Numerical Solutions for Midsize Tammes Problems: N = 61…100

By Laszlo Hars

Draft 01: December 6, 2020

Contents

1. Introduction

This document discusses the continuation of the work detailed in [14].

The referred document grew over 180 pages long, thus separating out the optimizations, which can be used for the larger cases of the Tammes problem made sense. We call, somewhat arbitrarily, the range of the problem size 61…100 as "midsize", based on their solvability on a cheap home computer. On a computer network or high-performance computers, the presented methods would lead to the numerical solutions of the Tammes problems above this range.

We use of the term "numerical solution" for

- finding arrangements of N points on the sphere
- with "large" shortest distances
- by numerical optimization methods
- starting from many random initial point sets.

The found shortest distances are close to the possible maxima, maximal at reasonably high probabilities.

The main contribution of this document is the speedup and the drastic reduction of the size of the encoding of the Tammes problem. The detailed description and listing of a *computer program*, which uses this optimized encoding together with open source, free tools, is also included. Using a cheap home computer, the discussed program finds good local optima at reasonable time. They are the solution of the Tammes problems at reasonably high probability. Still, if better point sets are found (with larger minimax distances), the table will be updated with the new data and with credit to the authors.

The program is placed in the *public domain*, anyone can use it for any purpose.

2. Computational Tools: Software and Hardware

As the **hardware**, a home-built PC was used, with:

- CPU: AMD Ryzen 5 2600 (64-bit six-core Intel compatible processor)
- Clock: 3400 MHz
- Memory: 16 GB DRAM

Software:

- Operating system: Windows 10
- Programing language: Julia 1.5 [10]
- Nonlinear optimization library: NLopt version 2.6.2 [11]
- Julia interface to the NLopt library: NLopt Julia module [12]
- Interactive plotting program: gnuplot 5.5 [13]

Resource use for a single optimization thread:

- CPU: 9.5%
- Memory: < 250 MB (N=80)

One can run multiple optimization threads in parallel. We did not see significant slow-down when running up to 6 independent numerical optimization processes in our platform, together with some everyday tasks, like email, document editing, etc. Due to hyperthreading support, the CPU can gracefully handle 10 threads, but with proportional slowdown compared to 6 optimization threads.

3. Need for Speed for the Numerical Solution of the Tammes Problem

The random-restart numerical optimizations we performed for small size Tammes problems in [14] get too slow as N increases beyond 60. At N = 60, in a single thread of a personal computer several days are needed for the number of restarts we have to perform.

The minimum number of restarts = 50,000…100,000 for reasonable confidence in that the best numerical optimum found is actually the global optimum, the solution of the corresponding Tammes problem. Experiments with the original optimization procedures indicated that close to N = 100 the desired number of random restarts is at least 200,000, which takes 2 months of continuous work of our home PC, in a single thread. In this work we present techniques, which provide at least a 3-fold speedup, in most of cases more than 10-fold.

4. The Tammes Problem

The Tammes problem asks to place a given number N points on the surface of a sphere such that the minimum distance between these points is maximal. The problem is named after a Dutch botanist, P. M. L. Tammes, who posed the problem in 1930 while studying the distribution of pores on pollen grains [1].

The Tammes problem can equivalently be formulated as finding the set of N congruent, pairwise non-overlapping circles on a sphere, which are the possible largest (circle packing).

Only small instances of the Tammes problem have been solved exactly: N = 3…14 and N = 24. Other instances remained unsolved, only special constructions and numerical (approximate) solutions exist. Using accurate number representations (e.g., 64-bit numbers), and setting the tolerance of the numerical optimizations to small values (e.g., to 10^{-16}) could provide 15 decimal digit accuracy, more than enough for practical applications.

Instead of the distances between pairs of points, any strictly monotonic function of these distances can be used. The simplest is the cosine of the angle between the vectors pointing to the spherical points from the center of the sphere. The unit sphere centered at the origin will be used in this document, when the cosine of this angle is the dot product of the vectors pointing to the two points:

For points $A = (x_A, y_A, z_A)$ and $B = (x_B, y_B, z_B)$, given by their Cartesian coordinates

$$\cos(AB\sphericalangle) = x_A x_B + y_A y_B + z_A z_B$$

The Tammes problem is equivalent to finding the minimum (over all sets $\mathcal{S}$ of N spherical points) of the largest $\cos(AB\sphericalangle)$ values (over all pairs of points A, B in the set), which is a typical mini-max optimization problem.

Maximum(.) and Minimum(.) are not differentiable functions, but the usual trick of employing an extra variable d with inequality constraints as below, makes the optimization problem smooth (differentiable) *inside* of the optimization domain:

Objective:

$$d \longrightarrow min$$

Subject to:

$$\forall (A, B) \in \mathcal{S}^2 : \cos(AB\sphericalangle) \leq d$$

5. Problem Encoding

In the Numerical_Tammes document [14] several problem encodings were discussed. The Marsaglia transform based encoding was found one of the best among them, which is adopted for this study.

The spherical points of the Tammes sets are mapped to the 2-dimensional plane. A couple of optimization variables can be eliminated with points fixed: P_{-1} to (0,0,1) and P_0 to $\left(\sqrt{1-d^2}, 0, d\right)$, where d is the optimization variable, the shortest distance among the points of the spherical set of cardinality N. It leaves 2N−3 free variables to optimize, which our Julia program stores in an array `x[1:2N-3]`. The spherical point P_i is mapped to the planar point (u_i, v_i), which is encoded as `(x[2i],x[2i+1])` for `i = 1..N-2`, and `d = x[1]`.

The Marsaglia transform [3] maps the points (u, v) inside the unit 2D disk, to the 3D points (x, y, z) on the sphere, with uniform areal density:

$$u^2 + v^2 < 1$$

$$x = 2u\sqrt{1-u^2-v^2}$$

$$y = 2v\sqrt{1-u^2-v^2}$$

$$z = 1 - 2\,(u^2+v^2)$$

The transform is not defined on the perimeter of the unit disk, when $u^2 + v^2 = 1$, that is, the spherical point (0,0,−1) is missing from the image. We fixed the point P_{-1} to (0,0,1), therefore it is more convenient to flip the sign of z. When used in numerical optimizations for the Tammes problem, this flipped Marsaglia transform has no problems with singularity: in feasible sets, no points get close to the fixed point (0,0,−1).

