Computational Optimization
Homework 4

Nedialko B. Dimitrov

By completing this homework assignment you will learn about:

- Creating parallel code, chunking operations, reassembling results.
- Testing distributions for uniformity.
- Speed of convergence for different random walks.

In this homework, you will test whether the uniformly random points we generated in the last homework are actually uniformly random. Lots of errors can happen when we write code: we might have made an error in the theory; we might have made an error in our coding of the theory; or we might not understand exactly how to use the code we wrote. Because coding can introduce many errors, it is critical we write some tests of our code – to make sure it is actually doing what we think it should be doing. That is what we'll do in this assignment. In the process, we'll also learn about how to write parallel programs, that use multiple processors or multiple machines to get the job done for us.

1 Set Up Basic Test Infrastructure

To test whether the points we are generating are actually uniformly random, we are going to divide the region into little equally-sized boxes and count the number of points in each box. Intuitively, if the number of points in each box are about the same, then we are generating uniformly random points. In this section, you'll create a couple of helper functions to facilitate this testing.

First, implement the following function:

```
def getChiSqSampleSize(nbins=None,freqs=None,min_num=20, conf=.95):
    """Return the number of samples required to get a good p-value out of
       the Chi-square test.

    The number of samples returned ensures that every bin has at least <
       min_num> samples in it with <conf> confidence.

    nbins -- the number of bins, if its a uniform distribution.
    freqs -- the expected frequencies of the bins, as an array

    One of <nbins>, <freqs> has to be given as input

    retval: the number samples required"""
```

For this function, model the expected number of points in each bin as a Normal distribution with the appropriate mean and variance. Then, use a union bound, also known as Boole's inequality, to

compute the probability that *any* bin has less than `min_num` samples. Finally, use `scipy.optimize.bisect` to compute the overall sample size required to ensure that all bins have at least `min_num` samples. We are going to use this function to figure out, based on the number of small boxes we've created inside the region, how many points we need to sample to check for uniformity. You might find the `scipy.stats` module useful.

Second, implement the following function:

```
def computeFeasibleBoxes(c, bbox, divisions=20):
    """Compute the feasible boxes inside a convex region

    c -- a ConvexRegion object.  Should have a .feasible function that
       checks if a point is feasible.
    bbox -- a bounding box for the convex region.  [ [coord_min, coord_max
       ] ]
    divisions -- integer, the number of bins for each coordinate.

    retval: a 0/1 array of shape [divisions]*dim, with 1 for feasible
       boxes.  dim is the dimension of the region"""
```

In this function, you will create little equally sized boxes inside the region. The function arguments specify the ConvexRegion object, as well as a bounding box for the region, and the number of divisions–the same number for each coordinate. For example, if the bounding box is `[[0,1],[0,1]]` and divisions is `2`, then you are creating `4` boxes. In general if we have n coordinates and divisions is k, then we are going to create k^n boxes. Only some of those boxes might be inside the convex region. We only want boxes that are *completely* inside the convex region, because we expect the number of samples in each box to be the same. If the box is only partially inside the region, then it may not get the same number of samples as other boxes. The return value of the function is an array with n dimensions each of size k with 0/1 entries corresponding to the k^n boxes we crated. An entry will be zero if any part of the box is outside the region, and one otherwise. I suggest first checking if the center of the box is inside the region, and only then checking all its corner points. You might find `itertools.product` useful. You might have to go back and add a `feasible(self, x)` method to your ConvexRegion class.

2 Implement Parallel Uniformity Testing

The `getChiSqSampleSize` may return millions of samples, depending on how small the boxes we create are. For example, if we have 40000 feasible boxes, the sample size we need is just under 2.2 million points. Because of this, we are going to implement parallel sampling. In other words, we are going to have 4 or so programs collecting samples simultaneously, and put the results back together for our uniformity testing. We are going to do this parallelization through `IPython.parallel`. I actually implemented this code twice, once through a `direct_view` and the second time through a `load_balanced_view`. What I describe below is the `load_balanced_view` way of doing things, because it ends up producing cleaner looking code. There are many ways to do it, of course, and you can do it in any way that makes sense to you.

