Numerical Solutions for the Thomson Problems

By Laszlo Hars

Draft 0.4: December 6, 2020 – January 20, 2022

Contents

1. Introduction

This document discusses numerical (approximate) solutions of the Thomson-P family of problems, related to our work on the Tammes problems: [14] and [15].

The latest version of this document is at: http://www.hars.us/Papers/Numerical_Thomson.pdf.

This document contains the description and listing of two computer programs

- for the numerical solutions for the Thomson-P problems
- handling hundreds of points
- written for a cheap home computer (~$500 in 2021).

These two programs find good, often optimal spherical point sets with respect to the Thomson objective function, in reasonable time. For the Thomson-1 problem of N ≤ 322 points the presented programs found all the best published point sets.

Only open source, free tools are used.

The program performs numerical optimizations starting from many pseudorandom sets of spherical points. The best local optimum in sufficiently many experiments is the true solution, at high probability.

Because a large number of independent numerical optimizations are performed, the programs are highly parallelizable. On modern microprocessors multiple threads can be started simultaneously, each running essentially as fast as a single thread alone. For large Thomson sets networks of computers could run many more autonomous threads.

Listing the coordinates of the points in the found good Thomson sets would be too long for this document. Instead, for each number of points the seed of the pseudorandom number generator, the iteration (restart) number of the best approximate solution is given in the tables in the end of this document. Next to the restart number the corresponding value of the objective function is tabulated. Giving a restart number as input to the described programs, they generate and print out the coordinates of the known best Thomson point sets, in mere seconds on a home computer. These computations slow down with larger number of points, but they take less than a minute for up to 322 points.

We did not perform numerical optimizations for N > 322 points, due to the lack of reliable data to compare our results to.

The programs described in this document are placed in the public domain, anyone can use them for any purpose, but without our claims for correctness or function.

2. The Thomson-P Problem

The objective of the original Thomson problem (Thomson-1 problem) 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 [1], after proposing an atomic model, later called the plum pudding model, based on his knowledge of the existence of negatively charged electrons within neutrally-charged atoms.

Mathematically, the objective of the Thomson problem is to minimize the sum of reciprocals of the distances between pairs of *N* points, that is to minimize the function:

$$U(N) = \sum_{1 \le i < j \le N} r_{ij}^{-1}$$

where r_{ij} is the spatial (3D) distance between the points P_i and P_j, which lie on the unit sphere:

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

$$\text{with } x_i^2 + y_i^2 + z_i^2 = 1 \text{ for all } 1 \le i \le N.$$

A generalization of the Thomson problem, the Thomson-P problem, is minimizing the sum of the *P*-th powers of the pairwise reciprocal distances.

$$U_P(N) = \sum_{1 \le i < j \le N} r_{ij}^{-P}$$

For *P*, large even numbers are interesting, because then the computations are faster (no square root is needed) and at large *P* values, the sum is dominated by the distances of the closest point-pairs. Because of this, the solutions of the Thomson-P problems are homogenous arrangements of points, similar to the solutions of the Tammes problems [14]. Unfortunately, *P* > 16 values lead to ill-conditioned matrices in most public domain numerical optimization algorithms, and they would fail to converge to a local minimum. The few algorithms, which do work, get very slow.

The solutions at *P* = 1 represent the minimum electrostatic potential energy configuration of N electrons constrained to the surface of a unit sphere, the original "Thomson-1" problem.

3. Numerical Solutions for the Thomson problems

3.1. Computational Tools: Software and Hardware

Hardware: a home-built cheap PC was used, with:

- CPU: AMD Ryzen 5 2600 (64-bit six-core Intel compatible processor)
- Clock: 3,400 MHz
- Memory: 16 GB DRAM

Software:

- Operating system: Windows 10
- Programing language: Julia 1.5 [10], and Visual C [19]
- Nonlinear optimization library: NLopt versions 2.6.1 and 2.7.1 [11]
- Julia interface to the NLopt library: NLopt Julia module [12]
- Interactive plotting program: gnuplot 5.5 [13] (only for graphical plots)

Resources used for a single optimization thread (N = 250):

- CPU load: 9.25%
- Memory: < 275 MB (for N = 256)

Multiple optimization threads can run in parallel. The used computer could perform 6 independent numerical optimization processes, each at essentially the same speed as a single active process (within 15%), even together with some light everyday tasks, like email, document editing, etc. Because of the supported hyperthreading, the CPU can handle more threads, but with proportional slowdown

compared to 6 optimization threads. This slowdown is caused by that the CPU cores are utilized nearly 100% with 6 threads, and hyperthreading a further thread suspends the previously executed thread. (Hyperthreading would be useful for I/O intensive tasks, but the numerical optimization is CPU intensive.)

3.2. Optimization Technique

The basic structure of the optimization is the following: A pseudorandom number generator is used to generate many random initial sets of N points on the sphere. From each of these sets, a numerical optimization is started, which either converges to a local optimum, or does not find a solution within reasonable time limits. This process is then restarted many times, keeping the results of the convergent optimizations. During the iterations the best local optimum is kept, printed on the console and saved in a text file. After sufficiently many iterations the found best solution is likely the global optimum.

3.3. Numerical Optimizations

One way to formulate the Thomson problems as numerical optimization problems is to map the spherical points to the (2 dimensional) plane, e.g., with cylindrical or stereographic projection, or by using polar coordinates (see in [14]). These mappings directly translate the Thomson problems to unconstrained nonlinear minimization problems. Many numerical algorithms have been published for these types of nonlinear, dense, unconstrained optimization problems, which efficiently find *local* minima, when started from a set of initial points.

Nevertheless, numerical optimizations find only local optima. The numerical optimizations are performed many times in a random search minimization strategy, such that the global optimum is likely among the found local minima. Experiments showed that the Thomson problem for larger N (e.g., $N > 200$) has a huge number of slightly different numerical optima, many of them within 0.001% of the global optima, the true solutions. Accordingly, our local optimization procedure must also be restarted from very many (pseudo)random initial conditions for a reasonable chance to hit the global optimum.

3.4. Numerical Optimization Algorithms

We tried all the suitable algorithms from the NLopt library.

With ***derivative free*** algorithms many experiments were conducted with various small N values, e.g., 33:

LN_PRAXIS	global optimum found at 1st iteration (20 sec)
LN_BOBYQA	global optimum NOT reached
LN_COBYLA	SLOW
LN_NEWUOA_BOUND	SLOW
LN_NELDERMEAD	SLOW
LN_SBPLX	SLOW

Similar experiments with local-search, ***gradient using*** algorithms, with I, the initialization seed of the pseudorandom number generator:

Algorithm	N = 5, I = 100	N = 19, I = 200
LD_MMA	OK	OK very slow
LD_CCSAQ	OK	OK very slow
LD_SLSQP	100% forced stop	100% forced stop
LD_LBFGS	OK	OK very fast
LD_TNEWTON_PRECOND_RESTART	OK	OK
LD_TNEWTON_PRECOND	OK	OK
LD_TNEWTON_RESTART	OK	OK slow
LD_TNEWTON	OK	OK very slow
LD_VAR1	OK	OK fast
LD_VAR2	OK	OK fast

The clear winner was the LD_LBFGS algorithm. It was the fastest, and it almost always converged. By using it in the Julia program as described later in this document, the solutions for Thomson problem can be found in mere seconds or minutes, up to N = 150, and even good solutions for the case N = 470 can be found in a few days using a single thread on our cheap home computer. (Larger N values not only made each numerical optimization slower, but due to the presence of many local optima, the necessary number of random restarts grew huge – for reasonable confidence that the found best local optimum is actually the global optimum, the solution of the Thomson problem.)