The inverse of the Marsaglia transform, needed for encoding the spherical points, is derived in [14]:

$$u = x\sqrt{\frac{1-\text{sign}(z)\sqrt{1-x^2-y^2}}{2x^2+2y^2}}; \quad v = y\sqrt{\frac{1-\text{sign}(z)\sqrt{1-x^2-y^2}}{2x^2+2y^2}}$$

6. Speed-up Ideas

As discussed above, the numerical optimizations have to be made faster in order to handle the Tammes problem for up to N = 100 on a home computer. Below the tweaks are described, which provided more than 3-fold speed-up, in most instances of the Tammes problem, over 10-fold.

6.1. Marsaglia Encoding without Circle-Constraints

The domain of the Marsaglia transform is the interior of the unit circle on the plane. To restrict the points there requires N−2 nonlinear constraints, that slows down the numerical optimization and prevents the use of certain optimization algorithms. One can remove the nonlinear constraints $u^2 + v^2 < 1$ by replacing $u^2 + v^2$ by $\min(1, u^2 + v^2)$ in the calculations, where applicable:

```
function XYZR(x::Vector)                    # Marsaglia transform
    N = (3+length(x))>>1
    X,Y,Z,R = zeros.(fill(N-2,4))
    for i = 1:N-2
        s = min(1.,x[2i]^2 + x[2i+1]^2)     # Planar points [u,v]: u[i] ~ x[2i], v[i] ~ x[2i+1]
        R[i] = r = sqrt(1-s)
        X[i] = 2x[ 2i ] * r
        Y[i] = 2x[2i+1] * r                 # coordinates of the SPHERICAL POINTS ---
        Z[i] = 2s - 1
    end
    return X,Y,Z,R
end
```

In the vicinity of a local optimum the numerical optimization trajectory does not get close to the boundary of the unit circle, so the convergence does not change there, but while the search trajectory is far from a local optimum, some search points could get outside of the circle. Our modified computation evaluates this search point as a poor solution candidate; thus, the numerical optimization rejects these search points. (If s = 1 then r = 0, and the corresponding spherical point P = (0,0,1) = P_{-1}.)

Note that with this modification at computing the gradients a division by 0 could result, which needs proper handling. Fortunately, the Julia programming language we used, handles this exception gracefully (giving a quotient = ∞, denoted by **`Inf`**).

A potential problem is that some initial population may cause the numerical optimization (which were convergent when the circle-constraints were used) to diverge, therefore somewhat more random restarts might be necessary. Experiments showed about 1.6% overall failure rate, which is insignificant.

6.2. Disregarding Point-Pairs Intially Far Away

In the initial population of spherical points, the distances between pairs of points can be computed, and only those are considered in the constraints, which are not "too large" (by setting various thresholds for experimenting).

The idea is that starting from a homogenous initial population of spherical the numerical optimization may not need to move points by much, preserving the "closeness" of points, thus point pairs at large initial distances will likely remain far during the optimization.

This modification did provide speed improvements, but setting a distance-threshold, which noticeably reduces the number of constraints, also makes the number of necessary random restarts larger (to find the true optimum). Also, the numerical optimizations produced some invalid Tammes sets, which had to be discarded.

The straightforward implementation takes $O(N^2)$ time during initialization of a numerical optimization step, which is just a small percentage of the overall numerical optimization time. Nevertheless, various improvements could be implemented, even though the expected speed-up is negligible.

6.2.1. Clusters

If there is a cluster of points, each close to the others, most of the initial short distances are among these points, and the optimization would likely result in infeasible solutions. Therefore, the initial point population has to be cluster-free, which can be achieved with Mitchell's algorithm [6], discussed more in Section 6.4. (Generate the initial points one-by-one; choose the next point from M uniformly random

candidate points, which is the farthest from the closest points already generated.) However, large M values make the initial populations less diverse, and that makes more random restarts necessary to find the true, global optimum.

Overall, we could not achieve significant speed-ups, but more experiments and trying various further improvement ideas could make this approach worthwhile.

6.3. Disregarding Far-Away Point-Pairs

Another speed-up idea is dynamically excluding far away point pairs from the constraints. This is time consuming, as at each evaluations of the constraints we have to look for all distances to be able to find the point pairs close by.

6.3.1. Underlying Idea

When the multi-dimensional search point, which the numerical optimization moves toward a solution, is *far from a local optimum*, a step the algorithm makes in the direction of the gradient may cause that our next set of adaptive constraints will be different. It could confuse the optimization algorithm with the inconsistencies between its past and current constraint sets. However, these improving steps are not very reliable far from a local optimum. The result of dynamically changing constraints is a different, but not necessarily inferior search trajectory.

Close to a local optimum our adaptively updated constraint sets remain mostly the same, not slowing down the numerical optimization algorithm in its search for the optimum.

6.3.2. The Number of Adaptive Constraints

The number of constraints cannot change during a numerical optimization run (because the internal data structures are already set up), thus we have to set this number to a fix value, like C = 2.5N, and select the number of the considered closest point pairs accordingly. It would be sufficient only to separate out the smallest C distances, and derive constraints from them. This is a relatively slow, $\mathrm{O}(N^2)$ time algorithm working on $\binom{N}{2}$ distances.

Note that one can replace the search for the C closest point-pairs with more sophisticated methods, including the spherical Voronoi diagram, or Delaunay triangulation. (The best published algorithms for these take $\mathrm{O}(N \log N)$ time, which could provide noticeable speed-ups for very large N values, but due to the complexity of the algorithms we don't expect improvements for moderate sized problems.)

Setting the best C value is a delicate process. Too many critical constraints make the corresponding constraint-set to change often, breaking the convergence. Too few constraints could result in infeasible result sets.

Experiments showed the optimum is about 2.375N. This sounds reasonable, because

- Far from a local optimum the numerical optimization trajectory does not promise predictable increase of the minimax distance, thus a different trajectory caused by the small critical constraint set may only cause some "random" change to the path to a local optimum. Therefore, reasonably small critical constraint sets only make random variations of the number of iterations needed to get into the vicinity of a local optimum.

- Close to a local optimum the critical constraints concern with neighboring nodes in the contact graph of the Tammes set. The maximum degree of each vertex is 5, thus the maximum size of the true critical constraint set is 2.5N. In reality most nodes of the contact graph are of less than degree 5, so the most promising sizes are between 2N and 2.5N.