Implement the following function:

```
def run_parallel_test( sampleFunc, c, bbox, divisions, samples_per_call =
    10000, name=''):
    """sampleFunc -- a function that returns samples from the region
    c -- the convex region, with a c.feasible(x) function
    bbox -- a bounding box for the region [[c_min, c_max]]
```

```
    divisions -- the number of divisions for each coordinate of the bbox

    retval: (feas,h,pval)
        feas -- 0/1 array of the feasible boxes
        h -- histogram of points in the feasible boxes
        pval -- the p-value of the chi-square test"""
```

The documentation above describes inputs and outputs well. To help point you in the right direction, consider the following. The function `scipy.histogramdd` will likely be useful. The function `sampleFunc` has to be entirely self-contained. In other words, it cannot refer to any variables that are not passed as arguments, or even modules that are not imported inside the function. This is because this function is going to be run in another program, or potentially another computer. So, it cannot know about anything that it does not create itself or is not passed into it. Finally, you might also find `scipy.stats.chisquare` useful.

Alternately, you can implement the `run_parallel_test` function using a `direct_view` and `push` and `pull` operations. I think that is a little easier to think about, but it leads to ugly code that is maybe less re-usable.

3 Run Some Uniformity Tests

With the code you wrote above, run the five following uniformity tests:

- **triangleHR**, the sampling function is hit and run with 20 steps; the region is the 2d triangle; the bounding box is `[[-.1,1.1],[-.1,1.1]]` and divisions is `200`. To help you debug, for this one I get 13534 feasible boxes, and a sample size of about 710000. This test took about 7 seconds to run on my machine, with 4 simultaneous samplers.
- **triangleBall30**, the sampling function is ball walk with 30 steps and radius 0.3; the region is the 2d triangle; the bounding box is `[[-.1,1.1],[-.1,1.1]]` and divisions is `200`. This test took about 2 seconds.
- **triangleBall300**, the sampling function is ball walk with 300 steps and radius 0.3; the region is the 2d triangle; the bounding box is `[[-.1,1.1],[-.1,1.1]]` and divisions is `200`. This test took about 30 seconds.
- **pyramidHR**, the sampling function is hit and run with 40 steps; the region is the 3d pyramid; the bounding box is `[[-1.1,1.1],[-1.1,1.1],[0,1.1]]`; and divisions is 60. To help you debug, I get about 23400 feasible boxes, and a sample size of about 1256265. This test took about 40 seconds to run on my machine.
- **randomWeights**, you come across some code for a convex region defined by its extreme points. That code generates uniformly random points in the region, by taking random convex combinations of the extreme points. Download `ConvexRegionByExtPoints` code from the class website, read over it to see how it works, and test the 2d hexagon `ConvexRegionByExtPoints( [[-2,-1], [-2,1], [-1,2], [1,2], [2,1], [2,-1], [-2,1], [-1,-2]])` with bounding box `[[-2.1,2.1],[-2.1,2.1]]` and divisions `200`. For me, this test took about a second.

4 Visualize Convergence to Uniform

Above, we could see if we are getting uniform distributions with our samplers. In the above tests, the ball walk with 30 steps should have been rejected as being not uniform, while the ball walk with

300 steps should have been ok. In this section, we are going to visualize the speed of convergence of these methods.

The chi-square test essentially looks at the L^2-distance between the target distribution–uniform in our case–and the observed distribution. If this distance is big, we are far away from uniform, and if it is small, we are close to uniform. Specifically, the test looks at the following statistic:

$$\sum_{b \in \text{Bins}} \frac{(o_b - e_b)^2}{e_b},$$

where o_b is the observed frequency–sample count in the box, over total samples in all boxes–and e_b is the expected frequency, `<npoints_in_feasible_boxes> * 1/<nfeasible_boxes>`. We are going to plot this distance as the number of steps of the random walk increases. If we had an infinite number of samples, and we are actually converging to the distribution we think, then eventually the o_b should be the same as the e_b. Of course, we don't have an infinite number of samples, but we are going to do this with a large number of samples. So, we are going to parallelize the part of the code that does the sample steps.

Write the following parallelized function:

```
def run_parallel_l2(reg, stepf, bbox , divisions, start_pt, npts, nsteps,
    chunksize=10000):
     """Return the l2 distance from uniform for a sampling scheme.

     reg -- region
     stepf -- the step function (will be called in parallel)
     bbox -- a bounding box
     divisions -- the number of divisions
     start_pt -- the starting point
     npts -- the number of points in the test
     nsteps -- the number of steps
     chunksize -- chunk size for parallelizing

     retval:  a list of l2 distances from uniform for each of the nsteps"""
```

Suppose that npts is 500000, here we have to take a step with that many points. You are going to parallelize taking that step, farming out chunk sizes of 10000 points to the engines. The `load_balanced_view` has a chunksize parameter, but for some reason that did not work for me. So, when implementing this function I split the points array into chunks myself using the following function, which may be helpful to you:

```
def chunk_array(a, chunksize=10000):
    return [ a[x*chunksize:min((x+1)*chunksize, a.shape[0])] for x in
       range(a.shape[0]/chunksize + 1) ]
```

Also, especially since you will have to loop this several times to do all the `nsteps` required, you may have to clean the results cache of your `load_balanced_view` object with `lv.results.clear()` to avoid an out of memory error.

