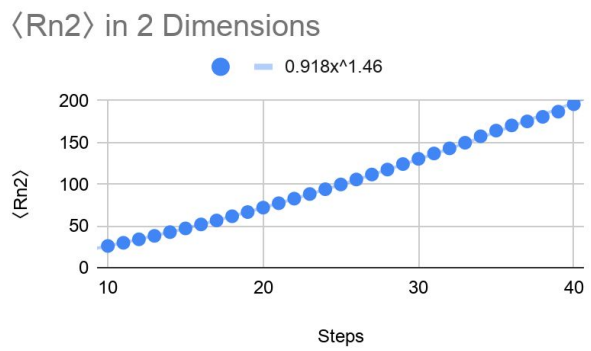
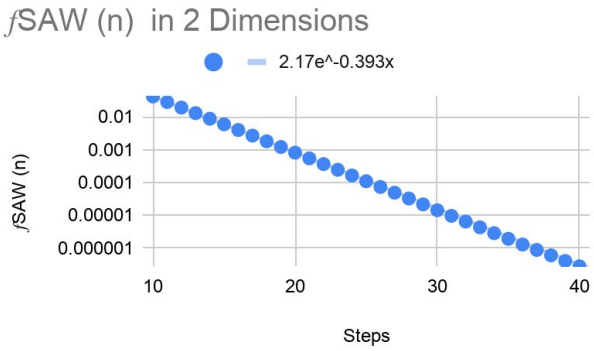
Peter Li 23375692
Professor Sateesh Mane
CSCI 370
September 10, 2019

Cover Document

Graphs and Analysis





Steps (n)	⟨Rn2⟩ in 2 dimensions	fSAW (n) in 2 dimensions
10	26.24085883	0.042051139
11	30.01446429	0.028680071
12	34.18957	0.019366756
13	38.30736807	0.013131953
14	42.79654348	0.008842297
15	47.23495818	0.005974161
16	52.01559686	0.004013885
17	56.72660639	0.002702565
18	61.78695995	0.00181303
19	66.7870839	0.001219269
20	72.10327725	8.17E-04
21	77.35642632	5.48E-04
22	82.86713401	3.66E-04
23	88.40548726	2.46E-04
24	94.12718499	1.65E-04
25	99.83288045	1.10E-04

26	105.7675551	7.35E-05
27	111.7235979	4.92E-05
28	117.6603013	3.27E-05
29	124.2426494	2.17E-05
30	130.4950909	1.44E-05
31	136.8994517	9.67E-06
32	143.03842	6.48E-06
33	149.7432401	4.33E-06
34	157.4509941	2.87E-06
35	164.2708441	1.93E-06
36	170.4783282	1.29E-06
37	175.2779661	8.85E-07
38	180.7173554	6.05E-07
39	186.9215686	4.08E-07
40	195.8214286	2.80E-07

Software Design and Optimizations

To simulate a self avoiding walk, the program must start at the origin and save the points that have been walked to. To achieve this, every time the walk visits a point, the point is checked to see if it has been visited before. If not, the point is walked to and the hashcode of the coordinates are saved to a local hashmap as a string. Otherwise the walk is not self avoiding and is rejected.

To simulate the random portion of the self avoiding walks, a thread safe random number is generated with the range 0 through 2^d exclusive with d being the dimension and stored as r.

E.g. 2 dimensions has $2 * 2 \rightarrow 4$ available paths

3 dimensions has $2 * 3 \rightarrow 6$ available paths

Then the axis the walk will take is generated by taking r mod dimension which gives you one of the dimensional axes to change. To generate the direction of the walk take r minus dimension to get an equal chance of getting a positive or negative integer.

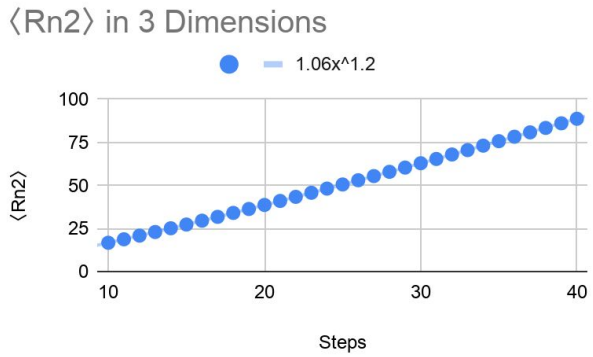
The end to end squared distance at each step is computed and saved in an array with the maximum amount of steps as the size at step index. The amount of completed walks for that step is also incremented by 1 at the same index. After all walks are completed, the total end to end squared distance is divided by the amount of completed walks to get the mean end to end squared distance for that step.