3.5. Derivatives of the Objective Funtion

For numerical optimizations, the derivatives of the objective function are useful. Since r_{ij} is symmetric to all coordinates appearing, a partial derivative can be translated to all others just by changing the indices of the points involved.

$$\frac{\partial r_{ij}^{-P}}{\partial x_i} = \frac{\partial \left(\left(x_i - x_j\right)^2 + \left(y_i - y_j\right)^2 + \left(z_i - z_j\right)^2 \right)^{-P/2}}{\partial x} =$$

$$= -P \cdot \left(x_i - x_j\right) \left(\left(x_i - x_j\right)^2 + \left(y_i - y_j\right)^2 + \left(z_i - z_j\right)^2 \right)^{-P/2-1}$$

The important special case of $P = 1$ is:

$$\frac{\partial r_{ij}^{-1}}{\partial x_i} = \frac{x_j - x_i}{\sqrt{\left(x_i - x_j\right)^2 + \left(y_i - y_j\right)^2 + \left(z_i - z_j\right)^2}^{\frac{3}{2}}} = \frac{x_j - x_i}{\left(\sqrt{2} \cdot \sqrt{1 - x_i x_j - y_i y_j - z_i z_j}\right)^{\frac{3}{2}}}$$

3.6. Encoding of the Spherical Points

In our experiments, encoding the Thomson problems by mapping spherical points to the plane with *stereographic projection*, was found the most stable (ensured convergence the most often). This projection is popular, because arcs of great circles map to straight lines, and it is very fast, because only the 4 basic arithmetic operations are used in both the transform and in its inverse.

The stereographic projection of spherical points (excluding the North Pole) is defined as

$$(x, y, z) \to (u, v):$$

$$u = \frac{x}{1-z};\ v = \frac{y}{1-z}$$

where $x^2 + y^2 + z^2 = 1$ and $z \neq 1$.

Elementary calculations show that the inverse transform of the stereographic projection is

$$x = \frac{2u}{u^2+v^2+1};\ \ y = \frac{2v}{u^2+v^2+1};\ \ z = 1 - \frac{2}{u^2+v^2+1}$$

One of the spherical points of the Thomson set, P_{-1}, can be fixed, e.g., to the forbidden location (0,0,1). In an optimal set no points is close to the others, that is, there will be no point close to this fixed point. It makes the encoding continuously differentiable around local optima.

One of the coordinates of another point can also be fixed, e.g., $y_0 = 0$, eliminating another dimension from the optimization problem: $P_0 = \left(\sqrt{1-z_0^2},\ 0,\ z_0\right)$. We put z_0 in the beginning of the vector of the optimization variables, in **`x[1]`** (in Julia, where vector indices start at 1) or **`x[0]`** (in C, where indexing starts at 0). However, with such fixed values, the distance-powers are not differentiable, at the South Pole – therefore appropriate measures are needed for some of the available optimization algorithms to avoid divisions by 0. A simple solution is to forbid Z_0 to be below −0.5, which is the smallest valid value at optimal arrangements for N ≥ 3. However, this bound was not necessary with the NLopt optimization algorithms: an optimization very rarely fails to converge among the many randomly restarted ones.

3.6.1. Partial Derivatives of the Inverse Stereographic Projection

For gradient-using optimization methods the partial derivatives of the inverse stereographic projection are needed. They are:

$$\frac{\partial x}{\partial u} = -2\frac{u^2 - v^2 - 1}{(u^2 + v^2 + 1)^2}$$
$$\frac{\partial x}{\partial v} = \frac{\partial y}{\partial u} = -\frac{4uv}{(u^2 + v^2 + 1)^2}$$
$$\frac{\partial y}{\partial v} = 2\frac{u^2 - v^2 + 1}{(u^2 + v^2 + 1)^2}$$
$$\frac{\partial z}{\partial u} = \frac{4u}{(u^2 + v^2 + 1)^2}$$
$$\frac{\partial z}{\partial v} = \frac{4v}{(u^2 + v^2 + 1)^2}$$

3.7. Homogenous Intial Point Set: Mitchell's Algorithm

Putting N uniform random points on the sphere is straightforward (wee e.g.,[20]):

$$x = \sqrt{1 - u^2}\cos\theta$$
$$y = \sqrt{1 - u^2}\sin\theta$$
$$z = u$$
with $\theta \in [0,2\pi)$ and $u \in [-1,1)$.

However, in such uniform populations of points, there are often clusters (points close to each other) and large empty areas. Therefore, the numerical optimizations could be better started from not uniform random, but more homogenous initial population. This looks promising, because no optimal point arrangements have clusters.

Homogenous point sets can be generated by Poisson-disc sampling, where no 2 points are less than a minimum distance apart, but efficient implementations are quite complex when large disks should also be handled. Mitchell's best-candidate algorithm [6] is a straightforward approximation of the Poison-disc distribution: Having produced some points of the initial set S for the current restart of the numerical optimizations, we generate a number of new uniform random candidate points. That candidate point is added to S, which is the farthest from all already chosen points (S). More precisely:

- Start with an arbitrary single point in S
- In each subsequent step, while |S| < N, add one more point to S, as follows
    - Generate M uniform random candidate points on the sphere, forming the set C
    - Find the shortest distance from the already placed points in S and each of the candidate points in C
    - The candidate point, which has the largest of these shortest distances is added to S, discarding the other M−1 candidates in C.

Setting the best M value is a delicate process, requiring many experiments. Too large numbers of Mitchell iterations make the initial population too regular, too homogenous. Such initial sets do not provide large variations, and so finding the global optima needs many restarts. Too small M values make inhomogeneous initial populations, where many optimizations diverge, or lead to inferior local optima.

To compare different settings, the number of convergent and divergent numerical optimizations in many experiments were counted, and checked, how many restarts were needed to find the known best Thomson sets. Evaluating the number of necessary restarts is not straightforward, because a lucky initial population can occasionally lead to a good Thomson set early. Performing the numerical optimizations is necessary with thousands of different randomness seeds. This procedure does exhibit a discernable trend.

The performed experiments showed the optimal Mitchell number M around 3 for small n, the number of points in the Thomson problem.

3.7.1. *Effects of the Mitchell Number*

The table below contains the number of restarts necessary to find the optimum for 6 midsize Thomson problems, with Mitchell number = 1…6.

N	Optimum	M = 1	M = 2	M = 3	M = 4	M = 5	M = 6
197	17878.3401625712	9342	4742	1604	4476	17408..	944
201	18627.5912262442	6608	138	99	68	1997	973
210	20370.2516151291	86	709	1331	558	468	241
258	30994.2135774865	2397	1493	611	996	588	1216
259	31239.4423068662	3679	292	1123	4213	4177	7263
260	31485.5781917796	56	943	1760	958	1202	6034..
sum		22168	8317	6528	11269	25840..	16671..

Table 1. Effects of the Mitchell Number

(A number followed by ".." indicates that the optimum was not reached even until this many restarts.)

There is a clear winner: M = 3 was noticeable better than the runner-up, M = 2. The experiments were performed with the multithreaded C implementation of random-restart numerical optimizations, but the result was consistent with hundreds of unorganized trials done with tweaking the Julia code. Unless otherwise noted, M = 3 was chosen for our subsequent experiments.

3.7.2. Varying Mitchell Number

We saw that the Mitchell number has a significant impact on the number of necessary restarts. It was a natural idea to vary this number (e.g., repeatedly going through M = 2, 3, 4 and 5) during the iterations of the Mitchell homogenization. Unfortunately, the idea did not improve the necessary number of iterations, showing that having only every 4^{th} initial point-configuration for a given Mitchell number can miss good initial sets.

4. Julia Program for the Random Restart Numerical Optimizations

The intention was to build a prototype, proof of concept program in Julia, and subsequently rewrite it in C. The (Windows) C program presented in the end of the document does not suffer from the relatively frequent breaking changes of the Julia specifications, has wider support and it allowed easier multithreading. Manual optimizations are also more straightforward, not requiring to dive into the intricacies of Julia. On the other hand, a program in a lower-level programming language (C) is longer, and may take longer to develop.

