Optimally Distributed Points on a Sphere and the Thomson Problem

Ben Gresham

May 3, 2016

Abstract

The optimal energy configuration of points on the surface of a 2-sphere, known as the Thomson problem, is an ongoing difficulty with many applications. This document looks at two Monte Carlo methods, the Metropolis and Wang-Landau algorithms, implementing them in the Python programming language then applying them to the Thomson problem and a simple harmonic oscillator. Polyhedra formations are also observed from the output and compared to existing structures.

Acknowledgements

Thanks to Roger Smith for guidance on the project and help understanding the different methodologies.

Contents

1 The Thomson Problem

1.1 Problem Origin

The aim of the Thomson problem is to find the minimum energy configuration of N point charges on the surface of a 2-sphere and is one of Smale's problems for the 21st century along with other challenges [1]. The problem was theorised in 1904 by the physicist J.J. Thomson after proposing his plum pudding atomic model [2]. In this model the charge of the atom is split into positive and negative charges, with the negative charges distributed as small pockets or "corpuscles" and the positive charges spread across the remaining space. The idea of the Thomson problem is to find the spacial configuration of these corpuscles in the atom so that they are in a stable optimised equilibrium pattern. The plum pudding model was later disproved by the Rutherford gold foil experiment and a new model of the atom was formed. Despite this the general problem is still useful and appears in many areas of science [3, 4, 5]. The objectives of the project were to find, or get close to, the minimum energy configurations and the relative density of energy states of the general problem for different N. The results would also be compared to polyhedra to see the relationship between the two. Unfortunately due to time constraints and difficulties the density of states for the Thomson problem could not be found and a different system was analysed.

1.2 Numerical Definition

The idea is to determine the minimum energy configuration of N electrons confined to the surface of a unit sphere. The force on each electron is given by Coulomb's inverse-square law [6] and the electrostatic potential energy. If each electron is labeled from 1 to N the electrostatic potential $E_{ij}(N)$ between points i and j can be defined as $k_e \frac{e^2}{r_{ij}^2}$ where e is the charge of an electron, k_e is Coulomb's constant and r_{ij} is the Euclidean distance between point i and j. To simplify this the constants can be removed without affecting the global minimum energy configuration. The total electrostatic potential is the sum of the potential pairs so this gives, for a system of N points, the normalised energy function (1).

$$E(N) = \sum_{i<j}^{N} \frac{1}{r_{ij}} \tag{1}$$

For very small N the solution is clear. When $N = 2$ the points simply oppose each other and for $N = 3$ they create an equilateral triangle whose plane passes through the centre of the sphere. As the number of points increases it becomes less intuitive but does tend towards certain common shapes and patterns. For the majority of cases there is no analytic solution, so instead computational methods and optimisation algorithms are used to find the energy minima.

1.3 Simulation Methods

1.3.1 Metropolis Algorithm

The first algorithm tested is a type of Monte Carlo method called the Metropolis algorithm. A Monte Carlo method involves generating a sequence of random numbers from a suitable probability distribution to create a random walk [7]. This random walk creates a Markov chain that tends towards the desired equilibrium as the algorithm is repeated. The algorithm works by having the distribution sample of the current step being dependent on the previous iteration. A key part of the algorithm is the accept-reject step where the proposed new value is compared to the current value. If the new value is an improvement on the current distribution then it is simply accepted and the algorithm starts again. If it is not an improvement then the proposed value and current value are used to calculate the acceptance ratio of the move. For this the Boltzmann probability will be calculated, however the acceptance ratio can be generated in other ways. Another key requirement of the Metropolis algorithm is that the probability distribution used is symmetric i.e. that the probability of suggesting a point y from a point x is the same as suggesting x from the point y. Two different symmetric distributions are tested, uniform and Gaussian, to compare the pros and cons of both.

The full algorithm is as follows:

1. Generate a random set S of N points on the unit sphere.
2. Pick a random point of S, call this x_{old}, and generate a new random point from either the uniform or Gaussian distribution, call this x_{new}.
3. Compare the current energy E of the system with the energy if x_{old} was replaced by x_{new}. Denote by E_{old} and E_{new} respectively.
4. If $E_{old} > E_{new}$ then replace x_{old} with x_{new}.
5. If $E_{old} < E_{new}$ then accept the new point x_{new} with probability given by $P(x_{old} \to x_{new}) = \frac{e^{\Delta E}}{kt}$, where $\Delta E = E_{old} - E_{new}$, k is the Boltzmann constant and t is the temperature of the system.
6. Repeat steps 2-5 for a given amount of iterations.

The generation of the initial point set will use the uniform distribution since a Gaussian distribution would result in a small cluster of points and the end result should be an even spread. As with any computer generated random number, the result is not truly random. The number is generated with the help of a seed which can vary with the programming language and operating system used. This seed is usually based on the unique timestamp of when the program is run but again can vary. Luckily with these pseudorandom numbers the effect will be negligible and the end results will not differ. In step 2 the new point generation from the Gaussian distribution will have its mean centred on the current comparison point and standard deviation set as a parameter to change. The Python programming language was used to apply this algorithm. Python was chosen for its ease of use and simplicity, as well as its ability to run scripts quickly. It also has great 3D visualisation tools to help better understand the results.

1.3.2 Wang and Landau Algorithm

Another fairly new Monte Carlo method is the Wang-Landau algorithm which calculates the density of energy states $g(E)$ of a system [8]. Like the Metropolis algorithm there is a random walk in energy space however the Wang-Landau algorithm visits all available energy regions in the walk to

create a histogram. The algorithm works by the observation that if a random walk is performed in the energy space with a probability distribution given by $\frac{1}{g(E)}$ then a flat histogram is created. If initially $g(E)$ is unknown then adjusted as the walk progresses until a flat histogram is obtained, the resultant density of states converges to the correct value. The algorithm was originally designed for discrete energy spectra but can be adapted to continuous systems by creating discrete "bins" of energy level ranges [9]. An important part of the algorithm is the necessity for all energy levels to be accessible so the upper and lower bounds need to be carefully computed. For a general problem the algorithm is as follows:

1. Set all density of states $g(E)$ of energies E to one as they are unknown.
   Set the initial modification factor $f = e$.
   Set an initial system state by random sampling with energy E_{old}.