Once, you've implemented the above function, compute L^2 distances based on these inputs:

- **Hit and Run**, let the region be the 2d triangle, the step function be the hit and run step, the bounding box `[[-.1,1.1],[-.1,1.1]]`, divisions is `200`, `start_pt = [0.25, 0.25]`, the number of points is 500000, and the number of steps is 70.
- **Ball Walk**, let the region be the 2d triangle, the step function be the ball step, the bounding box `[[-.1,1.1],[-.1,1.1]]`, divisions is `200`, `start_pt = [0.25, 0.25]`, the number of points is 500000, and the number of steps is 150.

Computing the first took about 43 seconds on my machine, and the second about 53 seconds.

Once you have the measures, plot the logs of the L^2 distances. I get a figure like the following:

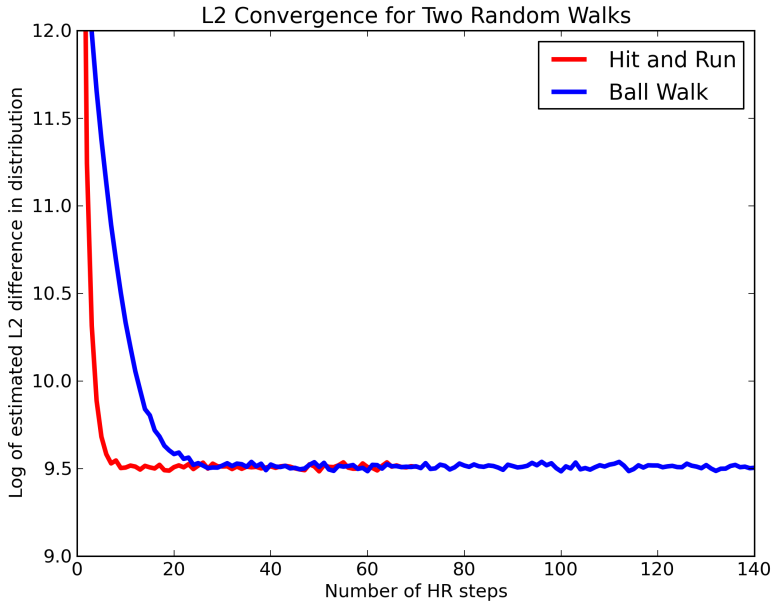

Figure 1: Convergence rates for hit and run and ball walk. The red line is the L^2 distance from uniform for hit and run for a 2d triangle, and the blue line is the same for the ball walk. We can see that hit and run converges much faster, around 10 steps, while the ball walk requires around 70 steps. In larger dimensions, the difference may be more pronounced. One could create similar plots for other distance measures, like L^1 or L^∞.

Also, in programming the previous assignment, many of you had invented your own methods of generating uniformly random points. Try programming one of your methods, and testing its convergence rate in comparison to the other two methods above. Did you get something super-fast? How do the two methods convergences compare in larger dimensions, say on the pyramid?

5 Final Thoughts

What to turn in: For this assignment, turn in three things:

1. Your Python code. In your code, comment which uniformity tests pass and which fail.
2. Three plots:
   - Use `pylab.imshow` to plot the histogram of samples on the 2d triangle for the **triangleBall30** test.
   - Use `pylab.imshow` to plot a similar histogram for the **triangleBall300** test.
   - The convergence plot described in the last section.
3. A short (less than one minute) explanation of your code that you can record through Canvas.

Key take-aways: From this assignment, you should take-away these things:

1. You should now know how to parallelize code, to do your computations on multiple processors and multiple machines at once. The key part here is the idea of *state*. An engine that executes some of the parallel computations, doesn't know very much about the overall program when we call it. The key to parallelizing code, is giving the engine just a small amount of information to do the computation and return the result. This is why we had to write self-contained functions when using the `load_balanced_view`.

2. Testing our code is important. Just because we wrote something doesn't mean it works like we expect it to. Testing your program for correctness is a key part programming, and often as complex as the original programming task.

3. You now know how to test for the convergence of random walks to their stationary distributions. You also have a method of testing the number of steps required to converge for specific instances of the random walk.