6.3.3. Update Schedule of the Critical Constraints

Our attempts of reducing the update frequency of the adaptive constraint-sets, failed. E.g., at the first N/2 evaluations of the constraints we always perform updates, then update the constraint at only every other constraint evaluation call. The underlying idea is that far from a local optimum the critical constraint set changes often, but close to a local optimum it stays mostly the same. However, in real life situations the optimization trajectory may get close to several local optima, but abandon them for a more promising direction. Some such trajectories are long, others are short, therefore, a fix schedule causes many numerical optimizations to fail to converge.

Similarly, considering the rate at which the value of the objective function f changes, can also be misleading: close to a sharp peak f may still change fast, and far from a small peak, it may change slowly.

A little more intelligent update schedule for the critical constraints looked more promising: consider the last few f values. Close to a peak the improvements get progressively smaller, by a quadratic trend – when using a gradient based optimization. If we notice this happening, we can skip updates of the critical constraints. However, close to a local minimum, the numerical optimization only performs few improvement steps, and accelerating these few steps do not promise significant overall speedup.

Reducing the number of constraints by a factor proportional to N gives large speedup, even if the reduction takes $\mathrm{O}(N^2)$ time, because the numerical optimizations steps took more than $\mathrm{O}(N^3)$ time, each (see in [14]). On the other hand, a sublinear or even linear reduction of the number of updates of the critical constraints promises much less of an impact.

The corresponding investigations are still unfinished. Any input from the reader is welcome.

6.4. Homogenous Intial Point Set: Mitchell's Algorithm

Putting N uniform random points on the sphere is straightforward, (e.g., rejecting samples of uniform random points in a cube, which are outside of the inscribed sphere, then projecting the remaining points to the surface of the sphere). However, in such population of points, there are clusters (points close to each other) and large empty areas, at high probability. One could start the numerical optimizations from different, not uniform random, but more homogenous initial populations. That is, because the optimum point arrangements do not have clusters.

One idea was to use Poisson-disc sampling, where no points are less than a minimum distance apart, but efficient implementations are quite complex. Mitchell's best-candidate algorithm [6] is a straightforward approximation of the Poison-disc distribution: Having generated a few points of the set, generate a number of new candidate points and pick the one furthest from all previous samples. More precisely:

- Start with any point, and in each further step deliver one more point
- In each step generate M uniform random candidate points
    - Find the shortest distance from the already placed points and each of the candidate points
    - Select the candidate, which has the largest of the just calculated shortest distances, discarding the other M−1 candidates.

Setting the optimum M value needs experimenting. Too large numbers of Mitchell iterations, makes the initial population too homogenous, too regular, and does not provide enough variations to find global optima. Too small M makes inhomogeneous initial populations, where many optimizations diverge, or lead to inferior local optima.

To be able to compare different settings, we printed out the number of convergent and divergent numerical optimizations, and also looked at how many restarts are needed to find the best known Tammes sets.

In the experiments, as expected, small Mitchell numbers caused too many divergent optimization calls.

Large Mitchell numbers made the number of necessary restarts consistently larger until the best, already known Tammes sets are found. These tests are harder to evaluate, because a lucky initial population can occasionally lead to a good Thomson set early, but performing the numerical optimizations with hundreds of different randomness seeds does exhibit a discernable trend.

The performed experiments showed the optimal Mitchell number around 5.

7. The Adaptive Random-Restart Numerical Optimization Program

The following one was the fastest of the hundreds of variations of the numerical optimization computer programs we have tried.

It starts with setting the parameters and defining an asynchronous task for catching keystrokes, which signal the user's request to stop early.

```
using NLopt
using Printf

if length(ARGS) > 0                         # N = number of circles packed on the sphere
   N = parse(Int,ARGS[1])
else
   print("Enter N: "); N = parse(Int,readline())
end
!(2 < N < 1000) && error("2 < N < 1000 needed, got $N")

MAXITERS = round(Int,2^(N/23))*10_000
MITCHELL = 5
CONSTRNT = round(Int,2.375N)                # OK: 2.2N...3.5N; max: ((N+1)*(N-2))>>1
I = fill(0,CONSTRNT); J = similar(I); cntCalls = 0
D = Dict{Symbol,Integer}()                  # create an empty dictionary
D[:Infeasible] = 0

DIAG = false                                # print information on inaccurate results?
mxeval = 3N                                 # starting value of estimated #evaluations of f()
seed = 0                                    # seed for hash-counter RNG   (UInt64)
eps = 1e-10                                 # tolerance in optimizations

filename = pwd()*"\\MAD\\CircPack-$(N).txt"
                                            # === Non-blocking, non-echoed keyboard input ===
ccall(:jl_tty_set_mode,Cint,(Ptr{Cvoid},Cint),stdin.handle,1)==0 ||
  throw("Terminal cannot enter raw mode.") # raw terminal mode to catch keystrokes
const chnl = Channel{Array{UInt8,1}}(0)     # unbuffered channel for key codes
@async put!(chnl,readavailable(stdin))      # async task catching keystroke: isready(chnl) ?
```

Next the objective function is defined, which returns the first optimization variable x[1], containing the cosine of the shortest distance among the spherical points. Since the cosine is a decreasing function, when it is minimized, the shortest distance is maximized, which is the objective of the Tammes problem.

```
function f(x::Vector, grad::Vector)         # cos(a) -> min, a = spherical distance -> max
   length(grad) > 0 && (fill!(grad,.0); grad[1] = 1)
   return x[1]
end
```

The function definition for the Marsaglia transform follows, which maps the planar points $(u_i, v_i) = (\mathtt{x[2i]}, \mathtt{x[2i+1]})$ to the spherical points $P_i = (X_i, Y_i, Z_i)$.

```
function XYZR(x::Vector)                    # Marsaglia transform (with –Z)
   N = (3+length(x))>>1
   X,Y,Z,R = (similar(1.:N-2) for _=1:4)
   for i = 1:N-2                            # Planar points [u,v]: u[i] ~ x[2i], v[i] ~ x[2i+1]
      s = min(1.,x[2i]^2 + x[2i+1]^2)       # outside of unit circle: (u,v) -> (0,0,1)
      R[i] = r = sqrt(1-s)
      X[i] = 2x[ 2i ] * r
      Y[i] = 2x[2i+1] * r                   # coordinates of the SPHERICAL POINTS ---
      Z[i] = 2s - 1
   end
   return X,Y,Z,R
end
```

The above XYZR function also returns the helper vector R, containing $\sqrt{1-u_i^2-v_i^2}$, which is useful for computing the gradients of the constraints. Note that if $u_i^2 + v_i^2 > 1$ the square root is not defined as real number, and it could happen early during the numerical optimization. To prevent that we replaced its value with $\max(1, u_i^2 + v_i^2)$, forcing it to the feasible range. This change does not alter the optimization process near a local optimum, because close to $u_i^2 + v_i^2 = 1$ the corresponding P_i would be too close to P_{-1}, which does not lead to solutions of the Tammes problem. Accordingly, the constraints remain differentiable near all local optima.