2. Set histogram entries $h(E)$ to zero and begin the random walk from the current system state.

3. Generate a new comparison state with energy E_{new}.

4. Accept the transition to the new state with probability
   $P(E_{old} \to E_{new}) = min(\frac{g(E_{old})}{g(E_{new})}, 1)$

5. If the move is accepted then
   $g(E_{new}) = g(E_{new}) \cdot f$, $h(E_{new}) = h(E_{new}) + 1$.

6. If the move is rejected then
   $g(E_{old}) = g(E_{old}) \cdot f$, $h(E_{old}) = h(E_{old}) + 1$.

7. Repeat steps 3-6 until the histogram is sufficiently flat i.e. the minimum divided by the average is above a certain value (typically 0.8 or higher).

8. Reduce the modification factor $f = \sqrt{f}$.

9. Repeat steps 2-8 until the modification factor is below a certain threshold.

Again like the Metropolis algorithm a symmetric probability distribution is needed for the generation of a new comparison state. The end result is the relative density of states for the system.

2 Metropolis Code

This section refers to functions inside the Metropolis algorithm programming code to better understand what they do and how they work. Full code documentation can be found in appendix A.

2.1 Initial Construction

The first step of generating a random point set is simple enough. Polar coordinates are used to restrict the domain as follows:

$$x = r \cdot cos(\theta)$$
$$y = r \cdot sin(\theta) \cdot cos(\phi)$$
$$z = r \cdot sin(\theta) \cdot sin(\phi)$$

Setting $r = 1$ means that all points will be on the unit sphere which leaves only having to generate values for θ and ϕ, where $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi]$. This is done using one of the python mathematical libraries. The full function $random\_uniform\_sphere()$ takes in an integer N as the number of points to generate and returns an array of coordinates, generated randomly from the uniform distribution.

2.2 Comparison Point Generation

Although the starting points will always be generated uniformly over the sphere, it is required that subsequent comparison points can also be generated by a Gaussian distribution. One way to do this in python is to use conversion algorithms such as the Box-Muller transformation. There are also built in methods in the python libraries to accomplish this but the Box-Muller algorithm is very simple and works conveniently with the required function output.

2.2.1 Box-Muller Algorithm

The algorithm is as follows:

Take two independent, uniformly distributed random variables R_1 and R_2 on the interval $(0, 1)$. Let S_1 and S_2 be given by:

$$S_1 = \sqrt{-2 \ln R_1} \cdot cos(2\pi R_2)$$
$$S_2 = \sqrt{-2 \ln R_1} \cdot sin(2\pi R_2)$$

Then S_1 and S_2 are two independent, uniformly distributed random variables.

This output is particularly useful since S_1 and S_2 can be assigned to θ and ϕ respectively, with minor adjustments. The values are currently unbounded and have mean value of zero whereas the mean must be centred on old points. To account for this the formula is adjusted as follows:

$$S_1 = \sqrt{-2 \ln R_1} \cdot cos(2\pi R_2) \cdot \sqrt{V} + \theta_{old}$$
$$S_2 = \sqrt{-2 \ln R_1} \cdot sin(2\pi R_2) \cdot \sqrt{2V} + \phi_{old}$$

where θ_{old} and ϕ_{old} are the values for the current point being compared against and V is the variance. The function $random\_gaussian\_sphere()$ thus generates N points, centred on the given values of θ and ϕ with extra parameter V and returns these points in an array. An example of possible generations can be seen in figure 1.

(a) Uniform (b) Gaussian

Figure 1: Uniform and Gaussian point generation examples.

2.3 Main Function

With an initial point set made the metropolis algorithm can be applied. The function *local_monte_carlo_metropolis*() starts by calculating the total energy of the points. From there the main loop starts iterating and a comparison point is made depending on the *type* parameter. To save on computation time the total energy of the system is not calculated with each loop, instead the energies of the old and new points relative to the rest of the system are found. The number of calculations for this step is reduced greatly if done this way and much more efficient for larger N. These relative point energies are then used in the accept-reject step and, if accepted, the difference is taken off the current energy total. After the set amount of iterations the main loop ends and the final point set is output.

2.4 Convex Hull and the Quickhull Algorithm

With an optimised point set a method is required to convey the results. Plotting the set in three dimensions helps but it is unintuitive about whether the results are truly optimised or simply close. One of the better ways to understand the results is to use the point set as if they were vertices of polyhedra. However, for the vast majority of cases, the appearance of the polyhedra will be unknown as a result of their irregularity. To solve this the convex hull of the set can be found, resulting in the formation of a convex polyhedron.

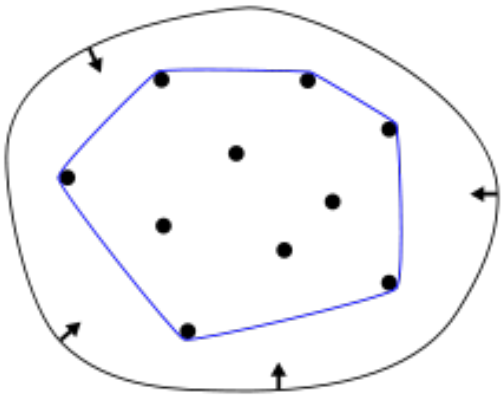


Figure 2: The elastic band analogy for complex hulls in two dimensions.

The convex hull of a set of points S is the smallest convex set that contains S [10]. For a set of points on a one dimensional line, the convex hull is a line connecting the two outermost points. In two dimensions this can be pictured as stretching an elastic band so that it contains the points then letting it snap into place which can be seen in figure 2. The remaining elastic band polygon will contain all the points in S and will also be the smallest possible convex polygon to do so. A commonly used algorithm to achieve this is the quickhull algorithm. The full algorithm is too long to convey but can be found in the original paper by C. Bradford Barber, et al [11]. The *pointplot*() function in the code takes a set of points and simply plots them on a sphere. The *convexhull*() function uses the quickhull algorithm from the convex hull python library to convert a set of points into a convex mesh and display it. The full optimisation process, put together with a flow diagram, can be visualised in figure 3.