4.1. Building blocks

The Julia program starts with setting parameters, like N, the number of restarts (MAXITERS), the size of the Mitchell's candidate set (3), the expected number of function evaluation in a numerical optimization, the relative tolerance (eps = 10^{-15}), and the Thomson-P power (PW = 1 is pre-selected).

```
using NLopt
using Printf

PW = 1; pw2 = .5PW                      # power used in Ep (Thomson-PW)
if     PW ==16 @inline PW2(x::Float64) = ((x^2)^2)^2
elseif PW ==12 @inline PW2(x::Float64) = (x^3)^2
elseif PW == 8 @inline PW2(x::Float64) = (x^2)^2
elseif PW == 6 @inline PW2(x::Float64) = x^3
elseif PW == 4 @inline PW2(x::Float64) = x^2
elseif PW == 2 @inline PW2(x::Float64) = x
elseif PW == 1 @inline PW2(x::Float64) = sqrt(x)
else println("PW must be in [1,2,4,6,8,12,16]"); exit(-1)
end
pwc = PW2(.5)                           # correction for left out 2's

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

if length(ARGS) <= 1
   ITER1 = 1; ITERX = round(Int,2^(N/37))*15PW;
elseif length(ARGS) == 2
   ITER1 = ITERX = parse(Int,ARGS[2]);
else
   ITER1 = parse(Int,ARGS[2]);
   ITERX = parse(Int,ARGS[3]);
end

MITCHELL = 3PW                          # increasing with PW
mxeval = 25N                            # starting value of estimated #evaluations of f

Alg = :LD_LBFGS                         # optimization algorithm
seed = 0                                # seed for bit-mixer RNG   (UInt64)
eps = 1e-15                             # tolerance in optimizations

filename = pwd()*"\\TSP\\Thomson$PW-$N-$MITCHELL$(ITER1==1 ? "" : -ITER1).txt"
```

Experiments showed that the number of evaluations of the objective function is about 5N per restarts. To be on the safe side, the initial estimate is set to `mxeval = 25N`. During restarts this value is updated by exponential averaging from the actual number of function evaluations of convergent minimizations. After a few dozen restarts, the average will settle to its correct value. When an optimization step takes more than 10 times as many internal updates (function evaluations), this optimization run would likely fail, therefore, the computation is stopped, and a new iteration is started.

Julia does not have built-in asynchronous keyboard input, so we set up a separate task, which puts key codes into a channel, at a key press. Other tasks can test if the channel has something in it, and if yes, perform a prescribed action, like stop.

```
# --- 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.")   # set raw terminal mode to catch keystrokes
const chnl = Channel()                       # unbuffered channel for key codes
@async while true put!(chnl,readavailable(stdin)) end
```

For convenience, we define vector generators as below. (They allocate memory, which can be slow. Using global arrays as done in the C version of the program would be faster.)

```
FVec(L::Integer) = similar(1.:L)             # generator for Float64 Vectors
FVec(L::Integer,K::Integer) = (similar(1.:L) for _=1:K)
```

For the stereographic encoding of the Thomson problem, and for computing its partial derivatives two short helper functions are defined:

```
function XYZQ(x::Vector)                     # Stereographic Projection
   global N; N::Integer
   X,Y,Z,Q = FVec(N-2,4)
   for i = 1:N-2
      Q[i] = q = 2.0/(x[2i]^2+x[2i+1]^2+1)   # Planar points u[i] ~ x[2i], v[i] ~ x[2i+1]
      X[i] = x[ 2i ] * q
      Y[i] = x[2i+1] * q                     # coordinates of the SPHERICAL POINTS ---
      Z[i] = 1 - q
   end
   return X,Y,Z,Q
end

function DXYZ(x::Vector, Q::Vector)          # PARTIAL DERIVATIVES of (u,v) -> (X,Y,Z)
transform
   global N; N::Integer
   Xu,Xv, Yv, Zu,Zv = FVec(N-2,5)
   for i = 1:N-2
      q = Q[i]^2 / 2
      Xu[i]= (-x[2i]^2 + x[2i+1]^2 + 1) * q  # dX/du
      Xv[i]= -2x[2i] * x[2i+1] * q           # dX/dv = dY/du
      Yv[i]= 2q - Xu[i]                      # dY/dv
      Zu[i]= 2x[ 2i ] * q                    # dZ/du
      Zv[i]= 2x[2i+1] * q                    # dZ/dv
   end
   return Xu,Xv, Xv,Yv, Zu,Zv                # Yu = Xv
end
```

The function `f()` below is the objective function. It also computes the gradients, with the chain rule of derivatives.

```
function f(x::Vector, grad::Vector)          # --- OBJECTIVE FUNCTION with GRADIENTS
   global N, pw2; N::Integer; pw2::Float64

   X,Y,Z,Q = XYZQ(x)                         # Spherical points
   Z0 = x[1]; X0 = sqrt(1-Z0^2)

   Xu,Xv, Yu,Yv, Zu,Zv = DXYZ(x,Q)           # partial derivatives of the 2D -> 3D mapping
   fill!(grad,.0)

   t = 1.0/(1.0-Z0);  s = PW2(t)             # dist(-1,0)
   grad[1] = pw2 * t*s
   for j = 1:N-2
      t = 1.0/(1-Z[j])
      s += t2   = PW2(t)                     # + sum of dist(-1,j)^pw2
      t = pw2 * t*t2
      grad[ 2j ] = t * Zu[j]                 # first assignments
      grad[2j+1] = t * Zv[j]

      t = 1.0/(1-X0*X[j]-Z0*Z[j])
      s += t2   = PW2(t)                     # + sum of dist(0,j)^pw2
      t = pw2 * t*t2
      grad[  1 ] += t * (-Z0/X0*X[j] + Z[j])
      grad[ 2j ] += t * (X0*Xu[j] + Z0*Zu[j])
      grad[2j+1] += t * (X0*Xv[j] + Z0*Zv[j])
   end

   for i = 1:N-3, j = i+1:N-2                # + sum of dist(i,j)^pw2
      t = 1.0/(1-X[i]*X[j]-Y[i]*Y[j]-Z[i]*Z[j])
      s += t2 = PW2(t)
      t = pw2 * t*t2
      grad[ 2i ] += t * (Xu[i]*X[j] + Yu[i]*Y[j] + Zu[i]*Z[j])
      grad[2i+1] += t * (Xv[i]*X[j] + Yv[i]*Y[j] + Zv[i]*Z[j])
      grad[ 2j ] += t * (Xu[j]*X[i] + Yu[j]*Y[i] + Zu[j]*Z[i])
      grad[2j+1] += t * (Xv[j]*X[i] + Yv[j]*Y[i] + Zv[j]*Z[i])
   end

   return s
end
```

The following functions generate pseudorandom, homogenous initial point populations, from which the numerical optimizations are started.

```
function h(x::Integer, y::Integer)       # RAX hash function for fixed size I/O
   r64(n::Unsigned,d::Integer) = (n<<d) | (n>>(64-d))
   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
   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
                                         # --- INITIAL STEREOGRAPHIC POPULATION
function init(N::Integer, iter::Integer, seed::Integer)   # set P[1..N-2]; implicit P[-1],P[0]
   X,Y,Z = zeros.(fill(N,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   # P2 is not too far/close to P1
   X[2],Y[2],Z[2] = cos(s), .0, sin(s)

   for i = 3:N                           # reconstructable pseudorandom population
      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
         v[:] = srnd(iter+i<<32,seed+k<<32)
         for j = 1:i-1                   # find largest cos(dist(Pk,Pj)) ~ closest
            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 of the largest minimal distance
         end
      end
      X[i],Y[i],Z[i] = vxx
   end

   x = zeros(2N-3)                       # optimization variables
   x[1] = Z[2]
   for i = 3:N
      x[2i-4] = X[i] / (1 - Z[i])
      x[2i-3] = Y[i] / (1 - Z[i])
   end
   return x
end
```

Next a Dictionary D is defined for the return values of the numerical optimizations and their number of occurrences, and a Set ST, to contain the different local optima found. A local optimum is characterized with the ratio of the objective function value and the first optimum, scaled up by 100 times the tolerance of the numerical search. It is rounded to an integer – to avoid problems with rounding errors and the binary representation issues of floating-point numbers.

```
D = Dict{Symbol,Integer}()              # empty dictionary for return values
ST = Set{Int}()                         # set of different solutions
```

Before the main loop of numerical optimizations, some housekeeping variables are initialized, the output file is opened and the parameters of the numerical optimization algorithms are set.

```
fl = open(filename,"w"); println("Output: $filename")
pos = 0;  f0 = Inf;  minkey = Inf;  mx = FVec(2N-3);  mf = Inf

tm = time();  s = "waiting for results"

opt = Opt(Alg, 2N-3)                    # OPTIMIZATION ALGORITHM (LD_LBFGS)
opt.min_objective = f
opt.ftol_rel = eps                      # relative tolerance for f
opt.lower_bounds = [-1.; fill(-Inf,2N-4)]
opt.upper_bounds = [ 1.; fill( Inf,2N-4)]
```

The main optimization loop calls the generation of the initial populations, performs the numerical optimizations, counts the occurrences of the return values, prints information about the current iterations on the console and in the output file, updates the average number of function evaluations and keeps track of the different local optima, with remembering the best local optimum.