This simple change also made the nonlinear constraints of the Marsaglia transform $u_i^2 + v_i^2 \leq 1$ implicit, and so unnecessary.

The next function, **`dists()`**, finds a given number (**`CONSTRNT`**) of point pairs, which are closer to each other than the other point pairs. First all the cosine-distances are computed (**`D[]`**), then the indices of the smallest **`CONSTRNT`** distances are found. Only explicitly requiring that these are smaller than **`x[1]`** represents the "critical" set of constraints.

This is a linear time operation on $\binom{N}{2}$ distances with Julia's **`partialsort`** built-in function.

```
function dists(Z0::Float64, X::Vector, Y::Vector, Z::Vector)
   N = length(X)+2
   M = ((N+1)*(N-2))>>1
   D = similar(1.:M)
   I,J = (similar(1:M) for _=1:2)
   k = 0; X0 = sqrt(1-Z0^2)
   for i = 1:N-3, j = i+1:N-2
      k += 1                                    # dist(Pi,Pj)
      D[k] = X[i]*X[j] + Y[i]*Y[j] + Z[i]*Z[j]
      I[k],J[k] = i,j
   end
   for j = 1:N-2
      k += 1                                    # dist(P0,Pj)
      D[k] = X0*X[j] + Z0*Z[j]
      I[k],J[k] = 0,j
      k += 1                                    # dist(P[-1],Pj)
      D[k] = Z[j]
      I[k],J[k] = -1,j
   end
   II = D.>=partialsort(D, CONSTRNT, rev=true) # select distances > CONSTRNT'th largest entry
   return D[II],I[II],J[II]
end
```

The following function, **`constr()`**, computes the critical constraints and their gradients for the numerical optimizations.

The constraints are simple: the cosine of every distance in the critical constraint set must not be larger than the cosine of the shortest distance, **`x[1]`**.

After the critical constraints, their derivatives are computed, to form the gradients. The chain rule is applied: the partial derivatives of the Marsaglia transform encoding are computed first, which are then multiplied by the partial derivatives of the constraints of the critical cosine-distances.

```
function constr(res::Vector,x::Vector,grad::Matrix)
   N = (3+length(x))>>1
   X,Y,Z,R = XYZR(x)
   X0 = sqrt(1-x[1]^2)

   D,I,J = dists(x[1], X,Y,Z)
                                          # --- CRITICAL CONSTRAINTS ---
   for k = 1:CONSTRNT                     # cos(P[i]-P[j] distances) ≤ x[1]
      res[k] = D[k] - x[1]
   end

   if length(grad) < 1 return res; end    # --- GRADIENT ---

   Xu,Xv,Yv = zeros.(fill(N-2,3))         # --- PARTIAL DERIVATIVES of the encoding
   for i = 1:N-2
      Xu[i]= -2*(2x[2i]^2+x[2i+1]^2-1) / R[i]  # dX/du
      Xv[i]= -2 * x[2i] * x[2i+1] / R[i]       # dX/dv = dY/du
      Yv[i]= -2*(2x[2i+1]^2+x[2i]^2-1) / R[i]  # dY/dv
   end
   Yu = Xv                                # dZ/du = 4u, dZ/dv = 4v

   fill!(grad, 0.0)
   for k = 1:CONSTRNT                     # distances of close-by P[i]-P[j] < x[1]
      i,j = I[k],J[k]
      if i > 0
         grad[ 1,  k] = -1
         grad[ 2i, k] = Xu[i]*X[j] + Yu[i]*Y[j] + 4x[ 2i ]*Z[j]
         grad[2i+1,k] = Xv[i]*X[j] + Yv[i]*Y[j] + 4x[2i+1]*Z[j]
         grad[ 2j, k] = Xu[j]*X[i] + Yu[j]*Y[i] + 4x[ 2j ]*Z[i]
         grad[2j+1,k] = Xv[j]*X[i] + Yv[j]*Y[i] + 4x[2j+1]*Z[i]
      elseif i < 0                        # distances P[-1]-P[i]
         grad[ 1,  k] = -1
         grad[ 2j, k] = 4x[ 2j ]
         grad[2j+1,k] = 4x[2j+1]
      else                                # distances P[0]-P[i]
         grad[ 1,  k] = -x[1]/X0 * X[j] + Z[j] - 1
         grad[ 2j, k] = X0*Xu[j] + 4x[1]*x[ 2j ]
         grad[2j+1,k] = X0*Xv[j] + 4x[1]*x[2j+1]
      end
   end

   return res
end
```

It is possible that the numerical optimization algorithm returns infeasible solutions (although it happened extremely rarely), therefore the results always have to be verified. This is done in the following **`verify()`** function:

```
function verify(x::Vector)                          # check if x[] is feasible
   N = (3+length(x))>>1
   OK = trues(1)                                    # (scalar is not accessible inside loops)
   X,Y,Z = XYZR(x)
   X0 = sqrt(1-x[1]^2)

   for i = 1:N-3, j = i+1:N-2
      if X[i]*X[j]+Y[i]*Y[j]+Z[i]*Z[j] > x[1]+eps  # points are too close
         DIAG && OK[1] && println("   --- FAIL --->");  OK[1] = false;
         DIAG && println("cos(dist[$i,$j]): $(X[i]*X[j]+Y[i]*Y[j]+Z[i]*Z[j]) > $(x[1]) = cos(a)")
   end   end

   for i = 1:N-2
      if Z[i] - x[1] > eps                          # P[-1] and P[i] are too close
         DIAG && OK[1] && println("   --- FAIL --->");  OK[1] = false;
         DIAG && println("cos(dist[-1,$i]): $(Z[i]) > $(x[1]) = cos(a)")
      end

      if X0*X[i] + x[1]*Z[i] - x[1] > eps           # P[0] and P[i] are too close
         DIAG && OK[1] && println("   --- FAIL --->");  OK[1] = false;
         DIAG && println("cos(dist[0,$i]): $(X0*X[i] + x[1]*Z[i]) > $(x[1]) = cos(a)")
      end

      if x[2i]^2 + x[2i+1]^2 - 1 > eps              # (u,v) is not in unit disk
         DIAG && OK[1] && println("   --- FAIL --->");  OK[1] = false;
         DIAG && println("Norm^2[$(2i)] = $(x[2i]^2 + x[2i+1]^2)")
      end
   end

   OK[1] || DIAG && (@printf("   cos(a) = % 1.9f\n",x[1]);
      for i = 1:N-2 @printf("% 1.9f   % 1.9f   % 1.9f\n", X[i],Y[i],Z[i]) end)
   return OK[1]
end
```

If the global variable **`DIAG`** is set to **`true`**, the function also prints debug information out, in addition to returning **`true`** or **`false`**, telling if the supplied data represents a feasible (approximate) solution of the Tammes problem.