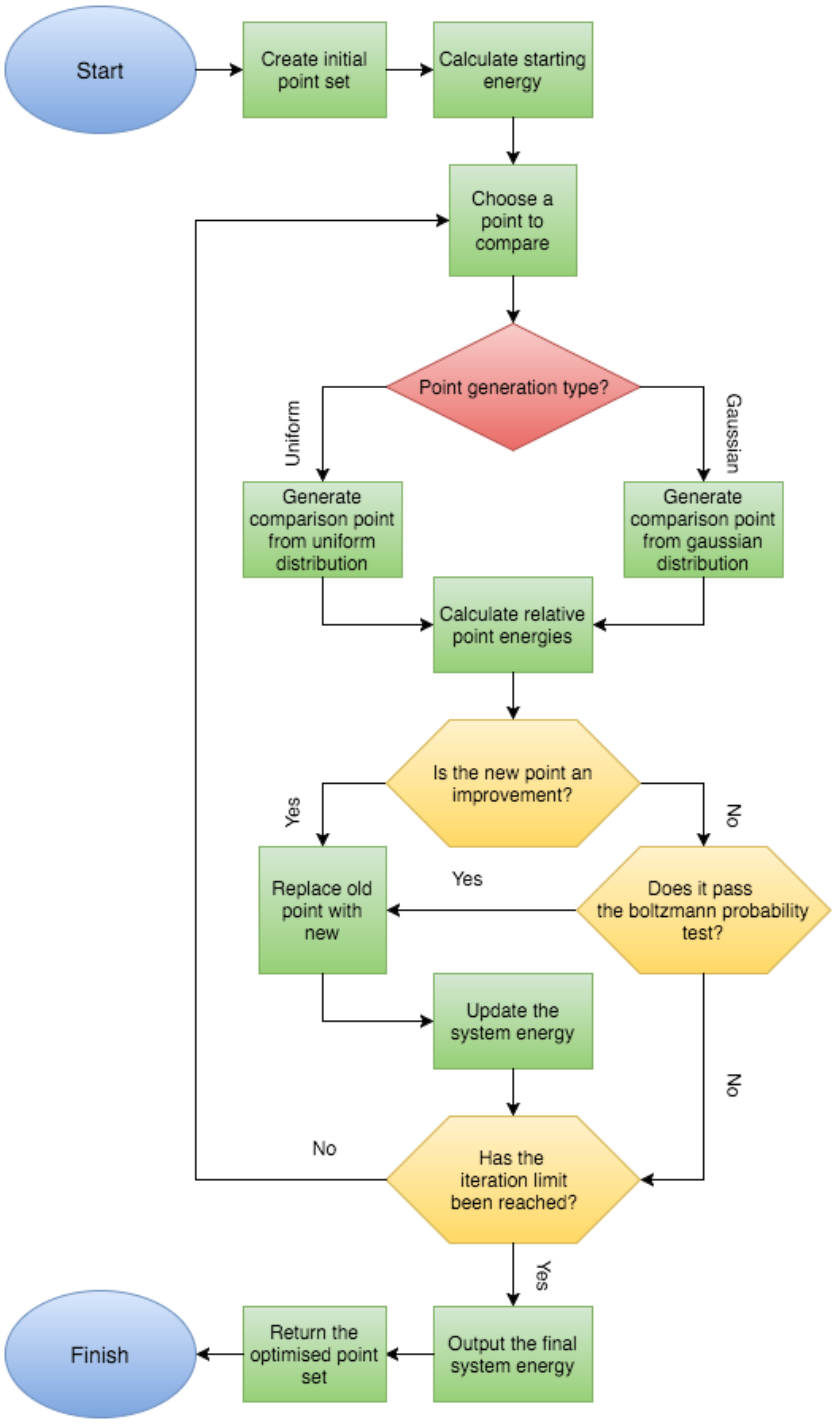
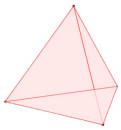
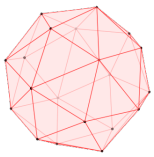


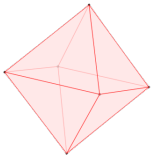
Figure 3: A flowchart showing each stage of the metropolis program.



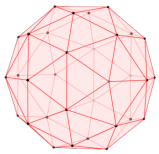
(a) $N = 4, E = 3.674325$



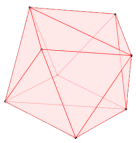
(e) $N = 20, E = 150.883740$



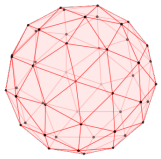
(b) $N = 6, E = 9.985288$



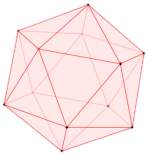
(f) $N = 32, E = 412.261596$



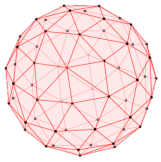
(c) $N = 8, E = 19.675290$



(g) $N = 50, E = 1055.183662$



(d) $N = 12, E = 49.165261$



(h) $N = 60, E = 1543.855053$

Figure 4: Example outputs for different N.

3 Algorithm Results and Polyhedra Formations

3.1 Initial Results

The output from the program gives an ending system energy and an interactive visual of how the points are arranged (figure 4). Notice for small N, figures 4a-4d, the general shape is quite uniform i.e. there are lines of symmetry, faces are triangular and either equilateral or isosceles. As N increases this trait diminishes, first noticeably in figure 4e where some distortion is apparent. This effect deepens in figures 4f-4h but the polyhedra seem to still have some uniformity to them.

3.2 The Platonic Solids

One of the first things the results can be compared to are known convex polyhedra. There are five regular convex polyhedra; the tetrahedron, octahedron, cube, icosahedron and dodecahedron with relative vertex numbers $N = 4, 6, 8, 12, 20$. These are the Platonic solids with every face the same regular shape, all edges the same length and all vertices lying on the surface of a sphere (figure 5).

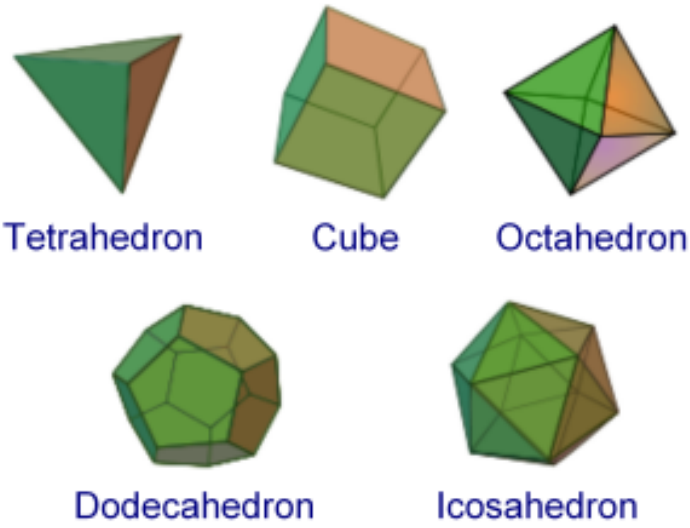


Figure 5: The Platonic solids.

