Monte-Carlo Project Report

Matthew Portman

Due: 6/14/19

Code Commentary

Monte-Carlo

In coding the Monte-Carlo method, our initial aim was to parallelize and potentially seek other means of sampling. After some issues with parallel implementation in python, we decided to switch our approach to using linear algebra methods via numpy in order to avoid endless serial looping. This certainly speeds up the code and only runs into a memory issue when the number of samples for a single run is exorbitantly large. We can get around that issue by, as suggested in the Confidence Interval document, doing more runs of a smaller sample size (N) in order to achieve the same variance as a bigger sample size with fewer runs. At that, we found the code to be most efficient using a sample size of about 10^5.

In order to avoid waiting forever for the code to run, we placed an upper limit of 10^5 on the number of sample runs to perform. As can be seen in Problem 1, the uncertainty found is still fairly close to a precision of 0.0001 but will take many, many more runs to get there. If we were more familiar with parallel programming in python, we would parallelize the variance loop which is undoubtedly the slowest part of our code.

Cube Integration

In implementing the cube integration, we found initial success in utilizing linear algebraic methods once more in order to efficiently generate the grid of cubes as opposed to permutatively doing so. Unfortunately, this runs into issues rather quickly as d or K gets large. Generating that many points is very taxing on memory and by using linear algebraic methods, we was unable to split it up into chunks since matching had to be done across the entire grid at once using arbitrary dimensional matrices and our current skill level.

Because of this, we are unable to guarantee the four digits of precision for $d > 2$. After many failed attempts at splitting up the process, we tested our method against a permutation-based method and found that, on average, the linear algebra method managed to perform the computation about three times faster. With the large sample size needed to guarantee four digits of precision and memory checking implementation yet-to-be-done, we decided that the permutation method would be far too slow and timely to implement so we chose to stick with linear algebra for our final report.

Problem 1

Note the following code (Figure 1) was run on a laptop with Ubuntu 18.10 Cosmic, an Intel i7-3537U @ 2.00 GHz, and 8 GB of memory. We used 10^5 samples per run for Monte-Carlo and K = 1000 for cube-integration.

We calculated the precision of the cube-integration method by subtracting the lower bound from the upper bound and dividing by two to represent the error bars on the middle point between the two.

```
In 2 dimensions,
Monte Carlo Volume and Relative Error 3.141705943852856 +/-
9.990757393488655e-05
Monte Carlo Time 13.203864097595215

Cube Integration Volume Bounds 3.13672 3.14552
Cube Integration Time 0.13787245750427246 with K: 1000

In 3 dimensions,
Monte Carlo Volume and Relative Error 4.188677271471969 +/-
9.999978218790742e-05
Monte Carlo Time 59.510037899017334

Cube Integration Volume Bounds 4.156380230354278 4.213561160519026
Cube Integration Time 33.735403299331665 with K: 380

In 4 dimensions,
Monte Carlo Volume and Relative Error 4.935308024136039 +/-
9.999898166753437e-05
Monte Carlo Time 182.92313742637634

Cube Integration Volume Bounds 4.57794462713896 5.208554001479121
Cube Integration Time 3.8219335079193115 with K: 62

In 5 dimensions,
Monte Carlo Volume and Relative Error 5.264790057216778 +/-
9.9995172722563e-05
Monte Carlo Time 457.86819100379944

Cube Integration Volume Bounds 4.249612012595134 6.108122731267859
Cube Integration Time 8.932408094406128 with K: 31
```

Figure 1: Code output for problem 1 up to $d = 5$.