The random initial point population is computed by a “bit mixer”, a hash-function like construct, transforming fix size inputs to fix size outputs, which are not correlated when tested by standard statistical- and randomness tests. See in [8] and [9]. The input consists of the restart-number, the number of uses within this iteration, and a 32-bit seed number. (This way the *same sequence of initial point populations* can be generated on any 64-bit computer, independent on the architecture and the programing environment.)

```
r64(n::Unsigned,d::Integer) = (n<<d) | (n>>(64-d)) # 64-bit left-rotation

function h(x::Integer, y::Integer)           # bit-mixer
   x,y = UInt64.(unsigned.([x,y]))
   for i = 1:7
      x = xor(x,r64(x,5), r64(x,9) ) + 0x49A8D5B36969F969
      y = xor(y,r64(y,59),r64(y,55)) + 0x6969F96949A8D5B3
   end
   return x+y
end

function srnd(i::Integer, j::Integer)        # uniform pseudorandom spherical points by hashing
   t = (significand(Float64(h(i+0xC90FDAA22168C235,j)))-1.) * 2pi
   u = 2significand(Float64(h(i,j+0x5A827999FCEF3242)))-3.
   s = sqrt(1-u^2)
   return [s*cos(t); s*sin(t); u]
end
```

The generation of the initial pseudorandom point population is done in two parts. First Mitchell’s algorithm is performed to generate homogenous spherical points (which do not likely have clusters of points all within a small distance). It is governed by the global variable MITCHELL.

```
function init(N::Integer, iter::Integer, seed::Integer)   # set P[1..N-2]; implicit P[-1],P[0]
   X,Y,Z = (similar(1.:N) for _=1:3)
   vxx,vx,v = zeros.((3,3,3))

   X[1],Y[1],Z[1] = 0,0,1        # first 2 points are special
   s = (significand(Float64(h(iter,seed)))-1.5)*2.5
   X[2],Y[2],Z[2] = cos(s), .0, sin(s)   # P2 is not too far/close to P1 (*pi: full Z range)

   for i = 3:N                   # random population from repeatable pseudorandom sequence
      c = [-1.0;1.0]
      for k = 1:MITCHELL         # a few random candidate points for P[i]...
         c[1] = -1.0             # keep point of largest shortest distance, preserve randomness
         v[:] = srnd(iter+i<<32,seed+k<<32)
         for j = 1:i-1           # find largest cos(distance(Pk,Pj)) ~ smallest distance
            t = v[1]*X[j] + v[2]*Y[j] + v[3]*Z[j]
            if  c[1] < t
               c[1] = t
               vx[:] = v
            end
         end                     # vx is the closest to Pk, cos(dist) = c[1]
         if  c[2] > c[1]
            c[2] = c[1]
            vxx[:] = vx          # vxx is the point with the largest minimal distance
         end
      end
      X[i],Y[i],Z[i] = vxx
   end
```

...

The second part of generating initial points finds the point pair at the shortest distance and rotates all the points, such that the closest points get to the special position of P_{-1} and P_0.

...

```
   closest = -ones(3)                  # closest points (i,j) and their distance
   for i = 1:N-1, j = i+1:N
      t = X[i]*X[j] + Y[i]*Y[j] + Z[i]*Z[j]
      t > closest[3] && (closest[:] = [i, j, t])
   end

   i,j = Int.(closest[1:2])            # i < j
   u = [X[i],Y[i],Z[i]];  v = [X[j],Y[j],Z[j]]
   if j < N-1                          # i and j are small: replace P[i] and P[j]
      X[i],Y[i],Z[i] = X[ N ],Y[ N ],Z[ N ]
      X[j],Y[j],Z[j] = X[N-1],Y[N-1],Z[N-1]
   elseif i < N-1                      # i is small, j is large: replace P[i] but not with P[j]
      k = 2N-1 - j
      X[i],Y[i],Z[i] = X[k],Y[k],Z[k]
   end                                 # i = N-1, j = N: no replacement

   s,c = u[2:3] ./ sqrt(u[2]^2+u[3]^2)
   R = [1 0 0; 0 c -s; 0 s c]          # Rotate around axis x: u.y <- 0

   u = R * u
   s,c = u[1:2:3] ./ sqrt(u[1]^2+u[3]^2)
   R = [c 0 -s; 0 1 0; s 0 c] * R      # Rotate around axis y: u.x <- 0, too

   u = R * v
   s,c = u[1:2] ./ sqrt(u[1]^2+u[2]^2)
   R = [s c 0; c -s 0; 0 0 1] * R      # Rotate around axis z: v.y <- 0, too
   (R*v)[1] < 0 && (R[1,:].*=-1)       # Reflect if needed: v.x > 0

   x = similar(1.:2N-3)                # optimization variables
   x[1] = closest[3]
   for i = 1:N-2                       # Rotate the random points P[1...N-2], P[-1],P[0] fixed
      u,v,w = R * [X[i],Y[i],Z[i]]
      s = u^2+v^2
      if s < 1e-99                     # P[i] = (0,0,-1): each point on the unit circle works
         x[2i:2i+1] = [1.,0.]
      else
         r = sqrt( (1 + copysign(sqrt(1-s),w))/2s )
         x[2i:2i+1] = [u,v].*r         # inverse Marsaglia transform: Sphere -> Plane
      end
   end
   return x
end
```

Before the restart-loop of numerical optimizations starts, some settings have to be made for the file of the results and for the optimization algorithm, and its data structures:

```
fl = open(filename,"w")
pos = 0                                  # seek point in the file fl
println("Output: $filename")

fm = 1.0;  xm = similar(1.:2N-3);  tm = time()
s = "waiting for first results"

opt = Opt(:LD_SLSQP, 2N-3)               # OPTIMIZATION ALGORITHM (LN_COBYLA, LD_SLSQP...)
opt.min_objective = f
opt.ftol_abs = eps
opt.lower_bounds =-ones(2N-3)            # could be omitted, enforced by the constraints
opt.upper_bounds = ones(2N-3)
inequality_constraint!(opt, constr, fill(eps,CONSTRNT))
```