The last two attributes are important as they allow calculating of their system energies as if they were solutions to the Thomson problem and directly compare them to the results. The visual differences are quite clear when comparing figure 5 to figures 4a-4e. For $N = 4, 6, 12$ the result is almost identical looking tetrahedrons, octahedrons and icosahedrons whereas for $N = 8, 20$ the results are quite different. The same basic shape is still there but changed slightly.

3.2.1 Platonic Solid Energies

From the traits of these regular polyhedra, where all edges are the same length and all vertices lie on a sphere, if a unit sphere is used then side lengths and other properties can be derived. To calculate the system energies the distance between every possible pair of vertices is required. If the polyhedra is projected onto a two dimensional plane then graph theory can be used to find the number of edges to calculate. Since all points interact with each other there will be complete graphs K_N. As a result of the Handshaking Lemma [12], a graph with all vertices of degree R, order N has size $S = \frac{N \cdot R}{2}$. For a complete graph, $R = N-1$ so there are $\frac{N \cdot (N-1)}{2}$ edges or unique pairwise interactions to calculate. Through trigonometry and geometric formulae the edge lengths can be segregated since, for the Platonic solids, there will be at most only five different lengths to calculate, all of which can be found from the regular side length L.

1. Tetrahedron $N = 4$, $S = 6$, $L = \sqrt{\frac{8}{3}} \simeq 1.632993162$

$$E = 6 \cdot \frac{1}{\sqrt{\frac{8}{3}}} \simeq 3.674234614 \tag{2}$$

2. Octahedron $N = 6$, $S = 15$, $L = \sqrt{2} \simeq 1.414213562$

$$E = 12 \cdot \frac{1}{\sqrt{2}} + 3 \cdot \frac{1}{2} \simeq 9.985281374 \tag{3}$$

3. Cube $N = 8$, $S = 28$, $L = \frac{2}{\sqrt{3}} \simeq 1.154700538$

$$E = 12 \cdot \frac{\sqrt{3}}{2} + 12 \cdot \frac{1}{\sqrt{\frac{8}{3}}} + 4 \cdot \frac{1}{2} \simeq 19.74077407 \tag{4}$$

4. Icosahedron $N = 12$, $S = 66$, $L = \sqrt{2 - \frac{2}{\sqrt{5}}} \simeq 1.051462224$

$$E = 30 \cdot \frac{1}{\sqrt{2 - \frac{2}{\sqrt{5}}}} + 30 \cdot \frac{1}{\sqrt{2 + \frac{2}{\sqrt{5}}}} + 6 \cdot \frac{1}{2} \simeq 49.16525306 \tag{5}$$

5. Dodecahedron $N = 20$, $S = 190$, $L = \frac{\sqrt{15}-\sqrt{3}}{3} \simeq 0.713644180$

$$E = 30 \cdot \frac{3}{\sqrt{15} - \sqrt{3}} + 30 \cdot \frac{1}{\sqrt{2 + \frac{3\sqrt{5}}{5}}} + 60 \cdot \frac{1}{\frac{1}{6}(1 + \sqrt{5})(\sqrt{15} - \sqrt{3})} + 60 \cdot \frac{1}{\frac{\sqrt{2}}{6}(1 + \sqrt{5})(\sqrt{15} - \sqrt{3})} + 10 \cdot \frac{1}{2} \simeq 152.152865633 \tag{6}$$

From equations (4) and (6) it is clear why there are different results for figures 4c and 4e. Through the Metropolis algorithm an improvement on the Platonic solids has been found. For $N = 8$ a square antiprism is formed by rotating the top side of a cube round 45°, resulting in faces made of isosceles triangles. This suggests that the more optimal solution involves reducing the polyhedral faces to triangles, regardless of whether the result is regular or irregular. For $N = 20$ there appears to be a random collection of different sized triangles in an incoherent mesh, however if looked at from certain angles, this is clearly not true (figure 6).

(a) $N = 8$ (b) $N = 20$

Figure 6: Different views of the output polyhedra.

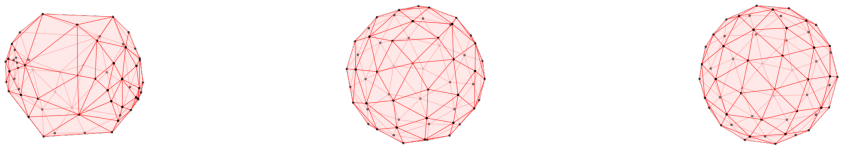
Both have multiple lines of symmetry and a unique pattern to them although with the $N = 20$ case the edges are not quite aligned. This suggests the true optimum solution will end with perfect alignment. As for the other three examples $N = 4, 6, 12$ the outputs are close but not quite as good as the Platonic solid solutions, albeit appearing like them. As a result the optimum solutions are most likely the Platonic solids themselves and if the algorithm ran for an infinite amount of time, the energy would approach that of (2), (3) and (5).

3.3 Common Attributes

It has been seen that a common trait is to create equilateral or isosceles triangles as the polyhedral faces but there are some more similarities between solutions. Pentagons and hexagons appear in all solutions frequently, regular and irregular, especially for large N. Small clusters of locally random looking configurations are also prevalent in very large systems, where the creation of an evenly spread mesh of triangles creates some distorted areas which are difficult to improve. Over time the algorithm smooths these out along with the rest of the mesh but the larger the amount of points in the system, the longer it takes to do this. Local minima are also a problem, where the algorithm has directed a point towards an area which seems to be optimal but the global minimum is elsewhere on the surface. This is a reason why the accept-reject step in the Metropolis algorithm is very useful as, with carefully defined parameters, the point may have a chance to escape these areas and move on to the optimum configuration.

3.4 Parameter Comparisons

Output differences can be seen depending on variation in parameters: size N, variance V, iterations I and uniform vs Gaussian generation types (figures 7, 8 & 9). In terms of iterations its clear that the longer the program is run for, the more accurate the outcome. The difference between figures 7a and 7b is huge compared to 7b and 7c, despite there being a tenfold increase of iterations in the first case and a hundredfold increase in the second. This may be a result of the algorithm plateauing, with very little change past a certain amount of iterations.