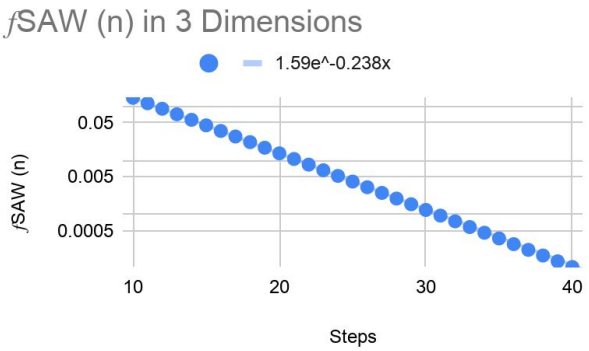
The fraction of walks completed at each step is generated by taking the completed walks at that step divided by the attempted walks.

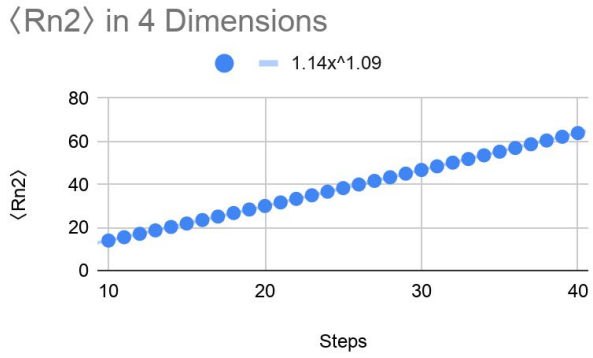
Each thread is given a Walk object to generate and each Walk object runs a loop which for n amount of times with n being the attempted walk count for that thread. The total amount of walks attempted would then be t number of threads multiplied by n number of walks.

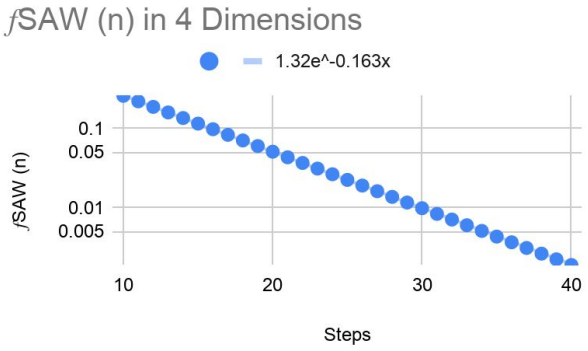
The program can be speed up by creating separate functions for the two cases: Self avoiding walks and self avoiding polygons. Doing this will decrease the amount of IF statements needed to differentiate between the two cases but I decided not to design my program in such a way. By not including different functions for the different cases, the amount of source code becomes more concise.

Higher Dimensions









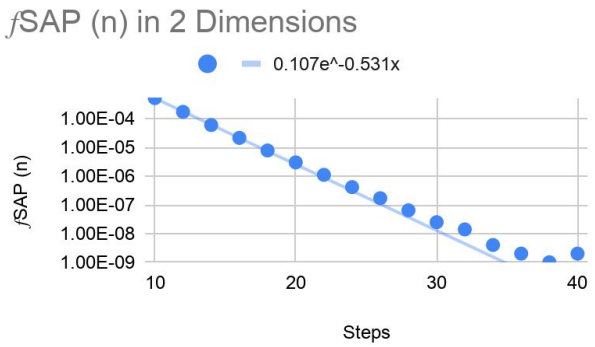
Steps (n)	R2n in 3 dimensions	fSAW in 3 dimensions	R2n in 4 dimensions	fSAW in 4 dimensions
10	16.8170986	0.145707833	14.01943565	0.25773117