The actual random restart loop is the following. It performs the numerical optimizations, verifies the result, updates the average number of function calls used to set the maximum allowed, saves and print information about the iterations.

```
for iter = 1:MAXITERS
   global fm, xm, s, mxeval, pos
   isready(chnl) && break                # break at a keystroke

   opt.maxeval = ceil(Int,10mxeval)      # stop at 10 times more function calls than average

   xinit = init(N,iter,seed)             # init() = feasible initial population

   minf,minx,ret = optimize(opt,xinit) # ==== OPTIMIZATION ====

   OK = verify(minx)

   if (!OK) D[:Infeasible] = 1 + get(D,:Infeasible,0) end
   D[ret] = 1 + get(D,ret,0)             # count values returned in "ret"

   if OK  &&  ret != :FORCED_STOP  &&  ret != :FAILURE  && ret != :MAXEVAL_REACHED
      mxeval = 0.97mxeval + 0.03opt.numevals
   end

   s = @sprintf("Iter#%6i,  time =%9.2f", iter, time()-tm);
   print("\e[2K\e[G$s")                 # info -> console - repeatedly overwrite current line
   seek(fl,pos);  write(fl,"# $s")      # for RESUME: overwrite last iter info in fl

   if  OK && minf < fm - 0.01*eps
      fm = minf;  xm[:] = minx
      @printf(   ",  f =% 1.12f\n", fm)
      @printf(fl,",  f =% 1.12f\n", fm)
      flush(fl);  pos = position(fl)   # flush optimum, move seek position (saved at abort)
   end
end
```

In the end the found best solution is printed and saved in the output file:

```
println(  "    ---- stopped ----");  write(fl,"#   ---- stopped ----\n")

for key in keys(D)  local s = "$(rpad(key,17))$(D[key])\n"; print(s); write(fl,"# $s")  end

t = @sprintf(" COS(angle) =% 1.14f @ (x,y,z) =\n",fm)
print(t); write(fl,"# $t")

s = @sprintf("% 1.9f   % 1.9f   % 1.9f\n", 0,0,1); print(s); write(fl,s)
s = @sprintf("% 1.9f   % 1.9f   % 1.9f\n", sqrt(1-xm[1]^2),0,xm[1]); print(s); write(fl,s)

Q = round.(hcat(XYZR(xm)...),digits=9).+.0               # put rounded X,Y,Z into Q, no -0.00
Q = sortslices(Q,dims=1,lt=(x,y)->x[[3,1,2]]>y[[3,1,2]]) # sort rows by Z,X,Y descending
for i = 1:N-2
   local s = @sprintf("% 1.9f   % 1.9f   % 1.9f\n", Q[i,1],Q[i,2],Q[i,3])
   print(s); write(fl,s)
end

close(fl)
```

8. Results

In the range N = 61…100 there are so many local optima of the numerical Tammes problem that our confidence in the global optimality of the found solutions is less and less as N growths. Using a home computer performing much more random restarts is impractical. With access to more computational power, the number of random restarts could be increased, and maybe better local optima could be found, which may turn out to be the global optima, the true solutions of the Tammes problem.

Below the results are summarized in a table. Note that there were test-runs for all problems, 5 with 150,000 iterations (N = 68, 73, 77, 81, 82), the others with 50,000 iterations, but with different generators for the initial populations. The above listed version of the program found *better* or the same local optima. The actual, total number of random restarts we have performed and listed in the second column of the table could be increased by 50,000, and in the aforementioned 5 cases, by 150,000.

The best solutions for some N values were found too close to the number of restarts. Shaded entries in the table indicate where 10% reduction of the number of restarts would miss the best solution, showing that more restarts could be necessary for finding the global optima.

The lists of the coordinates of the found best spherical point sets were too long to be included here, but anyone can recreate them with the program detailed in Section 7, above. Only the line

```
for iter = 1:MAXITERS
```

has to be replaced with the line, containing the entry “Opt. Iter#” in the table below. E.g., if the column contained “1234”, the replacement line would be:

```
for iter = 1234:1234
```

When started, the program would ask the number of points, N, then in few seconds the coordinates of the points of the numerical solution of the corresponding Tammes problem are printed on the console, and saved in the output file. (A subdirectory named MAD in the current directory has to be present for the output files. The directory name is the shorthand of MArsaglia w. Dynamic constraints.)

N	Restarts	Opt. Iter#	Optimum
61	60,000	1,648	0.89200844328203
62	60,000	4,392	0.89349685056883
63	70,000	9,974	0.89503618081411
64	70,000	5,349	0.89698816753708
65	70,000	33,199	0.89825910858396
66	70,000	5,665	0.89919577748308
67	80,000	23,502	0.90119822775634
68	80,000	24,435	0.90285692152157
69	80,000	39,249	0.90383460922318
70	80,000	30,825	0.90504303680279
71	80,000	5,768	0.90639673677562
72	90,000	48,542	0.90684928543572
73	90,000	67,941	0.90957174381548
74	90,000	80,847	0.91053262350104
75	100,000	12,019	0.91139090465160
76	100,000	74,505	0.91268728948391
77	100,000	91,891	0.91353634468780
78	100,000	46,319	0.91403444541297
79	110,000	85,644	0.91619796372558
80	110,000	106,955	0.91669066788385
81	110,000	881	0.91812050592972
82	120,000	99,829	0.91921610673818
83	120,000	14,144	0.91993788202315
84	130,000	25,084	0.92015169888905
85	130,000	77,276	0.92200728603694
86	130,000	123,548	0.92271617029789
87	140,000	97,812	0.92356759726451
88	140,000	55,055	0.92419417978360
89	150,000	101,789	0.92513747511725
90	150,000	148,292	0.92623115515484
91	160,000	2,417	0.92684609213400
92	160,000	82,952	0.92700341624231
93	160,000	21,881	0.92853006727172
94	170,000	99,547	0.92900246577367
95	180,000	104,390	0.92989681194411
96	180,000	135,953	0.93046202053222
97	190,000	36,320	0.93105553713269
98	190,000	133,987	0.93123999104557
99	200,000	4,708	0.93276201266923
100	200,000	8,416	0.93341118057491

8.1. Running Time

The slowest of the numerical optimizations for the Tammes problem was, of course, at N = 100. In a single thread on a 3.4 GHz six-core CPU of a home computer it took about 19 days of continuous work. Distributing the work to all 6 cores still required over 3 days. These present practical limits for this kind of inexpensive, simple computational platforms. On a computer network or a mainframe computer 2 or 3 orders of magnitude less time would be needed for similar computations.