(a) $I = 1000$
$E = 1808.092155$

(b) $I = 10000$
$E = 1545.443486$

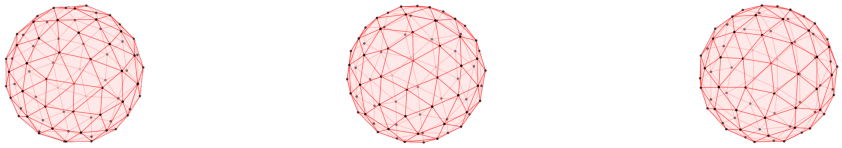
(c) $I = 1000000$
$E = 1543.857140$

Figure 7: The effect of number of Iterations I on the system energy for $N = 60$, $V = 0.00001$.

(a) Uniform
$E = 4463.882842$

(b) Gaussian
$E = 4449.315670$

Figure 8: Uniform vs Gaussian methods for $N = 100$, $I = 100000$, $V = 0.00001$.



(a) $V = 0.001$
$E = 2806.095916$

(b) $V = 0.0001$
$E = 2805.682371$

(c) $V = 0.00001$
$E = 2805.682825$

Figure 9: Variance changes for $N = 80$, $I = 100000$.

In the Gaussian vs uniform case it can be clearly seen that the Gaussian output is more consistent and the energy is lower which is expected since the previous solution is being improved on, rather than creating a new one at each step. The uniform generation allows for very rapid improvement at the start but when near the minimum it takes a long while to improve, meaning that to get the same results as a Gaussian generation it would take a few times as many iterations. The Gaussian generation is the opposite case, where it is slow to improve initially but is much better at refining the system to a more optimised state. The uniform generation method also allows for easy escape from local minima. An interesting turn is that of figure 9 where the change in variance doesn't seem to make a huge impact on the general output. Figure 9c is also slightly worse than 9b, the cause of which may again be the algorithm plateauing. Temperature is the final variable to be seen but is difficult to compare since it needs to be adjusted on a case by case basis, changing depending on all the other variables.

3.5 Smallest Known Energies

Typical Metropolis outputs can be compared with the current records of energy minima, which can be found at `https://en.wikipedia.org/wiki/Thomson_problem`, and calculate how close the algorithm gets (Table 1). An important note is that the Metropolis algorithm results use a different level of accuracy to that of the known minima so the differences must be considered with that taken into account. The key metric to look at here is the percentage difference, as the actual difference can be misleading of the algorithm's accuracy. The Metropolis algorithm appears quite consistent for small N, with little change in accuracy up to $N = 32$. After that the results can start varying wildly and are more luck based. It is much more difficult for the algorithm to improve as the number of points is increased and each small change can vary the system energy greatly. In general the Metropolis algorithm is very efficient and gives a very good result for its simplicity.

N	Metropolis Result	Known Minima	Difference	% Difference
4	3.674235	3.674234614	0.000000386	0.00001050558
6	9.985288	9.985281374	0.000006626	0.00006635763
8	19.675290	19.675287861	0.000002139	0.00001087151
12	49.165261	49.165253058	0.000007942	0.00001615368
20	150.881582	150.881568334	0.000013666	0.00000905744
32	412.261292	412.261274651	0.000017349	0.00000420825
60	1543.861837	1543.830400976	0.031436024	0.00203623559
80	2805.375951	2805.355875981	0.020075019	0.00071559616
100	4448.649992	4448.350634331	0.299357669	0.00672963293

Table 1: Comparison of typical Metropolis outputs after a million iterations with minima records.

3.6 Method Improvements

Although the Metropolis algorithm works well, there are definite flaws that can be improved upon. The initial system generation is done using the uniform distribution, with Gaussian distribution used after as the main comparison point generation type. A different approach could be to keep using the uniform distribution for a certain amount of iterations, since it is great at large, rapid improvements but not good at refinements, then reverting back to the Gaussian type. Variance could also be adjusted as the iterations progress, starting larger and lessening to improve the accuracy of the final result. The temperature could also be lowered over time like that of the simulated annealing technique [13] which helps to find of the global optimum while avoiding the trouble of local minima.

4 Wang-Landau Code

Full code documentation can be found in appendix B.

4.1 The Thomson Problem Problem

Because of the requirements of the Wang-Landau algorithm it is very difficult to apply it to the Thomson problem. The continuous energy spectra can be split, however the algorithm requires that an upper and lower bound is known so that all energy states can be visited. Unfortunately there is no upper bound on the system since the energy is calculated by the reciprocal of the distance, which tends to infinity as the points group together. A lower bound can always be set as zero but it is the infimum needed which is the aim of the problem itself. Thus a simpler example, which the Wang-Landau algorithm can be applied to, will be assessed.

4.2 Simple Harmonic Oscillator

The energy function to analyse is a simple harmonic oscillator potential given by $E(x) = x^2$. The minimum energy is zero for this system and the maximum will be restricted by the input domain e.g. for $x \in [-10, 10]$, $E(x) \in [0, 100]$. Analytically the density of states for a one dimensional simple harmonic oscillator is given by $g(E) \propto \frac{1}{\sqrt{E}}$ [14], so the algorithm's output can be compared to this.

4.3 Algorithm Walkthrough

The Python code is simple enough and generally follows the algorithm from 1.3.2. It begins by setting the initial values of f and $g(E)$ with a random starting energy. The main loop then runs with the starting modification factor adjusted after each loop until the predefined minimum is met. During these loops a flat histogram is made of the energy bins visited and the secondary loop begins. A comparison point is made and the energy is calculated for both the new and old points. This is then indexed to appoint each an energy bin for the transition probability step. The transition probability to the new state is tested and applied if successful, with $g(E)$ and $h(E)$ changed accordingly. The histogram is tested for flatness after each comparison. The resultant density of states values are finally output and plotted. The process can be visualised in figure 10.

4.4 Wang-Landau Results

Figure 11 shows the density of states for different bin sizes and indeed agrees with the expected outcome. No matter the size of the bins the general shape is always that of $g(E) = \frac{k}{\sqrt{E}}$ for various k. The density of states also agrees between different bin sizes i.e. the sum of $g(E)$ for $E \in [0, 2.5)$ and $E \in [2.5, 5)$ add up to that of $g(E)$ for $E \in [0, 5)$. Although this is a simple example it shows the effectiveness of the Wang-Landau algorithm.

