Let $\mathcal{S}$ represent the set of integer pairs (m, n) where $0 \leq m, n < 100$. These points constitute a square lattice in the plane. Later we'll turn $\mathcal{S}$ into a torus for the purposes of implementing a Metropolis-type algorithm on $\mathcal{S}$, but for now you can think about $\mathcal{S}$ in the usual way.

Someone tells us that a certain probability distribution (technically a probability mass function) p on $\mathcal{S}$ satisfies

$$p(m, n) \text{ is proportional to } \; q(m, n) = \frac{1}{1 + |m - 50|^\sigma + |n - 50|^\sigma}$$

for all $(m, n) \in \mathcal{S}$, where σ is a nonnegative number. You can find the actual values of p by normalizing via

$$p(m, n) = \frac{q(m, n)}{\sum_{(m,n)\in\mathcal{S}} q(m, n)} \; .$$

When $\sigma = 0$, $p(m, n)$ is the same for all (m, n), so the given probability distribution is uniform on $\mathcal{S}$. When $\sigma > 0$, the distribution peaks at $(m, n) = (50, 50)$. Larger σ makes the peak sharper and narrower. For every value of σ, p is symmetric about $(50, 50)$. Define the function $f : \mathcal{S} \to \mathbb{R}$ by

$$f(m, n) = m + n \text{ for all } (m, n) \in \mathcal{S} \; .$$

You'll discover — no surprise given the symmetry of p — that

$$E_p(f) = \sum_{(m,n)\in\mathcal{S}} f(m, n)p(m, n) = 100 \; .$$

In this toy example we we know $E_p(f)$, but let's pretend we don't and compute it two different ways using randomness. The first approach takes uniformly distributed samples from $\mathcal{S}$ and applies the Strong Law of Large Numbers . Let μ denote the uniform probability distribution on $\mathcal{S}$, so

$$\mu(m, n) = \frac{1}{10,000} \;\; \text{for all } (m, n) \in \mathcal{S}$$

because $\mathcal{S}$ contains 10,000 points. Let Z_k, $k > 0$, be an iid sequence of samples from $\mathcal{S}$ taken according to μ. SLLN tells us that if $g : \mathcal{S} \to \mathbb{R}$ is any function, then

$$\frac{1}{N} \sum_{k=1}^{N} g\left(Z_k\right) \tag{1}$$

converges with probability 1 as $n \to \infty$ to $E_\mu(g)$, the expected value of $g(Z)$ when Z is uniform on $\mathcal{S}$. Let's set

$$g(m, n) = 10,000\, f(m, n)p(m, n) = 10,000\, (m + n)p(m, n) \;\; \text{for all } (m, n) \in \mathcal{S} \; .$$

Then

$$E_\mu(g) = \sum_{(m,n)\in\mathcal{S}} g(m, n)\mu(m, n) = \sum_{(m,n)\in\mathcal{S}} f(m, n)p(m, n) = E_p(f) \; ,$$

which is what we want to calculate. The variance of the Nth estimate in (1) is given by

$$\frac{E_\mu\left(g^2\right) - \left(E_p(f)\right)^2}{n} \; ,$$

and the value of the "spikiness parameter" σ affects the numerator significantly — the larger σ is, the bigger that numerator.

Here's a short Python program that implements the SLLN approach to computing $E_p(f)$. The variable spike is what I called σ above.

```
import math
import random

spike = float(input('Enter spikiness value: '))
N = int(input('Enter number of iterations: '))

def q(m,n):
   return 1/(1 + (abs(m - 50))**spike + (abs(n - 50))**spike)

time_avg = 0

for k in list(range(N)):
    m = random.randint(0,99)
    n = random.randint(0,99)
    time_avg = (k*time_avg + 10000*(m + n )*q(m , n)/normalization)/(k + 1)
    k += 1

print('Computed mean = ' + str(time_avg))
```

I suggest running the program several times for for maybe three different values of the spike parameter (say 0.2, 2, and 11) and for different numbers of iterations (say 10,000, 50,000, and 100,000 or more) and see what happens. Run it a few times for each parameter pair and see how the computed means vary between runs. Remember that the theoretical limiting value is 100. Note that this code doesn't keep a record of the random samples from $\mathcal{S}$ and uses this to update the mean estimate iteratively:

$$\frac{1}{k+1}\sum_{l=1}^{k+1} g(Z_l) = \frac{1}{k+1}\left(k\left(\frac{1}{k}\sum_{l=1}^{k} g(Z_l)\right) + g\left(Z_{k+1}\right)\right) .$$

Now for the Monte Carlo-type approach to computing $E_p(f)$. I'd like you to construct code that implements a Metropolis-type Markov chain on $\mathcal{S}$. Leave the spike parameter and number of iterations as user inputs. Initialize your iteration at a uniformly randomly chosen $(m, n) \in \mathcal{S}$. Let X_k be the state of the Markov chain at iteration time $k > 0$. You'll choose X_{k+1} as follows:

- Propose: choose Y_{k+1} uniformly over the nine nearest neighbors of X_k, including X_k itself. If $X_k = (m, n)$, these nine neighbors are arrayed like this:

$$\begin{array}{ccc} (m-1, n+1) & (m, n+1) & (m+1, n+1) \\ (m-1, n) & (m, n) & (m+1, n) \\ (m-1, n-1) & (m, n-1) & (m+1, n-1) \end{array}$$

  Do all the addition here mod 100 — in effect, we're wrapping $\mathcal{S}$ into a torus, so, for example, $(0, 17)$ is just to the right of $(99, 17)$ and $(37, 99)$ is

just below $(37, 0)$. The torus thing forces everyone to have nine neighbors, which makes the coding easier.

- Accept/reject: set $X_{k+1} = Y_{k+1}$ if $q(Y_{k+1}) \geq q(X_k)$. If $q(Y_{k+1}) < q(X_k)$, set $X_{k+1} = Y_{k+1}$ with probability $q(Y_{k+1})/q(X_k)$ and set $X_{k+1} = X_k$ with $1-$ that probability.

Assemble a set $\mathcal{P}$ of all the X_k-values you generate. Compute

$$\frac{1}{N}\sum_{k=1}^{N} f(X_k) \ ,$$

which we know from Markov chain theory converges to $E_p(f) = 100$ with probability 1 as $N \to \infty$.

Run experiments with your code using different spike and N values to see how things work. (Full disclosure: I haven't done this myself yet, but I did the experiments I described above with the SLLN approach. I'll get around to it.)

You're probably way better with Python than I am, but here's a way to implement the random Accept/reject thing.

```
random.random()
```

generates a uniformly random number in $[0, 1]$. To do whatever with probability say 0.4,

```
if random.random() <= 0.4:
    do(whatever)
```

When you run your code, you assemble a multi-set $\mathcal{P}$ of points in $\mathcal{S}$, and if your number of iterations is sufficiently large, the Monte Carlo run will have "burned in," and the points in $\mathcal{P}$ will be distributed according to the probability distribution p. If you know how to do 3-d graphs and/or histograms, it might be fun to compare the $\mathcal{P}$-histogram with a graph of $p(m, n)$ vs. (m, n).

Another item you might want to play with is the proposal process above. Instead of proposing uniformly among the nine nearest neighbors, how about reach out farther, e.g. to all $(m + j, n + k)$ where $|j|$ and $|k|$ are less than or equal to 2 or some larger step size — maybe make step size an input to your code to see how fast (in the sense of number of iterations) larger step sizes make your computed meaan converge at the cost of a more complex proposal process (and who knows what all this might entail in terms of run time).