9. The Thomson-P Problem

The objective of the Thomson problem (Thomson-1) is to determine the minimum electrostatic potential energy configuration of N electrons constrained to the surface of a unit sphere. The electrons repel each other with a force given by Coulomb's law. The physicist J. J. Thomson posed the problem in 1904 [16].

Mathematically, the objective function of the Thomson problem is to minimize the sum of the reciprocals of the distances between pairs of points.

A generalization of the Thomson problem, the Thomson-P problem, is minimizing the sum of the *P*-th powers of the pairwise distances. For *P* negative even numbers are interesting. If *-P* is large, the sum is dominated by the distances of the closest point-pairs.

9.1. Numerical Solutions of the Thomson-P problem

A fast solver for the Thomson-P problem is discussed in [15]. The spherical points are mapped to the (2-dimensional) plane with stereographic projection, encoding the Thomson-P problems as a nonlinear minimization problem, with no constraints. Many numerical algorithms have been published for these types of nonlinear, dense, unconstrained optimization problems, which efficiently find local minima.

For larger N values the numerical optimizations have to be restarted many times, such that the global optimum is likely found among the many local optima. E.g., at N = 200 for the Thomson-1 problem, hundreds of slightly different numerical optima were found, most of them within 0.001% of the global optimum, the true solution. With larger *-P* values the solvers get slower and slower, and they fail to converge increasingly more often (due to their internal matrices becoming ill-conditioned).

9.2. Thomson-P Sets as Initial Population for the Tammes Problem

An intriguing idea is that the numerical solution of the Thomson-P problem could serve as a *starting set of spherical points, for the numerical Tammes problem*. Unfortunately, it did not fully work out. The exact solutions of the Tammes problems were not found, only *slightly inferior solutions were reached, but faster than with other methods*.

At high powers P the minimum distance dominates the sum, thus the minimum energy state is close to arrangements with the possible largest minimal distances, which is the solution of the Tammes problem.

There are published lists of the best known arrangements of Thomson-1 points, which we could use for the starting populations of the numerical Tammes problems. They each give one initial point population for candidate solutions of the Tammes problem, but most likely they will not lead to the global optimum by numerical optimizations.

At the Thomson-1 problem, the total energy is significantly affected by larger distances, thus higher -*P* values are desirable, but we could not find public lists of optimal Thomson-P point sets for larger -*P* values. We had to perform the optimizations ourselves [15]. It turned out that for large N values there are a many local optima, which we can use for initial Tammes populations:

- Start with a random population of spherical points (with Mitchell's homogenization)
- Find a local numerical optimum of the Thomson problem with exponent *P* (which is empirically determined), using the random population as the initial value
- Discard the solution if the same objective function value has already been processed (within a relative accuracy of 10^{-13})
- Perform a numerical Tammes optimization for each Thomson-type initial population

9.3. Results of Experiments

The -*P* power cannot be chosen to be very large, because the matrices used by the numerical optimizations become ill-conditioned. Up to around N = 35, *P* = −16 seems to work, but for larger N values *P* = −12 was the largest, which did not cause most of the numerical optimizations to fail.

The numerical optimizations for the Thomson-12 problem did return different point sets when the optimization started from different random initial populations, but they were quite similar. E.g., at N = 55, all the power-energy values were within ±0.3%, and all the subsequent Tammes optimizations led to the same point set, which was close to the global optimum, but it did not actually reach it.

Below is the output of the program, which is the combination of two programs: the one described in [15], and the one discussed in this document, in Section 7. The program still has merits, because it found approximate solutions quickly (within 0.016% error of the cosine of the minimax angular distances).

```
N = 55
Mitchell = 5
Iter#     1,  time =     0.58,  h = 646313.13021382 -> f = 0.880927373689 *
Iter#     3,  time =     9.38,  h = 645231.88925865 -> f = 0.880927373689
Iter#     4,  time =    12.54,  h = 647462.79810613 -> f = 0.880927373689
Iter#    13,  time =    23.81,  h = 644077.34757877 -> f = 0.880927373689
Iter#   181,  time =   198.49,  h = 647173.35990667 -> f = 0.880927373689
Iter#   408,  time =   408.48,  h = 648529.76126256 -> --- FAIL ---
Iter#   743,  time =   710.76,  h = 646867.00260243 -> f = 0.880927373689
Iter#  1191,  time =  1150.84,  h = 645257.51609919 -> f = 0.880927373689
Iter#  3102,  time =  2992.15    ---- stopped ----

Thomson-XTOL_REACHED     13
Thomson-FAILURE          1507
Thomson-FTOL_REACHED     1578
Thomson-MAXEVAL_REACHED  3

Tammes- Infeasible       1
Tammes- FTOL_REACHED     4
Tammes- ROUNDOFF_LIMITED 3
Tammes- MAXEVAL_REACHED  1
```

The asterisk '*' indicates the best solution, which was already found in the first iteration, in 3 seconds. The 'h' value is the power-energy (the objective function value) of the numerical Thomson-12 problem.

There were 8 different such values found in 3100 random restarts, but only half of the numerical optimizations succeeded.

The Tammes optimization turned these slightly different initial populations into the same, near optimal spherical point set. The true optimum was found in [14]: 0.880785046522.

Note that the removing Mitchell's homogenizing algorithm from generating the initial point set (with setting M = 1) made the solutions of the Thomson-12 problem slightly more diverse, and the Tammes optimization found 2 different local optima. The better one was found in iteration 19 (after 19 restarts), still under 20 seconds. (Many more numerical optimizations fail, which makes the progress to look faster.)

```
N = 55
Mitchell = 1
Iter#     6,  time =     3.01,  h = 644077.34757877 -> f = 0.890290022548 *
Iter#    19,  time =    17.91,  h = 646313.13021382 -> f = 0.880857032376 *
Iter#    80,  time =    47.20,  h = 647462.79810613 -> f = 0.880857032376
Iter#    91,  time =    55.50,  h = 645231.88925865 -> f = 0.880857032376
Iter#   311,  time =   192.66,  h = 645257.51609919 -> f = 0.880857032376
Iter#  1181,  time =   739.73,  h = 647173.35990667 -> f = 0.880857032376
Iter#  2708,  time =  1642.57,  h = 645787.08333106 -> f = 0.880857032376
Iter#  3123,  time =  1916.21    ---- stopped ----

Thomson-XTOL_REACHED     2
Thomson-FAILURE          2749
Thomson-FTOL_REACHED     369

Thomson-MAXEVAL_REACHED  2
Tammes- Infeasible       0
Tammes- ROUNDOFF_LIMITED 4
```

As the output shows, without Mitchell's homogenization only 12% of the restarted numerical optimizations converge, but the local optima of the Tammes problem was slightly better among the 371 convergent cases.