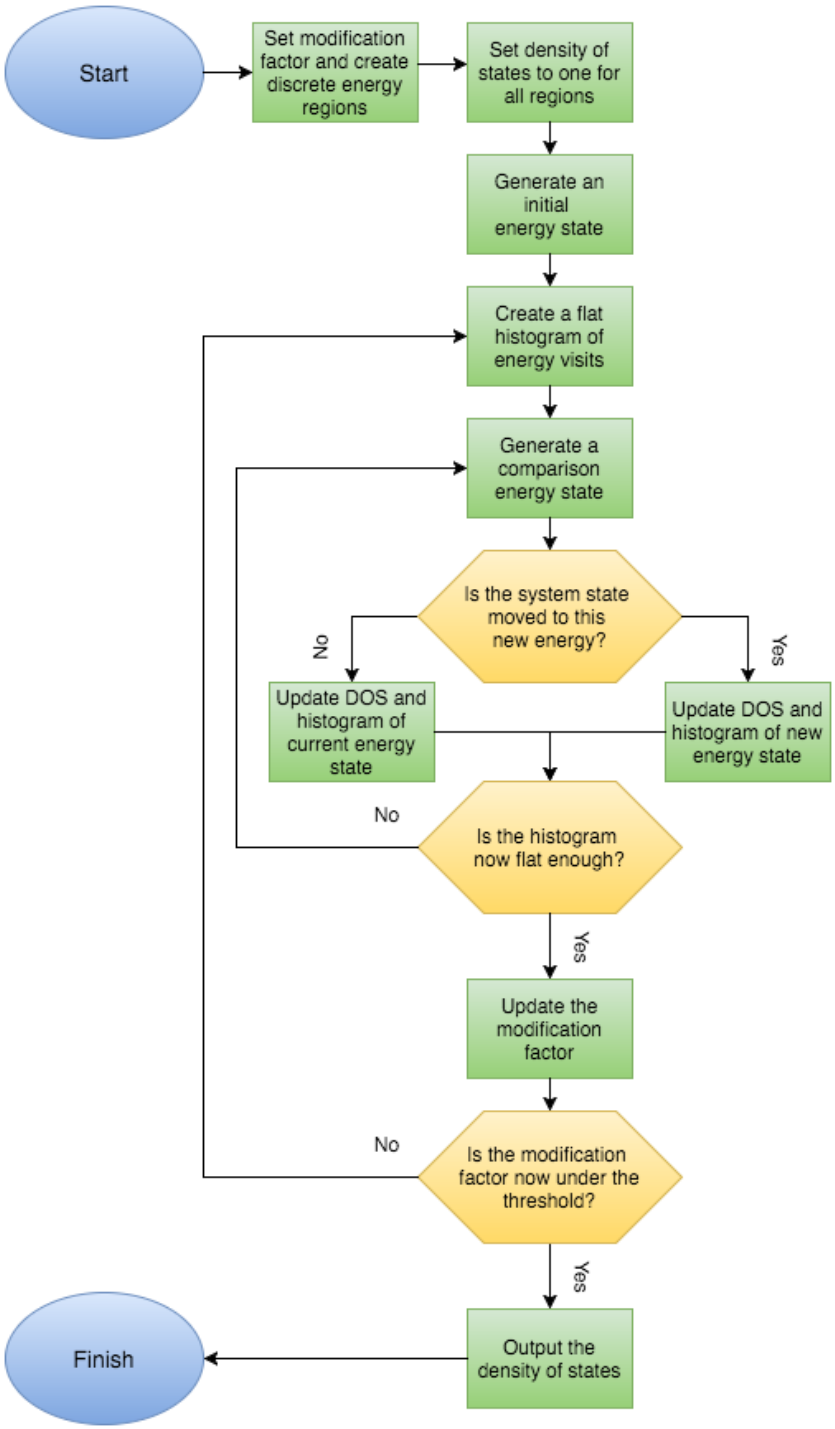


Figure 10: Workflow of the Wang-Landau algorithm.

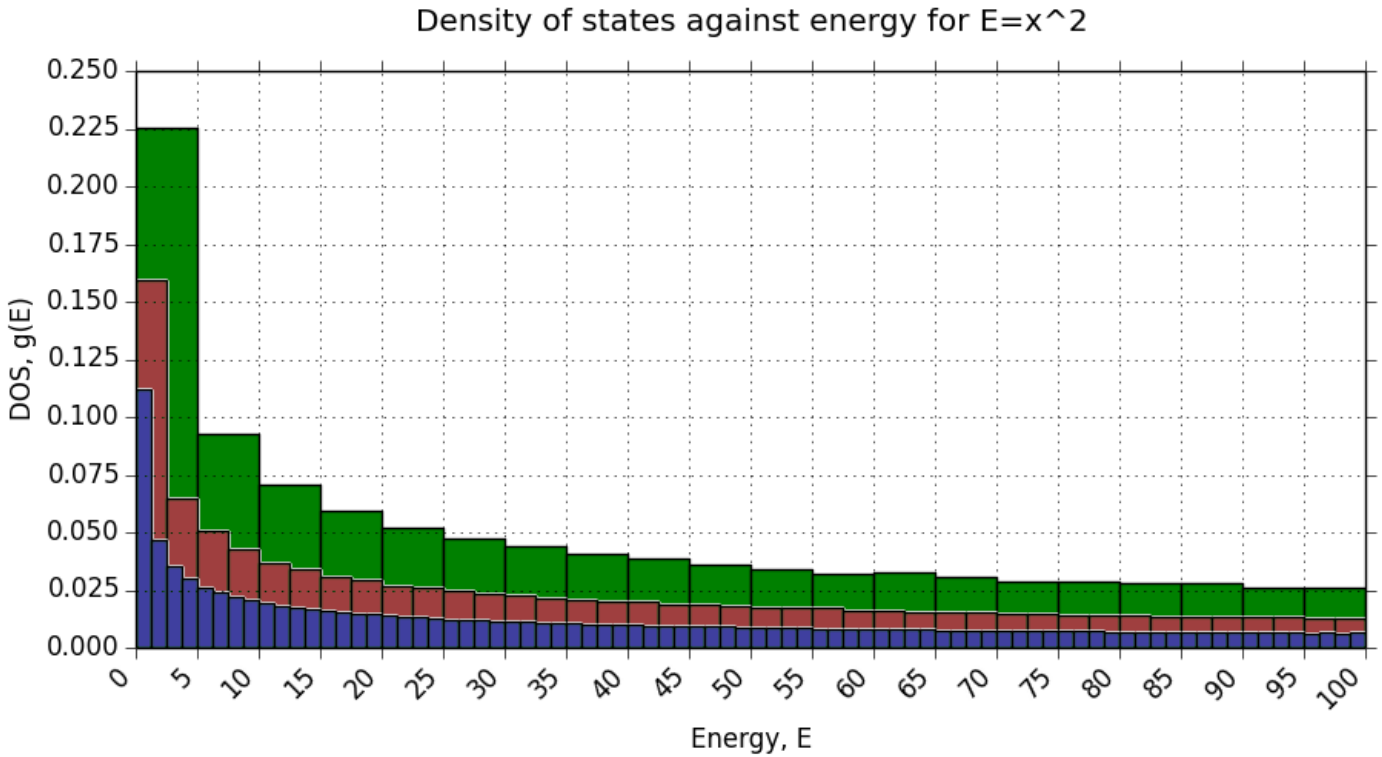


Figure 11: DOS with different discrete energy regions.