The current state of the optimization process is printed to the console (iteration number, time, objective function value), and also saved in the output file. To reduce the printed data, the unimportant pieces of this information is constantly overwritten in the terminal and written (and flushed) to disk. This way, at an unexpected interrupt (e.g., at a power outage), the work can be resumed from the last completed iteration, but a large number of physical disk-write operations are performed (which is not good for solid state storage devices). In the C variant of the program, data is written to the disk only once in every 5 minutes.

```
for iter = ITER1:ITERX
   global mxeval, pos, f0, s, tm, minkey, mx, mf
   isready(chnl) break                  # break at a keystroke

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

   opt.maxeval = ceil(Int,10mxeval)     # stop at 10x more function evaluations than average
   minf,minx,ret = optimize(opt,xinit)  # ==== OPTIMIZATION ====
   D[ret] = 1 + get(D,ret,0)            # count occurrences of ret values

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

   if  ret != :FORCED_STOP  &&  ret != :FAILURE  &&  ret != :MAXEVAL_REACHED
      mxeval = 0.97mxeval + 0.03opt.numevals
      f0 == Inf && (f0 = minf)
      key = round(Int,minf/f0/100eps)   # fingerprint of a local optimum

      if key in ST continue end         # already handled Thomson solution
      c = ' '
      push!(ST,key)
      if  minkey > key
          minkey = key
          mx[:] = minx; mf = minf       # mx[:] = ... copies array
          c = '*'                       # printing * marks the best solution so far
      else c = ' '
      end
      s = @sprintf(",  f = %-15.14g %c\n", minf*pwc,c);  print(s)
      write(fl,s); flush(fl);  pos = position(fl) # saved for resume after interrupt
   end
end
```

In the third line of the main loop, we check if a keyboard key has been pressed. If yes, the loop terminates, with printing out the best point set found so far.

The above loop prints out all the different objective function values. The function values, which are smaller than the previous values, are marked with an asterisk, printed in the end of the corresponding lines. However, the list of different objective function values can be very long for large number of spherical points.

A simpler output structure can be used instead, if only the so-far best local optimum is of interest. The *variant* below only prints (and saves into the output file) the objective function values, which are less than all the previously found local optima.

```
for iter = ITER1:ITERX
   global mxeval, pos, f0, s, tm, mx, mf
   isready(chnl) break                    # break at a keystroke

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

   opt.maxeval = ceil(Int,10mxeval)       # stop at 10x more function evaluations than average
   minf,minx,ret = optimize(opt,xinit)    # ==== OPTIMIZATION ====
   D[ret] = 1 + get(D,ret,0)              # count occurrences of ret values

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

   if ret != :FORCED_STOP  &&  ret != :FAILURE  &&  ret != :MAXEVAL_REACHED
      mxeval = 0.97mxeval + 0.03opt.numevals
      if  minf < mf - 0.01*eps
         mf = minf;  mx[:] = minx
         s = @sprintf(",  f = %-15.14g\n", minf*pwc);  print(s)
         write(fl,s); flush(fl);  pos = position(fl)   # saved for resume after interrupt
      end
   end
end
```

The only remaining task after the loop terminates (or interrupted by hitting ENTER) is to sort and print the results to the console and save these 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(" OPT = %-15.14g @ (x,y,z) =\n",mf*pwc)
print("\n$t"); write(fl,"\n#$t")

t = @sprintf("% 1.15f % 1.15f % 1.15f\n", 0,0,1); print(t); write(fl,t)

x0 = sqrt(abs(1-mx[1]^2))
T = round.(vcat([x0 0 mx[1]],hcat(XYZQ(mx)[1:3]...)),digits=15).+.0   # rounded X,Y,Z; no -0.0
T = sortslices(T, dims=1, rev=true, by=x->x[[3,1,2]])          # sort rows by Z,X,Y descending

for i = 1:N-1
   local t = @sprintf("% 1.15f % 1.15f % 1.15f\n", T[i,1],T[i,2],T[i,3])
   print(t); write(fl,t)
end
close(fl)
```

4.2. Using the Numerical Optimization Program

The code segments listed above, in Section 4.1, (with one of the variants of the main loop) have to be copied into a file in the order they have been listed. The result is the Julia program, used to find good numerical solutions to the Thomson problem.

The program needs command line parameters (or it would ask the number of points, N, and work as if it was the only parameter given). It then starts a random-restart optimization loop, and prints running status information. The program stops after a preset or given number of iterations, and it prints the coordinates of the points in the found best Thomson set. (The output is also saved in a file.)

The parameters:

1. One parameter: N. In this case, the restart loop runs from iteration number 1 to the default limit
2. Two parameters: N, *Iter#* (a single iteration number). It is used to print out the coordinates of the points in the Thomson set derived from starting the numerical optimization from the initial point set corresponding to restart number Iter# (which is the seed of the pseudorandom number generator).
3. Three parameters: N, with the starting and ending iteration numbers. It is used to continue the optimization loop after interruption, or when the optimizations are distributed among threads.

Some parameters are hard coded. They have to be edited in the program code for changes:

- PW (1): the power used in the Thomson-P problems. PW = 1, 2, 4, 6, 8, 12 and 16 are supported. Larger values make the numerical optimization algorithm to diverge, most of the time (due to some internal matrices becoming ill conditioned).
- ITERX (`round(Int,2^(N/37))*10PW`): the default loop limit. It increases exponentially with N, and linearly with PW.
- MITCHELL (3PW): The number of candidate points in Mitchell's algorithm
- mxeval (25N): The starting value of the estimated number of evaluations of the objective function, f() in a numerical optimization step. mxeval is updated during the computations, so only the first few restarts of the optimization are affected, unless a too small value prevents normal operation.
- Alg (:LD_LBFGS): The name of the optimization algorithm in the NLopt library
- seed (0) The seed for bit-mixer RNG (UInt64). When the work is distributed among threads, each could use a different seed to generate a different sequence of initial point population.
- eps (10^{-15}): The relative tolerance of the changes of the objective function, during optimizations.

5. C Program for the Thomson-1 Problem

The above discussed Julia numerical optimization program was rewritten in C (for the most important P = 1 case), because C allows simpler control over the optimizations of operations and memory use. Various manual optimizations were performed, including re-building the NLopt library (NLopt.dll) with full compiler optimizations. This was done in CLion [17], a CMAKE based Integrated Development Environment (IDE) [16]. The source code of the NLopt numerical optimization library (of version 2.7.1) contains all the necessary CMAKE files, which allows building the library (NLopt.dll and NLopt.lib) directly, with the C compiler of the Visual Studio 2019 Community Edition (which is free for personal

use). Compiling the NLopt library directly in Visual Studio is not much harder, but requires several configuration steps.

The numerical optimization C program was built in the development environment Visual Studio 2019, Community Edition. The physical stack and heap size were specified each as 500 MB, to prevent using virtual memory. Since the used PC had 16 GB RAM, the memory reserved for the multithreaded program was insignificant. Note that Windows manages the heap and stack sizes, and despite the high requested values the actual memory use of each instance of the running C program was 11…15 MB for the problem sizes N = 200…300.

A running instance of the program, or a parallel thread of the multithreaded implementation takes about 9.5% of the CPU processing power. With 6 parallel threads the overall CPU utilization was 60…75% while also performing everyday tasks (like email and word processing).

Unfortunately, the multithreaded C program is not directly usable under a Linux-type operating system: it uses a handful of Windows specific functions from the Windows system libraries. There are equivalent Linux functions, therefore porting the program to Linux should not be difficult.

5.1. Running the C Program

The compiled executable (ThomsonX.exe) has to be copied to a directory (we used D:\Thomson), where text files of the running information (snapshots) of the optimization process will also be stored.

If the NLopt library (nlopt.dll) is not globally accessible (e.g., from the Windows System32 directory), it has to be copied into the directory D:\Thomson.

The run-time library of Visual Studio (in our case vcruntime140_1.dll) is also needed. If it is not globally accessible, it has to be copied to the directory D:\Thomson, too.

In this directory the Thomson optimization program can be started, from a command prompt, e.g., from the Windows terminal:

```
d:
cd \Thomson
for /l %n in (3, 1, 322) do ThomsonX.exe %n 3 6 50000
```

When the work finishes (in a couple of weeks), the directory D:\Thomson will contain hundreds of saved (final) snapshot text files, with the results of the numerical optimizations.

5.2. Multithreading

The C program code is listed in the Appendix, Section 9, at the end of this document. It is a typical multithreaded program, with a few shared global variables, like N, iter, Mitchell… A command line parameter sets the number of threads. Each of them performs a numerical optimization with an initial point set, dependent on the value of “iter”. After the current value of “iter” is taken by the thread, it increments “iter”, such that the thread, which becomes available next, would use the next iteration count.

The threads also compare the variable “iter” and the shared maximum “iterX”. If the maximum is reached, the terminates.

5.3. Running Multiple Programs vs. Multithreading

Because the CPU has 6 cores, the best is to set the number of threads = 6, as discussed below.

At N = 197 the CPU use was 56…57%, the memory use 75…82 MB.

The wall timer, when 6 independent single threaded program was started:

```
Number of points (> 2): 197
Iter#     0,  time =     0.85,  f = 17878.3827457723
Iter#    21,  time =    27.54,  f = 17878.3827457721
Iter#  2682,  time =  6424.19,  f = 17878.3417261790
```

The corresponding wall timer values of a single C program, running 6 parallel threads, should be around 6 times less. We measured:

```
Iter#     0,  time =     1.98, f =  17878.3827457723
Iter#    21,  time =     9.70, f =  17878.3827457721
Iter#  2682,  time =  1063.84, f =  17878.3417261790
```

Indeed, 6*1063.84 = 6383, which is very close to 6424.19 the time used by the single threaded program, when 6 copies of it had been started. We see an extra second for starting up the threads, which is actually less than the time needed for starting 6 copies of the single threaded C program, but this is not shown in the printed information: the time values were from the start of the optimization.

Of course, we can use more than 6 threads, in the hope that Windows distributes its other tasks more evenly, which may result in a slight speedup of the optimization process. The timing with 9 threads:

```
Iter#     6,  time =     2.44, f =  17878.6910394117
Iter#     0,  time =     3.00, f =  17878.3827457723
Iter#    21,  time =    10.76, f =  17878.3827457721
Iter#  2682,  time =  1077.19, f =  17878.3417261790
```

It shows no significant differences from the case of 6 threads, except starting 9 threads takes more time than starting 6 threads, which adds a little time to the first iterations. This effect can be seen even better with the timing values of 12 threads. There is actually a small slowdown, the CPU use increases to 88…94% and the memory use increases to 150…160 MB (seen in the Windows Task Manager):