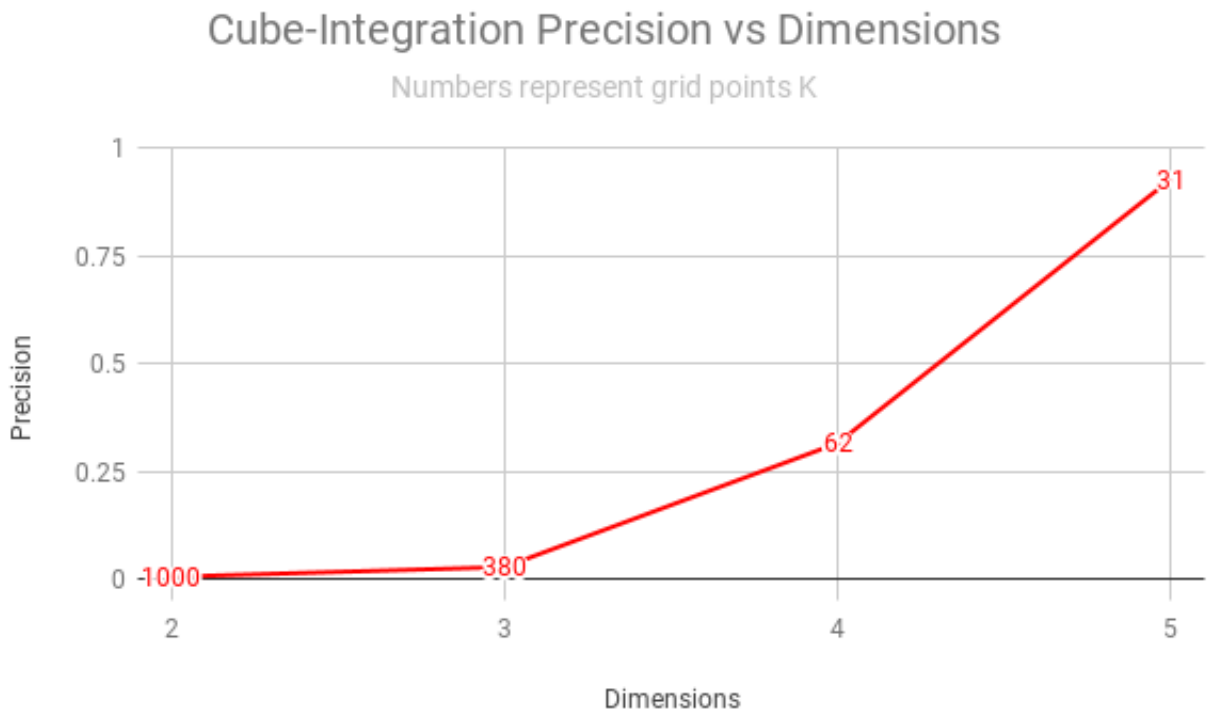

Figure 2: Precision of the cube-integration method per dimension with labels indicating the number of points K.

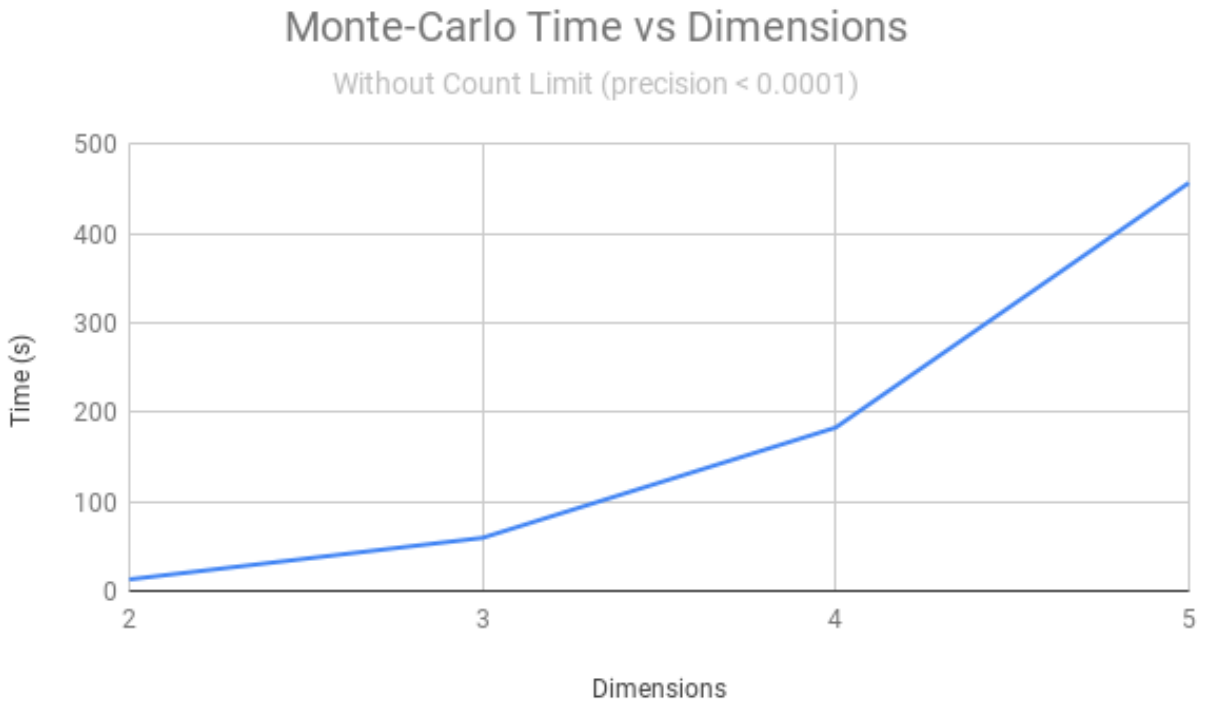

Figure 3: Time elapsed for Monte-Carlo to run and achieve desired 0.0001 precision with $N = 10^5$. Stopped at $d = 5$ due to time constraints.