5 Conclusion

Two Monte Carlo algorithms have been looked at, the Metropolis Algorithm to create a Markov chain random walk in probability space and the Wang-Landau algorithm to find the density of states of a system. Both of these have been applied in Python with the different problems of system simulation seen in each, as well as how algorithm implementation works. The Metropolis Algorithm has been shown to work very well despite its simplicity, giving highly accurate results even for large values of N. Some weaknesses are apparent but with adaptations such as variance adjusting and simulated annealing, the process can be made even better. Larger systems saw problems in uniformity, with local clusters of grouped points being difficult to fix and apparent irregularities in point distributions. An alternative to Monte Carlo methods could be to look at other more complex algorithms such as genetic or relaxation algorithms and compare the results of each.

It has been seen that the Platonic solids and polyhedra in general are good indicators of optimal energy configurations, with three regular polyhedra being minima of the problem. Research could be done into non-regular polyhedra and polyhedra not confined to a sphere's surface to further explore the relationship between the Thomson problem and the shapes it creates. Although the Wang-Landau algorithm could not be applied to the Thomson problem, the density of states of the simple one dimensional harmonic oscillator was found that matched the analytic solution. Despite this being a simple problem a thorough understanding of the Wang-Landau algorithm was made so in the future the Thomson problem could be returned to and attempted again.

While no proofs were made about the solution minima, results agreeing with other's findings were established. This is reasonable enough considering the non-analytic nature of the problem, in fact only for the first several values of N does the problem have rigorous proofs. Overall the project was a success in understanding the Thomson problem and its attributes in addition to Monte Carlo algorithms and numerical optimisation methods.

References

[1] S. Smale (1998). *Mathematical problems for the next century*. The Mathematical Intelligencer Volume 20, Issue 2, Pages 7-15.

[2] J.J. Thomson (1904). *On the structure of the atom: an investigation of the stability and periods of oscillation of a number of corpuscles arranged at equal intervals around the circumference of a circle; with application of the results to the theory of atomic structure*. Philosophical Magazine Series 6, Volume 7, Issue 39, Pages 237-265.

[3] T. LaFave Jr. (2013). *Correspondences between the classical electrostatic Thomson problem and atomic electronic structure*. Journal of Electrostatics Volume 71, Issue 6, Pages 1029-1035.

[4] M. Bowick, A. Cacciuto, D.R. Nelson & A. Travesset (2002). *Crystalline order on a sphere and the generalized Thomson problem*. Physical Review Letters Volume 89, Issue 18, Page 185502.

[5] J.-S. Chen (2007). *Ground state energy of unitary fermion gas with the Thomson problem approach*. Chinese Physics Letters Volume 24, Number 7, Pages 1825-1828.

[6] L.-C. Tu & J. Luo (2004). *Experimental tests of Coulomb's Law and the photon rest mass*. Metrologia Volume 41, Number 5, Page S136.

[7] S. Raychaudhuri (2008). *Introduction to Monte Carlo simulation*. IEEE, Winter Simulation Conference 2008, Pages 91-100.

[8] F. Wang & D.P. Landau (2001). *Efficient, multiple-range random walk algorithm to calculate the density of states*. Physical Review Letters Volume 86, Issue 10, March 2001, Page 2050.

[9] C. Zhou, T.C. Schulthess, S. Torbrgge & D.P. Landau (2006). *Wang-Landau algorithm for continuous models and joint density of states*. Physical Review Letters Volume 96, Issue 12, Page 120201.

[10] S. Ramaswami (1993). *Convex Hulls: Complexity and Applications (a Survey)*.

[11] C.B. Barber, D.P. Dobkin & H. Huhdanpaa (1996). *The quickhull algorithm for convex hulls*. ACM Transactions on Mathematical Software Volume 22, Issue 4, Pages 469-483.

[12] A.M. Joan & R.J. Wilson (2003). *Graphs and applications: an introductory approach*. (Volume 1) Springer Science & Business Media.

[13] D. Bertsimas and J. Tsitsiklis (1993). *Simulated annealing*. Statistical science, Volume 8, Number 1, Pages 10-15.

[14] D.L. Feder (2013). *Physics 451 - Statistical Mechanics II - Course Notes*. Retrieved from `http://people.ucalgary.ca/~dfeder/451/notes.pdf`.