```
Iter#     6,  time =     3.99, f =  17878.6910394117
Iter#     0,  time =     4.79, f =  17878.3827457723
Iter#    21,  time =    11.13, f =  17878.3827457721
Iter#  2682,  time =  1112.26, f =  17878.3417261790
```

In this light, for the best performance the number of threads should be the same as the number of physical CPU cores (6 in our case).

5.4. Periodically Saving Snapshots

To be able to see if the program is alive and works, the program constantly prints status information on the terminal. Every time a numerical optimization finishes, the program prints the corresponding iteration (restart) number, and the elapsed time since the restart loop started. The printed information overwrites the previous one, printed in the same line, unless a new best solution is found. In that case the new, current best solution is also printed, and a new line starts. This way the user sees how the optimization progresses.

The optimization process has to run for a very long time, several days for larger N values. If a power outage, playing children, overheating, hardware error, or something else crashes the PC, we want to be

able to resume work with minimal loss of already performed computation. There are several options we can choose from:

1. The information on the current iteration, which is printed to the terminal, can be also saved into a file, constantly overwriting the last line, until a new best solution is found. This method was implemented in the Julia version of the optimization program, performing a very large number of physical file I/O (needing fflush()).
2. The screen information can also be written into an array shared by the threads, and this array is occasionally written to disk. This method is much better, but slightly slower than the next option, which was actually implemented.
3. Always, when a certain preset time is elapsed (e.g., 5 minutes) the current content of the *buffer* of the Windows *terminal* is read, and saved to disk, after rewinding the file. The file buffer is then flushed, such that a crash does not destroy the information still in the buffer. This method only requires a few lines of code, no shared arrays, and it does not have a noticeable impact on the program performance.

5.5. Interrupting the Program

In Windows the standard way of checking if a key is pressed (signaling the user's intention to interrupt the optimization) is using the intrinsic functions _kbhit() and _getch(). The drawback of using them in a *single threaded* program is that we can only check the keyboard between restarts of the numerical optimizations, which can take several seconds for larger N values – and so the user does not see any reaction for a while. (There is no such delay in the multi-threaded program, as the main thread works in parallel with the optimization threads.)

An easy alternative was to use Ctrl-C for stop. To be able to gracefully terminate the program, we have to define a signal handler function. The signal handler prints an acknowledgment (e.g., in the title of the terminal), and broadcasts a STOP signal (e.g., assigns "iter" a value above the last iteration number, like iterX+1). This is seen by every thread, which then terminates.

When a thread terminates, it destroys its NLopt optimization object and the memory it has allocated, then it returns to the main thread, where its handle will be closed.

The timer loop in the main thread also checks for the STOP signal, and breaks out of the loop, if requested. After the main loop exited, the program waits for all the threads to terminate, closes their handles, and saves the final snapshot of the terminal to disk. It is followed by closing the snapshot file, and freeing the allocated global memory.

5.6. Printing the Coordinates of the Best Solution

For larger N values (we ran the program up to N = 322), the every list of coordinates takes several pages, and they are rarely used for anything. Instead of always printing them, the coordinates are only printed, on-demand. If the command line parameters set the starting iteration at least as large as the stopping iteration number, a single numerical optimization is performed with the stopping "iterX" value, and the optimum together with the coordinates of the corresponding points are printed to the output file.

Note that this numerical optimization is always performed in a single thread, therefore the "threadcount" command line parameter has no effect, and can be set to any value (allowed 1…33). It is

included in the output file name, though, thus the threadcount can be used to distinguish this special output file from the snapshot files of the true optimization. The use of threadcount = 33 is recommended, when 33 threads are otherwise not used.

5.7. Memory Pool

In the thread functions large chunks of memory (to be used for temporary data) have to be allocated. It is better done only once for each thread, before its restart-optimization loop starts. This removes the overhead of allocating / freeing temporary storage during numerical optimizations.

5.8. Objective Function Speedups

By moving the computation of the partial derivative directly into the objective function, we can eliminate the arrays Q, and Yu of the Julia program. The corresponding data is in-lined.

Scaling the function value and the derivatives by $2\sqrt{2}$ many multiplications with constants can be removed. Only the final optimization results have to be corrected accordingly.

The distances of the spherical points are computed as before, using the formula:

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} = \sqrt{2} \cdot \sqrt{1 - x_i x_j - y_i y_j - z_i z_j}$$

Instead of indexing array elements, pointer arithmetic is used.

Since we only deal with the Thomson-1 problem here, the PW2(x) function of the Julia program is directly substituted with sqrt(x), with the corresponding trivial simplifications.

5.9. Achieved Speedup

For a difficult case: N = 201, the Julia program ran 5620-3002 = 2618 iterations in 137,889 seconds, that is one iteration took 137889/2618 = 52.67 seconds.

The C program performed 10,535 iterations in 13,083 seconds. It means 13083/10535 = 1.24 sec per iteration, which represents a huge, 52.67/1.24 = 42.5-fold speedup. At other N values we observed similar performance increase.

6. Experiments

Table 3 and Table 4 below contain our numerical optimization results for the Thomson-1 problem, illustrating the viability of the chosen approach, the random restart numerical optimizations.

- The first column (*N*) shows the number of points in the Thomson-1 set.
- The second column (E_i) contains the best solutions found so far according to https://en.wikipedia.org/wiki/Thomson_problem (taken 12/30/2020) – for comparison.
- The third column shows the best Thomson solutions found with the programs listed in this document, using a home PC.
- The 4th column of the table (*Iter#*) has the iteration numbers of our optimization program, at which the optimum was found. In Table 3 the results are listed up to *N* = 257 points, obtained by the Julia program as discussed in Section 4 (especially Section 4.2). Table 4 lists the results of the multi-threaded C program for *N* = 258…322 points, as described in Section 5.
- The coordinates of the corresponding Thomson sets can be obtained with both programs. The Julia program has to be started with two command line arguments:

  `<N>` and `<Iter#>`.

  For the C program 5 arguments have to be given:

  `<N> <Mitchell> <threads> <end_iter> <start_iter>`.

  If `<end_iter>` is not larger than `<start_iter>`, a single numerical optimization is performed with initial values derived from the seed `<start_iter>`, and the coordinates of the corresponding local optimum set is saved in the snapshot file, to disk. The computations should take just seconds on a home PC.

Blue iteration numbers indicate, where the found best solution needed more iterations than the hardcoded limit, in the Julia program. For 200 ≤ N ≤ 257 a second round of optimizations were performed with the Julia program, with iteration limits 6 times of the default. Several times the found optima were slightly improved, which indicate that for larger problem sizes, some of our tabulated results could be improved with more random restarts.

All the solutions found by the presented programs agreed with the best published results.

With the iteration limits *hardcoded* in the Julia program, until N = 257 only 4 of the found solutions were inferior to the Wikipedia entries: N = 197, 201, 256 and 257 (disregarding the minor differences, caused by rounding- and tolerance issues at N = 107 and 139). For these N values larger number of restarts (10,000) reached the best published results:

N	E_i	*Num-Opt*	*Iter#*
197	17878.340162571	17878.340162571	**7004**
201	18627.591226244	18627.591226244	**3386**
256	30506.687515847	30506.687515847	**6649**
257	30749.941417346	30749.941417346	**9864**

Table 2 . Difficult Cases

The running time of the Julia program for N = 256 was 11 days in a single thread of a home PC, and for N = 257 it was 12 days. (Re)computing the coordinates of the points of any single restart iteration takes less than 2 minutes.

N	E_i	Num-Opt	Iter#
2	0.500000000		
3	1.732050808	1.7320508075689	1
4	3.674234614	3.6742346141748	1
5	6.474691495	6.4746914946882	1
6	9.985281374	9.9852813742386	1
7	14.452977414	14.452977414221	1
8	19.675287861	19.675287861233	1
9	25.759986531	25.759986531270	1
10	32.716949460	32.716949460148	1
11	40.596450510	40.596450508191	1
12	49.165253058	49.165253057629	1
13	58.853230612	58.853230611702	1
14	69.306363297	69.306363296626	1
15	80.670244114	80.670244114294	1
16	92.911655302	92.911655302545	1
17	106.050404829	106.05040482862	1
18	120.084467447	120.08446744749	1
19	135.089467557	135.08946755668	1
20	150.881568334	150.88156833376	1
21	167.641622399	167.64162239927	1
22	185.287536149	185.28753614931	1
23	203.930190663	203.93019066288	1
24	223.347074052	223.34707405181	1
25	243.812760299	243.81276029877	1
26	265.133326317	265.13332631736	1
27	287.302615033	287.30261503304	1
28	310.491542358	310.49154235820	1
29	334.634439920	334.63443992042	1
30	359.603945904	359.60394590376	1
31	385.530838063	385.53083806330	1
32	412.261274651	412.26127465053	1
33	440.204057448	440.20405744765	1

N	E_i	*Num-Opt*	*Iter#*
34	468.904853281	468.90485328134	1
35	498.569872491	498.56987249065	1
36	529.122408375	529.12240837541	1
37	560.618887731	560.61888773104	9
38	593.038503566	593.03850356645	10
39	626.389009017	626.38900901682	3
40	660.675278835	660.67527883462	1
41	695.916744342	695.91674434189	1
42	732.078107544	732.07810754367	1
43	769.190846459	769.19084645916	1
44	807.174263085	807.17426308463	1
45	846.188401061	846.18840106108	1
46	886.167113639	886.16711363919	1
47	927.059270680	927.05927067971	1
48	968.713455344	968.71345534379	1
49	1011.557182654	1011.5571826536	1
50	1055.182314726	1055.1823147263	1
51	1099.819290319	1099.8192903189	1
52	1145.418964319	1145.4189643193	2
53	1191.922290416	1191.9222904162	1
54	1239.361474729	1239.3614747292	1
55	1287.772720783	1287.7727207827	1
56	1337.094945276	1337.0949452757	3
57	1387.383229253	1387.3832292528	1
58	1438.618250640	1438.6182506404	1
59	1490.773335279	1490.7733352787	4
60	1543.830400976	1543.8304009764	1
61	1597.941830199	1597.9418301990	1
62	1652.909409898	1652.9094098983	2
63	1708.879681503	1708.8796815033	1
64	1765.802577927	1765.8025779273	1
65	1823.667960264	1823.6679602639	1
66	1882.441525304	1882.4415253042	1

N	E_i	*Num-Opt*	*Iter#*
67	1942.122700406	1942.1227004055	1
68	2002.874701749	2002.8747017487	3
69	2064.533483235	2064.5334832348	6
70	2127.100901551	2127.1009015506	1
71	2190.649906425	2190.6499064258	1
72	2255.001190975	2255.0011909750	1
73	2320.633883745	2320.6338837454	3
74	2387.072981838	2387.0729818383	1
75	2454.369689040	2454.3696890396	1
76	2522.674871841	2522.6748718414	1
77	2591.850152354	2591.8501523539	1
78	2662.046474566	2662.0464745663	52
79	2733.248357479	2733.2483574788	5
80	2805.355875981	2805.3558759812	1
81	2878.522829664	2878.5228296641	1
82	2952.569675286	2952.5696752865	1
83	3027.528488921	3027.5284889211	2
84	3103.465124431	3103.4651244308	2
85	3180.361442939	3180.3614429386	1
86	3258.211605713	3258.2116057128	4
87	3337.000750014	3337.0007500145	17
88	3416.720196758	3416.7201967584	4
89	3497.439018625	3497.4390186247	2
90	3579.091222723	3579.0912227228	2
91	3661.713699320	3661.7136993200	1
92	3745.291636241	3745.2916362407	1
93	3829.844338421	3829.8443384214	2
94	3915.309269620	3915.3092696204	9
95	4001.771675565	4001.7716755650	1
96	4089.154010060	4089.1540100556	1
97	4177.533599622	4177.5335996222	1
98	4266.822464156	4266.8224641562	1
99	4357.139163132	4357.1391631318	1

N	E_i	*Num-Opt*	*Iter#*
100	4448.350634331	4448.3506343313	1
