a spin up state and half the cells in a spin down state.

As long as the B field is sufficiently large to counteract the coupling constant, the spins will have a tendancy to orient themselves based on the external magnetic field. The magnitization of the atoms depends on the average of the magnitization of the spin states in the lattice. It was found that the Hysteresis curve has a more pronounced change if the coupling constant depends on temperature of the system.

Specific References

Curie Temperature: https://www.britannica.com/science/Curie-point (https://www.britannica.com/science/Curie-point)

MIT notes: see notes in PHYS 481 for referneces