Monte-Carlo is less efficient but can reach the precision needed (see Figure 3). As d grows large it is more efficient for memory since the number of cubes increases exponentially in the cube-integration formulation. This as opposed to Monte-Carlo which simply needs to check more points to accommodate uncertainty in higher dimensions.

For Monte-Carlo, and with a memory checking and distributing function, we can push d as far as possible - eventually the calculation will take far too long for the precision but in implementation the code can work with any d.

For the cube-integration method, anything above two dimensions takes up too much space using a linear algebraic method to generate the coordinates of each square and thus the method loses precision (see Figure 2). With a permutation method, it takes a much longer time although doing so would avoid the memory issues of the first method. If we insist on 4 digits of precision, this method is unable to compete with the Monte-Carlo method.

When we aim for 8 digits of precision, only Monte-Carlo is viable due to the sheer number of hypercubes required in the integration method. In demanding such high precision though, and in serial, Monte-Carlo takes a long time.

For 8 digits of precision, we anticipate the answer being the same since we maintain the 95% confidence interval standard for Monte-Carlo and the final standard deviation, per the definition $\sigma^2 = \tilde{\sigma}^2(n,k)/nk$, reduces over time.

Problem 2

For this problem, we calculated the cube-integration volume as recommended by taking the total inside volume and adding the volume of half the cubes by the boundary.

d	Wikipedia	Monte-Carlo Volume ± 0.0001	Cube Integration Volume	Difference
2	3.142	3.1415	3.1348	0.0067
3	4.189	4.1884	4.1748	0.0136
4	4.935	4.9340	4.9045	0.0295
5	5.264	5.2636	5.1674	0.0962
6	5.168	5.1671	5.4258	0.2587

Note* - From a previous test on a different computer.

Based on the previous problem and on the Wikipedia estimates, we believe Monte-Carlo is giving the more accurate answer.

Problem 3

For Monte-Carlo, we anticipate that for a higher the number of samples, the more precise the answer will be. If we maintain the standard of 95% confidence for either number of samples though, the accuracy of both should be about the same. The most accurate value using this method is limited to 3.1415 using a precision of 4 digits.

```
Number of Monte-Carlo samples: 100000
And number of grid points K: 1000

Using Monte-Carlo in 2D we get pi to be 3.1416658445040215 +/-
9.996062380457008e-05
And in 3D we get pi to be 3.141408470031546 +/- 9.997636159494185e-05

Cube Integration pi 3.14112
```

Figure 4: Test of concept for calculating pi using Monte-Carlo in 2 and 3D as well as cube-integration in 3D given user input N and K.

Using the deterministic method however, we anticipate better accuracy for higher samples but the same precision.

In order to accurately calculate π, we would use $d = 2$ for the hypercube method to be more efficient and 2 and 3 for Monte-Carlo so that we may check both answers against each other. We're not familiar with variance reduction techniques per se but, in the Monte-Carlo method, using multiple values of d to calculate π should effectively reduce the variance by introducing more samples.

Conclusions

Monte-Carlo is a far better method for calculating the volume of an n-ball precisely and accurately. With appropriate parallelizing, it will be able to compete time-wise with the less accurate cube-integration method for any dimension d.