A Metropolis Algorithm Code

```
1  import numpy as np
2  import matplotlib as mpl
3  import matplotlib.pyplot as plt
4  from mpl_toolkits.mplot3d import Axes3D
5  from mpl_toolkits.mplot3d.art3d import Poly3DCollection
6  from scipy.spatial import ConvexHull
7
8  # Plot creation and parameters
9  fig = plt.figure()
10 ax = Axes3D(fig)
11 ax.set_aspect("equal")
12 ax.axis("off")
13 ax.set_xlim([-1,1])
14 ax.set_ylim([-1,1])
15 ax.set_zlim([-1,1])
16
17 def random_uniform_sphere(N): # N random points on a unit
       sphere using a uniform distribution
18  points=[]
19  for i in range(0,N):
20   theta = np.random.uniform(0,np.pi)
21   phi = np.random.uniform(0,2*np.pi)
22   points.append([np.cos(theta),np.sin(theta)*np.cos(phi),np
         .sin(theta)*np.sin(phi)])
23  return points
24
25 def random_gaussian_sphere(N,theta,phi,variance): # N
       random points on a unit sphere centered on a given point
        using the gaussian distribution
26  points = []
27  for i in range(0,N):
28   bm_rand1 = np.random.uniform(0,1)
29   bm_rand2 = np.random.uniform(0,1)
30   theta_gaus = np.sqrt(-2*np.log(bm_rand1))*np.cos(2*np.pi*
         bm_rand2)*np.sqrt(variance)+theta
31   phi_gaus = np.sqrt(-2*np.log(bm_rand1))*np.sin(2*np.pi*
         bm_rand2)*np.sqrt(2*variance)+phi
```

```
32    points.append([np.cos(theta_gaus),np.sin(theta_gaus)*np.
          cos(phi_gaus),np.sin(theta_gaus)*np.sin(phi_gaus)])
33   return points
34
35  def distance(point1,point2): # Distance between 2 points
36   return np.sqrt((point2[0]-point1[0])**2+(point2[1]-point1
          [1])**2+(point2[2]-point1[2])**2)
37
38  def local_monte_carlo_metropolis(points,iterations,
       temperature,type,variance): # Applies the Metropolis
       algorithm to a set of points
39   system_energy = 0
40   for i in range(0,len(points)):
41    for j in range(0,len(points)):
42     if j<=i:
43      continue
44     else:
45      system_energy += 1/(distance(points[i],points[j]))
46   print "starting energy = %f" % system_energy
47
48   for n in range(0,iterations):
49    i = np.random.randint(0,len(points)-1) # Pick a random
          point from the pointlist
50
51    if type=="uniform": # Generates the compared point by a
          uniform random distribution
52     random_point = random_uniform_sphere(1)[0]
53    elif type=="gaussian": # Generates the compared point by
          a local gaussian distribution centered on the chosen
          existing point
54     theta = np.arccos(points[i][0])
55     phi = np.arctan2(points[i][2],points[i][1])
56     random_point = random_gaussian_sphere(1,theta,phi,
           variance)[0]
57    else:
58     raise ValueError, "Invalid type"
59
60    old_point_energy = 0
61    new_point_energy = 0
```

```
62    for j in range(0,len(points)): # Compare the energies of
          the old and new point
63      if i==j:
64        continue
65      else:
66        old_point_energy += 1/distance(points[i],points[j])
67        new_point_energy += 1/distance(random_point,points[j])
68    if old_point_energy > new_point_energy: # The new point
          is improved so replaces the old point
69      points[i] = random_point
70      system_energy += (new_point_energy - old_point_energy)
71      print "energy down -> current energy = %f, energy change
            = %f" % (system_energy,2*(new_point_energy -
            old_point_energy))
72    else: # If the new point is not an improvement it still
          may be chosen according to its boltzmann probability
73      j = np.random.uniform(0,1)
74      if j<=np.exp((old_point_energy-new_point_energy)
            /(1.3806503*(10**-23)*temperature)):
75        # print "exp(delta(e)/kt = %f)" % np.exp((
              new_point_energy-old_point_energy)
              /(1.3806503*(10**-23)*temperature))
76        points[i] = random_point
77        system_energy -= (old_point_energy - new_point_energy)
78        print "energy up -> current energy = %f, energy change
            = %f" % (system_energy,2*(new_point_energy -
            old_point_energy))
79
80  print "final energy = %f" % system_energy
81
82  return points
83
84 def pointplot(points): # Displays a set of points in 3D
85  # Draws a sphere
86  phi = np.linspace(0,2*np.pi,200)
87  theta = np.linspace(0,np.pi,200)
88  xm = np.outer(np.cos(phi),np.sin(theta))
89  ym = np.outer(np.sin(phi),np.sin(theta))
90  zm = np.outer(np.ones(np.size(phi)),np.cos(theta))
91  ax.plot_surface(xm,ym,zm,alpha=0.05,linewidth=0,color="k")
```

```
92
93  # Draws the set of points
94  ax.scatter([i[0] for i in points],[i[1] for i in points],[
        i[2] for i in points])
95
96  def convexhull(points): # Creates a convex hull for a set
        of points
97  pa = np.asarray(points)
98  hull = ConvexHull(pa)
99  ax.scatter(pa[:,0],pa[:,1],pa[:,2],s=10,color='k',alpha=1)
100 for i in hull.simplices:
101   verts = [zip(pa[i,0],pa[i,1],pa[i,2])]
102   poly = Poly3DCollection(verts)
103   poly.set_facecolors((1,0.9,0.9,0.7))
104   poly.set_edgecolors((1,0,0,0.5))
105   ax.add_collection3d(poly)
106
107 def main(): # Calls the desired functions and displays them
108  iterations = 1000000
109  temperature = 5*10**16
110  type = "gaussian"
111  variance = 0.00001
112  rus = random_uniform_sphere(20)
113  lmcm = local_monte_carlo_metropolis(rus,iterations,
        temperature,type,variance)
114  #pointplot(lmcm)
115  convexhull(lmcm)
116  plt.show()
117
118 if __name__ == "__main__":
119     main()
```

B Wang-Landau Algorithm Code

```
from __future__ import division
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
from scipy.spatial import ConvexHull

f = np.exp(1)
N = 80
xmax = 10
Emax = xmax**2
Emin = Emax/N
E = np.linspace(Emin,Emax,N)
gE = [1]*(N)

print E,'\n\n','f_values:'

xold = np.random.uniform(-xmax,xmax)
while f >= np.exp(10**(-6)):
 print f
 flat_test = 0
 hE = [0]*(N)
 while flat_test <= 0.95:
  xnew = np.random.uniform(-xmax,xmax)
  Eold = (xold)**2
  Enew = (xnew)**2
  for index in range(0,N):
   if Eold <= E[index]:
    old_N = index
    break
  for index in range(0,N):
   if Enew <= E[index]:
    new_N = index
    break
  p = np.random.uniform(0,1)
  if p < min(gE[old_N]/gE[new_N],1):
   xold = xnew
```

```
39      gE[new_N] *= f
40      hE[new_N] += 1
41    else:
42      gE[old_N] *= f
43      hE[old_N] += 1
44    if sum(hE) != 0:
45      flat_test = min(hE)/(sum(hE)/len(hE))
46  f = f**(1/2)
47
48 print '\n',gE/sum(gE)
49
50 fig, ax = plt.subplots(figsize=(10,5))
51 plt.subplots_adjust(left=0.1,bottom=0.15)
52 ax.bar(E-E[0],gE/sum(gE),width=Emax/N,align='edge',color='
       green')
53 ax.grid()
54 plt.xticks(rotation=45,ha='right')
55 ax.tick_params(which='both',length=5,direction='out')
56 ax.set_xticks(np.linspace(0,Emax,21),minor=False)
57 ax.set_yticks(np.linspace(0,0.25,11),minor=False)
58 ax.set_xlabel('Energy, E')
59 ax.set_ylabel('DOS, g(E)')
60 ax.set_title('Density of states against energy for E=x^2',y
       =1.05)
61 plt.show()
```