With Mitchell number M = 10, most of the numerical optimizations of the Thomson-12 problem converged (73%), and so the procedure appears slower. Still, there are only 3 different Tammes sets found, all suboptimal:

```
N = 55
Mitchell = 10
Iter#     1,  time =     0.61,  h = 646313.13021381 -> f = 0.890660770162 *
Iter#     2,  time =     6.88,  h = 645231.88925865 -> f = 0.887963044669 *
Iter#    12,  time =    20.08,  h = 647462.79810613 -> f = 0.887963044669
Iter#    15,  time =    24.47,  h = 644077.34757877 -> f = 0.887963044669
Iter#   392,  time =   484.16,  h = 645257.51609919 -> f = 0.880852610230 *
Iter#   408,  time =   507.66,  h = 644369.93876821 -> f = 0.880852610230
Iter#   794,  time =   973.83,  h = 647173.35990667 -> f = 0.880852610230
Iter#   803,  time =   987.12,  h = 645787.08333107 -> f = 0.880852610230
Iter#  2808,  time =  3483.62,  h = 648529.76126256 -> --- FAIL ---
Iter#  3101,  time =  3842.71    ---- stopped ----

Thomson-XTOL_REACHED     19
Thomson-FAILURE          834
Thomson-FTOL_REACHED     2246
Thomson-MAXEVAL_REACHED  1

Tammes- Infeasible       1
Tammes- FTOL_REACHED     6
Tammes- ROUNDOFF_LIMITED 2
Tammes- MAXEVAL_REACHED  1
```

With Mitchell number M = 25 even more numerical optimizations of the Thomson-12 problem converged (87%), and so the procedure appears still slower. There are 4 different Tammes sets found, all suboptimal:

```
N = 55
Mitchell = 25
Iter#     1,  time =     0.59,  h = 646313.13021382 -> f = 0.890660770110 *
Iter#     2,  time =     6.48,  h = 644077.34757877 -> f = 0.890290022542 *
Iter#     6,  time =    12.30,  h = 645231.88925865 -> f = 0.887963045074 *
Iter#     9,  time =    17.52,  h = 647462.79810613 -> f = 0.880882905478 *
Iter#   306,  time =   389.65,  h = 645787.08333107 -> f = 0.880882905478
Iter#   356,  time =   453.93,  h = 645231.88926818 -> f = 0.880882905478
Iter#   794,  time =  1033.53,  h = 647173.35990667 -> f = 0.880882905478
Iter#  2414,  time =  3177.07,  h = 645231.88981728 -> f = 0.880882905478
Iter#  2796,  time =  3671.95,  h = 645231.88990047 -> f = 0.880882905478
Iter#  3105,  time =  4098.95,  h = 646867.00260244 -> f = 0.880882905478
Iter#  3115,  time =  4115.54    ---- stopped ----

Thomson-XTOL_REACHED     20
Thomson-FAILURE          421
Thomson-FTOL_REACHED     2672
Thomson-MAXEVAL_REACHED  1

Tammes- Infeasible       0
Tammes- FTOL_REACHED     5
Tammes- ROUNDOFF_LIMITED 5
```

Interestingly, the best local optimum for the Tammes problem, which was found with the Thomson-12 initialization (h = 644077.34757877), was the same with every Mitchell number we have tried.

9.4. Conclusion

The solutions of the Thomson-P problems as starting points for the numerical Tammes problem, lead quickly to quite good Tammes sets, but not to the true solutions of the Tammes problems.

10. References

[1] Tammes, R.M.L.: On the origin number and arrangement of the places of exits on the surface of pollen grains. Rec. Trv. Bot. Neerl. 27, 1–84 (1930)

[2] https://mathworld.wolfram.com/SpherePointPicking.html

[3] Marsaglia, G. "Choosing a Point from the Surface of a Sphere." *Ann. Math. Stat.* **43**, 645-646, 1972.

[4] Muller, M. E. "A Note on a Method for Generating Points Uniformly on *N*-Dimensional Spheres." *Comm. Assoc. Comput. Mach.* **2**, 19-20, Apr. 1959.

[5] Salamin, G. "Re: Random Points on a Sphere." math-fun@cs.arizona.edu posting, May 5, 1997.

[6] Bridson, R. "Fast Poisson Disk Sampling in Arbitrary Dimensions." https://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph07-poissondisk.pdf

[7] Hardware Bit-Mixers. Cryptology ePrint Archive: Report 2017/084: https://eprint.iacr.org/2017/084

[8] L. Hars, G. Petruska, "Pseudorandom Recursions - Small and Fast Pseudorandom Number Generators for Embedded Applications". EURASIP Journal on Embedded Systems, vol. 2007, Article ID 98417, 13 pages, 2007. doi:10.1155/2007/98417

[9] L. Hars, G. Petruska: "Pseudorandom Recursions II". EURASIP Journal on Embedded Systems 2012, 2012:1 doi:10.1186/1687-3963-2012-1

[10] The Julia Language: https://julialang.org/

[11] Steven G. Johnson, The NLopt nonlinear-optimization package, http://github.com/stevengj/nlopt

[12] The NLopt module for Julia, https://github.com/JuliaOpt/NLopt.jl

[13] Thomas Williams & Colin Kelley, "gnuplot 5.5, An Interactive Plotting Program". http://sourceforge.net/projects/gnuplot

[14] L. Hars. "Numerical Solutions of the Tammes Problem for up to 60 Points". http://www.hars.us/Papers/Numerical_Tammes.pdf.

[15] L. Hars. "Numerical Solutions of the Thomson-P Problems". http://www.hars.us/Papers/Numerical_Thomson.pdf.

[16] J. J. Thomson. "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. 7 (39): 237–265 (1904).