Project Code

```
import numpy as np
       import sys
       import time
       #import multiprocessing as mp


       radius = 1 # Hard-coded because it doesn't change.

def user_input():

           while True:

               try:
       #             dim = int(input("Please input a dimension: "))
                   N_0 = int(input("Number of Monte-Carlo samples: "))
                   K = int(input("And number of grid points K: "))

               except ValueError:
                   print("Something went wrong, please try again.")
                   continue

       #        if dim < 2:
       #            print("The number of dimensions must be positive and greater than 1.")
                   # For 1D, python throws an error due to the array manipulation.
       #            continue

               if N_0 < 0:
                   print("The number of samples must be positive.")
                   continue

               if K < 0:
                   print("The number of samples must be positive.")
                   continue

               else:
                   # Success!
                   break

           return [N_0, K] #[dim, N_0, K]
```

```
# A function which calculates the volume of the big hypercube.

        def hypercube(s_radius, dim):

            volume = (2*s_radius)**dim # For this project, s_radius = 1
                                       # but for generality I kept a variable.

            return volume

# Used for determining chunk size for the Monte Carlo algorithm.

        def mem_check_mc(N, dim):

            count = 1 # Keeping track.

            while True:

                try:
                    #print("Trying: ", N)

                    # Here we try to initialize an array and if that fails...
                    some_array = np.empty([N, dim])

                except MemoryError:
                    N = N/10 # ... we divide by 10 and try again.
                    count *= 10 # How many 10's we've divided out.
                    continue

                break

            # For memory-sake
            del some_array

            return [N, count]


# Determines the variance^2 per the formula given
        # in the confidence interval sheet.

        def sample_variance(volumes):

            k_runs = volumes.size
            var_squared = 1/(k_runs-1)*np.sum((volumes-np.mean(volumes))**2)

            return var_squared

# Vectorized Monte-Carlo.
```

```
        def vector_mc(s_radius, dim, N, seed):

            # Using the memory check to chunk out the samples.
            # Doesn't matter much in the final implementation with
            # the wrapper function but still playing it safe.
            chunk_size, chunk_num = mem_check_mc(N, dim)

            inside_hyp = 0

            # Iterate over the chunks!
            for chunk in range(chunk_num):

                gen = np.random.RandomState(seed) # Seeding the distribution.

                while True: # Just in case.

                    try:

                        # Generating the samples somewhere between -1 and 1
                        samples = gen.uniform(-s_radius, s_radius, [chunk_size, dim])

                    except MemoryError:
                        print("Something went wrong! Try memory check again.")
                        return 0

                    break

                # Taking the norm of each coordinate with vectorized operations.
                dist = np.linalg.norm(samples, axis = 1)

                # Sorting so as to avoid checking every point.
                dist = np.sort(dist)

                # Determining where the samples no longer satisfy the radial condition.
                inside_hyp += np.where(dist <= 1)[-1][-1] + 1 # +1 to account for python array n

            return inside_hyp/N # Returns the ratio.

    # Wraps the Monte-Carlo function to perform the variance estimate.

        def mc_wrapper(radius, dims, N, rel_error):

            count = 1 # Start small.
            relative_error = 1 # Need a starting value for the while loop.

            volumes = np.array([]) # Initializing.

            vol_cube = hypercube(radius, dims) # Calculating volume of the big cube.
```

```
        #     total_start = time.time() # Previously used for timing.

            # Begin while loop with high count as a stopping condition.
            # Will stop when either the count is too high or the relative
            # error passes below the desired threshhold.

            while (relative_error > rel_error) and (count < 100000):

                seed = time.time() # Seeding with time and count in case it runs too fast.

                # Appending each sample set to an array to make calculating the mean easy.
                volumes = np.append(volumes, vol_cube*vector_mc(radius, dims, N, int(seed+count)

                # Calculating the variances and relative error
                if count >= 2:
                    variance = sample_variance(volumes) # Actually a squared value.
                    std = np.sqrt(variance/count) # Standard Deviation.

                    relative_error = 2*std/np.mean(volumes) # 95% confidence

                count += 1

            total_end = time.time()
            #print("Total Monte Carlo time: ", total_end - total_start, "s")

            return [np.mean(volumes), relative_error]

    # This code performs the hypercube integration using
        # linear algebraic/vectorized techniques.

        def cube_int_linalg(s_radius, K, dim):

            start = time.time()

            # Memory error catching loop of try/excepts.
            while True:

                num_cubes = K**dim # May change with memory errors.
                grids = np.empty([dim,K]) # K points!

                try: # Initialize big empty arrays.

                    #print("Trying K = ", K)
                    coords = np.empty([dim,num_cubes])
                    coord_dist = np.empty([num_cubes])