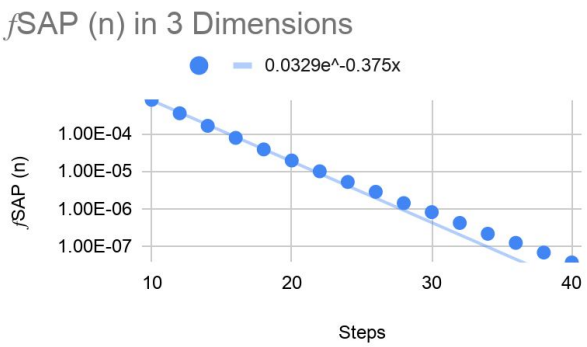
11	18.84206953	0.115584003	15.57124364	0.219634267
12	20.95370558	0.091345163	17.15079427	0.18687991
13	23.05323002	0.072274699	18.72668788	0.15907769
14	25.22836313	0.057029587	20.32609762	0.135250946
15	27.39102081	0.045041695	21.9226732	0.115025702
16	29.62456023	0.035493784	23.53918655	0.097743957
17	31.84782718	0.027989433	25.15261719	0.083077573
18	34.13181816	0.022036721	26.78359832	0.070573679
19	36.40954932	0.01735496	28.41303884	0.059957838
20	38.74071083	0.013651762	30.05729661	0.050914218
21	41.07105336	0.010743194	31.70116734	0.043242907
22	43.44641176	0.008443691	33.35580588	0.036710571
23	45.81323556	0.006640986	35.00963651	0.031168546
24	48.22701193	0.005216466	36.67678843	0.026450337
25	50.62652174	0.00410056	38.34376299	0.022449965
26	53.07070516	0.003221264	40.02245524	0.01904838
27	55.48435561	0.002529852	41.69581509	0.016162541
28	57.95796127	0.001986454	43.38645865	0.013712934
29	60.41672753	0.001559405	45.06570394	0.011636654
30	62.93732655	0.001223309	46.75931099	0.009871236
31	65.4071794	9.60E-04	48.44864496	0.008372957
32	68.00140617	7.52E-04	50.15176096	0.007099665
33	70.54610222	5.90E-04	51.84417754	0.006022001
34	73.1311696	4.63E-04	53.54606813	0.005106828
35	75.72632393	3.63E-04	55.24686814	0.004330417
36	78.30630878	2.85E-04	56.95071402	0.003672322
37	80.8452843	2.23E-04	58.69128354	0.003112811
38	83.41030572	1.75E-04	60.42731771	0.002640265
39	86.08765966	1.37E-04	62.16326282	0.002238985
40	88.72046474	1.07E-04	63.87637472	0.001898479

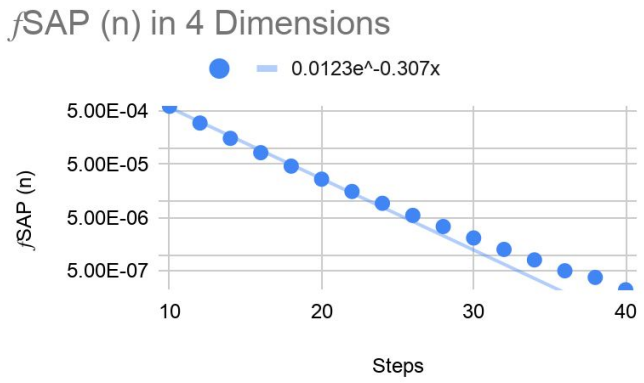
Instead of hard coding each dimensional case, the program dynamically adjusts for the amount of walks and dimensions given to it as parameters. This is done by keeping track of coordinates in an array of size dimension with each space in the array holding the vector of its

corresponding dimension at dimension index. There is theoretically no hard limit to how many steps or dimensions the program can compute but the computation time and memory requirements increase as the amount of steps and dimensions increase with computation time being the most significant increase.

Self-avoiding Polygons







Steps (n)	*f*SAP (n) in 2 dimensions	*f*SAP (n) in 3 dimensions	*f*SAP (n) in 4 dimensions
10	5.34E-04	7.98E-04	6.07E-04
12	1.78E-04	3.50E-04	2.95E-04
14	6.15E-05	1.62E-04	1.52E-04
16	2.19E-05	7.74E-05	8.23E-05
18	7.96E-06	3.81E-05	4.61E-05
20	3.06E-06	1.93E-05	2.63E-05
22	1.13E-06	9.95E-06	1.55E-05
24	4.19E-07	5.17E-06	9.29E-06
26	1.72E-07	2.82E-06	5.52E-06

28	6.50E-08	1.41E-06	3.42E-06
30	2.50E-08	8.11E-07	2.09E-06
32	1.40E-08	4.18E-07	1.28E-06
34	4.00E-09	2.15E-07	8.12E-07
36	2.00E-09	1.23E-07	5.08E-07
38	1.00E-09	6.80E-08	3.82E-07
40	2.00E-09	3.70E-08	2.24E-07

To create self avoiding polygons, the same code for self avoiding walks is used with a couple of IF statements for the self avoiding polygon case. The hashcode of the starting point is not saved to hashmap so the walk can close on itself. When the walk reaches the origin again, the amount of steps is recorded and the array storing amount of completed walks is incremented at step index before starting a new walk.