101	4540.590051694	4540.5900516944	1
102	4633.736565899	4633.7365658987	1
103	4727.836616833	4727.8366168330	5
104	4822.876522746	4822.8765227487	9
105	4919.000637616	4919.0006376157	18
106	5015.984595705	5015.9845957047	7
107	5113.953547724	5113.9535477138	8
108	5212.813507831	5212.8135078306	1
109	5312.735079920	5312.7350799202	1
110	5413.549294192	5413.5492941924	6
111	5515.293214587	5515.2932145866	1
112	5618.044882327	5618.0448823266	3
113	5721.824978027	5721.8249780271	12
114	5826.521572163	5826.5215721627	4
115	5932.181285777	5932.1812857773	8
116	6038.815593579	6038.8155935785	76
117	6146.342446579	6146.3424465786	7
118	6254.877027790	6254.8770277896	10
119	6364.347317479	6364.3473174792	22
120	6474.756324980	6474.7563249797	8
121	6586.121949584	6586.1219495841	1
122	6698.374499261	6698.3744992606	105
123	6811.827228174	6811.8272281741	7
124	6926.169974193	6926.1699741935	1
125	7041.473264023	7041.4732640232	24
126	7157.669224867	7157.6692248670	9
127	7274.819504675	7274.8195046756	12
128	7393.007443068	7393.0074430680	6
129	7512.107319268	7512.1073192683	19
130	7632.167378912	7632.1673789125	1
131	7753.205166941	7753.2051669406	10
132	7875.045342797	7875.0453427971	8
133	7998.179212898	7998.1792128984	9

N	E_i	*Num-Opt*	*Iter#*
134	8122.089721194	8122.0897211942	7
135	8246.909486992	8246.9094869920	1
136	8372.743302539	8372.7433025386	1
137	8499.534494782	8499.5344947815	4
138	8627.406389880	8627.4063898801	3
139	8756.227056057	8756.2270569530	9
140	8885.980609041	8885.9806090406	1
141	9016.615349190	9016.6153491899	8
142	9148.271579993	9148.2715799935	9
143	9280.839851192	9280.8398511922	52
144	9414.371794460	9414.3717944599	7
145	9548.928837232	9548.9288372318	10
146	9684.381825575	9684.3818255749	3
147	9820.932378373	9820.9323783732	1
148	9958.406004270	9958.4060042699	6
149	10096.859907397	10096.859907397	1
150	10236.196436701	10236.196436701	7
151	10376.571469275	10376.571469275	6
152	10517.867592878	10517.867592878	83
153	10660.082748237	10660.082748236	54
154	10803.372421141	10803.372421141	8
155	10947.574692279	10947.574692279	8
156	11092.798311456	11092.798311456	143
157	11238.903041156	11238.903041156	237
158	11385.990186197	11385.990186197	1
159	11534.023960956	11534.023960956	27
160	11683.054805549	11683.054805549	11
161	11833.084739465	11833.084739465	28
162	11984.050335814	11984.050335814	37
163	12136.013053220	12136.013053220	12
164	12288.930105320	12288.930105320	14
165	12442.804451373	12442.804451373	16
166	12597.649071323	12597.649071323	123

N	E_i	Num-Opt	Iter#
167	12753.469429750	12753.469429750	5
168	12910.212672268	12910.212672268	10
169	13068.006451127	13068.006451127	11
170	13226.681078541	13226.681078540	165
171	13386.355930717	13386.355930717	21
172	13547.018108787	13547.018108787	62
173	13708.635243034	13708.635243034	22
174	13871.187092292	13871.187092292	1
175	14034.781306929	14034.781306929	32
176	14199.354775632	14199.354775632	17
177	14364.837545298	14364.837545298	569
178	14531.309552587	14531.309552588	70
179	14698.754594220	14698.754594220	8
180	14867.099927525	14867.099927525	82
181	15036.467239769	15036.467239769	6
182	15206.730610906	15206.730610906	4
183	15378.166571028	15378.166571028	8
184	15550.421450311	15550.421450311	1036
185	15723.720074072	15723.720074072	280
186	15897.897437048	15897.897437048	1
187	16072.975186320	16072.975186320	9
188	16249.222678879	16249.222678879	514
189	16426.371938862	16426.371938864	22
190	16604.428338501	16604.428338501	19
191	16783.452219362	16783.452219363	29
192	16963.338386460	16963.338386461	7
193	17144.564740880	17144.564740880	81
194	17326.616136471	17326.616136471	184
195	17509.489303930	17509.489303930	18
196	17693.460548082	17693.460548082	70
197	17878.340162571	17878.340162571	**7004**
198	18064.262177195	18064.262177195	113
199	18251.082495640	18251.082495640	196
200	18438.842717530	18438.842717530	245

N	E_i	Num-Opt	Iter#
201	18627.591226244	18627.591226244	**3386**
202	18817.204718262	18817.204718262	60
203	19007.981204580	19007.981204580	274
204	19199.540775603	19199.540775603	651
205		19392.369152388	12
206		19585.955856549	272
207		19780.656909314	62
208		19976.203261203	237
209		20172.754680661	578
210		20370.251615129	693
211		20568.740602776	280
212	20768.053085964	20768.053085964	488
213		20968.612025491	74
214	21169.910410375	21169.910410375	363
215		21372.348789341	366
216	21575.596377869	21575.596377869	53
217	21779.856080418	21779.856080418	27
218		21985.263948921	182
219		22191.485474814	155
220		22398.655602531	174
221		22606.881547259	23
222		22816.025570249	57
223		23026.165881160	8
224		23237.244873487	3
225		23449.436460667	12
226		23662.511122654	221
227		23876.576893718	136
228		24091.578985867	36
229		24307.599313290	288
230		24524.485350945	139
231		24742.382494768	126
232	24961.252318934	24961.252318934	608
233		25181.057399530	1

N	E_i	*Num-Opt*	*Iter#*
234		25401.931786640	92
235		25623.763144201	37
236		25846.500563152	136
237		26070.367015459	898
238		26295.126047904	1221
239		26520.874117442	444
240		26747.508205092	9
241		26975.190283978	756
242		27203.799285553	1886
243		27433.367352376	418
244		27663.905112181	1355
245		27895.540745391	355
246		28128.051464319	5421
247		28361.532777263	17
248		28596.052102948	1213
249		28831.473909851	575
250		29067.889163423	940
251		29305.233861625	5335
252		29543.522868125	197
253		29782.917364395	84
254		30023.222861843	4728
255	30264.424251281	30264.424251281	307
256	30506.687515847	30506.687515847	**6649**
257	30749.941417346	30749.941417346	**9864**

Table 3. Best Results of the Julia Program

For the number of points N > 257 the much faster multithreaded C program was used, with Mitchell number 3. The results are tabulated below, with highlighted (blue) iteration numbers, when more than 50,000 restarts led to the listed results (4 cases of N = 306, 315, 317 and 321).

N	E_i	*Num-Opt*	*Iter#*
258		30994.2135774868	611
259		31239.4423068662	1123
260		31485.5781917802	1760
261		31732.7427877295	3223
262		31980.8518238660	1199
263		32229.9148956472	2913
264		32479.9081411766	249
265		32731.0123974668	6659
266		32982.9836290832	2588
267		33235.9620660390	633
268		33489.8744849436	3243
269		33744.8007327701	2598
270		34000.6484970195	40677
271		34257.4417381447	8915
272	34515.193292681	34515.1932926817	10303
273		34774.2578509450	658
274		35034.0334667000	2803
275		35294.8940512130	14388
276		35556.6333035795	17841
277		35819.3408109533	1006
278		36083.0368937559	2103
279		36347.6584257292	1587

N	E_i	*Num-Opt*	*Iter#*
280		36613.2695900773	1005
281		36879.8056153435	3166
282	37147.294418462	37147.2944184620	14968
283		37416.0757427823	1101
284		37685.6897932422	1326
285		37956.2394559865	8052
286		38227.6733205442	4850
287		38500.1343322855	1856
288		38773.5812836889	17850
289		39048.0491867372	15313
290		39323.4100620378	22241
291		39599.7118168330	10848
292	39877.008012909	39877.0080129083	1624
293		40155.3490760206	1974
294		40434.7622974846	1280
295		40715.1417167072	2166
296		40996.4139085292	654
297		41278.7032930107	1894
298		41561.9207687963	25409
299		41846.0811834311	2832
300		42131.2641372452	44534

N	E_i	*Num-Opt*	*Iter#*
301		42417.3566298396	45153
302		42704.4937584797	10025
303		42992.5289668779	6886
304		43281.5133649633	22859
305		43571.5504536883	14785
306	43862.569780797	43862.5697807962	**78567**
307		44154.5477059508	8530
308		44447.4530563589	1376
309		44741.5642838065	2714
310		45036.4985722224	25326
311		45332.4021055010	1139
312	45629.313804002	45629.3138040044	48767
313		45927.1925470308	38645
314		46225.9797575717	35275
315	46525.825643432	46525.8256434321	**173295**
316		46826.5884028806	48055
317	47128.310344520	47128.3103445197	**69909**
318	47431.056020043	47431.0560200432	31743
319		47734.8137639722	9808
320		48039.4689301648	4665
321		48345.1609527838	**65053**
322		48651.7621823975	7427

Table 4. Best Results of the Multithreaded C Program

N	E_i	Num-Opt	Iter#
334	52407.728127822		
348	56967.472454334		
357	59999.922939598		
358	60341.830924588		
372	65230.027122557		
382	68839.426839215		
390	71797.035335953		
392	72546.258370889		
400	75582.448512213		
402	76351.192432673		
432	88353.709681956		
448	95115.546986209		
460	100351.763108673		
468	103920.871715127		
470	104822.886324279		

Table 5. A Few Published Results for N > 322

7. Conclusions

The presented computer programs found all of the best published solutions to the Thomson-1 problem for N ≤ 322, when performing sufficiently many random restarts. Table 3 and Table 4 list these optima together with the iteration number, where they were found. Starting the programs with the appropriate command line arguments runs a single numerical optimization, which give the coordinates of the points in the corresponding Thomson set. With the C program the corresponding computations take only seconds for the number of points N ≤ 322.

The optimum for some of the larger N values were reached only at large number of restarts, showing the limitations of the random restart numerical optimization: fewer restarts would miss the best solutions. In Table 4 the results for N = 258…322 are listed, which were obtained by the multi-threaded C program. 50,000 restarts were performed for N = 258…299, and 175,000 restarts for N = 300…322. Beyond that the random-restart numerical optimization technique becomes impractically slow for a home PC.

In the range N = 258…299, fewer restarts were needed, still, in 4 cases the best local minima were found after more than 20,000 restarts, too close to the 50,000 set as the limit. For these N values the numerical optimizations were repeated with Mitchell number 2, instead of 3, effectively performing another 50,000 random restarts. The found local optima, as summarized in the Table 8, were not improved, indicating that the solution found before could actually be the true, global minima.

The performed experiments indicate that the number of necessary restarts of randomly initialized numerical optimizations, to find the best known local (and hopefully global) optima, grows rapidly with N. This growth has been shown in [18] to be super-exponential, (because the number of solutions

of polynomial systems of equations grows very fast with N. At a local optimum the partial derivatives are 0, and a significant portion of such solutions correspond to local minima.)

…N	35	75	120	175	225	300	325
Restarts	1	10	100	1,000	10,000	100,000	1,000,000

Table 6. Estimates on the Number of Necessary Restarts, Dependent of N

The discussed C program can also handle larger numbers of points, e.g., N = 500, too, but this may not be practical on personal computers. The running time of the individual numerical optimizations increases cubically with N (due to internal operations on dense matrices of dimensions proportional to N), but even more significantly, the necessary number of random restarts increases super exponentially.

Using a cheap personal computer, this random-restart numerical local optimization method seems to be feasible up to N = 325. Faster workstations may find good local minima till N ≈ 360, at reasonable running times. Handling even larger N values may need a network of computers, or a fast mainframe.

Even though we don't know, how many local optima exist, and can only estimate the number of restarts needed for some confidence in the global optimality of the best numerical solutions, we hope that the presented results of our experiments are useful. E.g., other optimalization strategies can be compared to the presented one.

8. References

[1] 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).

[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. “L. Hars. “Numerical Solutions of Mid-Size Tammes Problems: N = 61…100”. http://www.hars.us/Papers/Midsize Numerical_Tammes.pdf.

[16] CMake: an open-source, cross-platform family of tools designed to build, test and package software. https://cmake.org/.

[17] CLion: A cross-platform IDE for C and C++. https://www.jetbrains.com/clion/

[18] Garcia, C. B., and Tien-Yian Li. "On the number of solutions to polynomial systems of equations." *SIAM Journal on Numerical Analysis* 17.4 (1980): 540-546.

[19] Microsoft C++, C, and Assembler documentation. https://docs.microsoft.com/en-us/cpp/?view=msvc-170.

[20] Sphere Point Picking, Wolfram MathWorld: https://mathworld.wolfram.com/SpherePointPicking.html.

9. Appendix

9.1. Experiments with the Multithreaded C Program

According to the experiments discussed earlier, the Mitchell number was set to 3, and 50,000 or 175,000 (pseudo)random restarts were performed, dependent on the number of points N.

The numerical optimization results are stored in separate files for each N. The following AutoHotkey (AHK) script collects the relevant information from these files and prints it out.