                    #break
```

```
        except MemoryError as err:

            K = int(K/2) # Try and try again.

#             print("Sorry! K is too high. Cannot compute the volume of the sphere to the
#             print("Attempting a computation with fewer K... \n")
#             print("New K: ", K)

            continue

        vol_cube = (2/K)**dim # Now that K is set, we can calculate the volume
                              # of each little cube.

        h = s_radius/K # Separation h.
        norm_h = h*np.sqrt(dim) # Extent of h in n-dimensions.

        try: # To be elucidated...

            # Generating the grid in every dimension
            for i in range(dim):
                grids[i,] = np.linspace(0, s_radius-h, K)

            # This is the expensive part. With K^d coordinate points, this
            # rapidly blows-up. Having the extra options set to False in meshgrid
            # helps but unfortunately the operation of putting everything together
            # is what takes up the most memory.
            coords = np.array(np.meshgrid(*grids, sparse=False, copy=False)).T.reshape(-

            # Calculate the distance to the 'starting' corner of the cubes.
            # We add the norm of h first for inside/outside logic below.
            coord_dist = np.linalg.norm(coords, axis = -1) + norm_h

            del coords # Saving memory!

            coord_dist = np.sort(coord_dist) # For checking inside/outside efficiently.

            break # Out of the while True loop.

        except MemoryError as err: # If the grid generation is too hefty.

            K = K - 20 # Since I was able to initalize the arrays in the first
                       # place we're probably pretty close.

            continue

    end = time.time()

    coords_time = end-start
```

```
inside = 0
outside = 0
between = 0

start = time.time()

# *************** HARD-CODING PHI ********************

# For generic phi functions see the commented out block below
# Using sort is faster for such large arrays so I've left this code in.
# Using np.where is essentially a vectorized phi anyway!

# Since we sorted, we can find the index at which the furthest point of the cube is
# at or inside of the circle radius and that's the number of cubes inside.
inside = np.where(coord_dist <= s_radius)[-1][-1]

coord_dist = coord_dist - norm_h

# For the outside, we can find the index at which the closest point of the cube is
# at or outside the circle radius and subtract that number from the total number of
outside = num_cubes - np.where(coord_dist >= s_radius)[-1][0]

# They all sum to the number of cubes so we simply use that to find the number betwe
between = num_cubes - inside - outside

# *************** FOR GENERIC PHI ********************

# Check each point. If the furthest point of the cube is inside then we know
# that cube is definitely inside

#for i in range(len(coord_dist)):
#    if phi(coord_dist[i] + norm_h):                    # < s_radius:
#        inside += 1

# If the furthest edge is out but the closest point is in, we know the cube is betwe

#    elif phi(coord_dist[i]):                     # < s_radius:
#        between += 1

# If neither then the cube is definitely outside.

#    else:
#        outside += 1

# ***************************************************

end = time.time()
```

```
            check_time = end-start

            if between < 0:
                print("Something went wrong! The number of little cubes is incorrect.")
                #print(num_cubes)
                #print(inside)
                #print(outside)
                return 0, 0

            low_bound = inside*vol_cube # The hypersphere definitely can't be smaller than that.
            up_bound = (inside+between)*vol_cube # Or larger than this.

            #print("Coordinate Generation", coords_time)
            #print("Checking Each Coord", check_time)

            return low_bound, up_bound, K

    # PROBLEM 1
        N_0 = 100000
        N_cube = 1000
        rerror = 0.0001

        for i in range(3):

            dims = i+2 # Iterating over the dimensions.
            start = time.time()
            mc_vol, mc_err = mc_wrapper(radius, dims, N_0, rerror)
            end = time.time()
            mc_time = end-start
            print("In", dims, "dimensions,")
            print("Monte Carlo Volume and Relative Error", mc_vol, "+/-", mc_err)
            print("Monte Carlo Time", mc_time, "\n")

            start = time.time()
            low, up, K_final  = cube_int_linalg(radius, N_cube, dims)
            end = time.time()
            cube_time = end-start

            int_vol = low + (up-low)/2
            print("Cube Integration Volume Bounds", low, up)
            print("Cube Integration Time", cube_time, "with K:", K_final, "\n")

    # PROBLEM 2
        N_0 = 1000000
        rerror = 0.0001

        for i in range(4):
```

```
            dims = i+2
            mc_vol, mc_err = mc_wrapper(radius, dims, N_0, rerror)
            print("In", dims, "dimensions,")
            print("Monte Carlo Volume and Relative Error", mc_vol, "+/-", mc_err)

            low, up, _ = cube_int_linalg(radius, int(N_0**(1/dims)), dims)
            int_vol = low + (up-low)/2
            print("Cube Integration Volume", int_vol, "\n")

# PROBLEM 3
        N_0, K = user_input()
        print()
        #mc_2, mc_2_err = mc_wrapper(radius, 2, N_0, rerror)
        #mc_3, mc_3_err = mc_wrapper(radius, 3, N_0, rerror)
        print("Using Monte-Carlo in 2D we get pi to be:", mc_2, "+/-", mc_2_err)
        print("And in 3D we get pi to be:", 3*mc_3/4, "+/-", mc_3_err)
        print()

        low, up, _ = cube_int_linalg(radius, K, 2)
        int_vol = low + (up-low)/2
        print("Cube Integration pi:", int_vol, "\n")

def phi(point, circle_radius):

            if point <= circle_radius:
                return True
            else:
                return False
```