```
#SingleInstance Force
#NoEnv
SetBatchLines -1
SetFormat FloatFast, 0.16

Loop d:\Thomson\Thomson-???-03-06.txt {     ; Loop over files in alphabetical order
	Loop Read, %A_LoopFileFullPath%     ; Read lines of each file
		if (A_Index == 1)
			N := SubStr(A_LoopReadLine, 5, 3)
		else {                          ; Store information in arrays
			I%A_Index% := SubStr(A_LoopReadLine, 6, 6)
			T%A_Index% := SubStr(A_LoopReadLine,21, 9)
			V%A_Index% := SubStr(A_LoopReadLine,35,18)
			if V%A_Index% is not number  ; Last line
			{
				L := A_Index - 1
				W := V%L%      ; Best solution
				Break
			}
	}
	Loop % L - 1 {                      ; Find 1st solution
		J := A_Index + 1
		R := 1.0 - W / V%J%
		if (R < 2.0e-14)
			Break
	}
	A .= N ", " I%J% ", " T%J% ", " V%J% ", " W "`n"
}
MsgBox %A%                                  ; Print CSV data
```

The printout from the AHK MsgBox is then copied and pasted to the table below:

N	Restarts	Time (s)	Solution @ $2 \cdot 10^{-14}$	Best Solution
197	1604	353.06	17878.3401625712	17878.3401625712
201	99	25.02	18627.5912262442	18627.5912262442
210	1331	365.78	20370.2516151291	20370.2516151291
258	611	340.58	30994.2135774868	30994.2135774868
259	1123	617.48	31239.4423068662	31239.4423068662
260	1760	1048.41	31485.5781917802	31485.5781917802
261	3223	1885.20	31732.7427877299	31732.7427877295
262	1199	650.71	31980.8518238665	31980.8518238660
263	2913	1727.21	32229.9148956473	32229.9148956472
264	249	144.92	32479.9081411769	32479.9081411766
265	6659	3905.24	32731.0123974670	32731.0123974668
266	2588	1605.57	32982.9836290834	32982.9836290832
267	633	386.44	33235.9620660392	33235.9620660390
268	3243	1996.39	33489.8744849437	33489.8744849436
269	2598	1694.78	33744.8007327706	33744.8007327701
270	40677	25893.29	34000.6484970195	34000.6484970195
271	8915	5785.62	34257.4417381447	34257.4417381447
272	10303	6564.81	34515.1932926817	34515.1932926817
273	658	424.91	34774.2578509450	34774.2578509450
274	2803	1958.84	35034.0334667006	35034.0334667000
275	14388	10299.50	35294.8940512137	35294.8940512130
276	17841	12937.33	35556.6333035795	35556.6333035795
277	1006	748.21	35819.3408109534	35819.3408109533
278	2103	1442.06	36083.0368937564	36083.0368937559
279	1587	1123.22	36347.6584257299	36347.6584257292

N	Restarts	Time (s)	Solution @ $2 \cdot 10^{-14}$	Best Solution
280	1005	754.49	36613.2695900775	36613.2695900773
281	3166	2378.93	36879.8056153439	36879.8056153435
282	14968	10565.86	37147.2944184622	37147.2944184620
283	1101	842.73	37416.0757427828	37416.0757427823
284	1326	1044.00	37685.6897932424	37685.6897932422
285	8052	5857.98	37956.2394559865	37956.2394559865
286	4850	3405.15	38227.6733205448	38227.6733205442
287	1856	1307.14	38500.1343322855	38500.1343322855
288	17850	13072.71	38773.5812836889	38773.5812836889
289	15313	11582.57	39048.0491867372	39048.0491867372
290	22241	17707.80	39323.4100620381	39323.4100620378
291	10848	8426.98	39599.7118168336	39599.7118168330
292	1624	1285.40	39877.0080129087	39877.0080129083
293	1974	1730.42	40155.3490760212	40155.3490760206
294	1280	1043.88	40434.7622974852	40434.7622974846
295	2166	1826.45	40715.1417167077	40715.1417167072
296	654	531.86	40996.4139085298	40996.4139085292
297	1894	1464.74	41278.7032930114	41278.7032930107
298	25409	21196.19	41561.9207687963	41561.9207687963
299	2832	2487.07	41846.0811834316	41846.0811834311
300	44534	40022.58	42131.2641372455	42131.2641372455

Table 7. Results of the 6-threaded C Program (M = 3, MaxIter = 50,000)

At a few cases the optimum was found at a restart number close to our preset limit (e.g., N = 270 needed 40,677 random restarts). At the four N values, where the best solution was only found after more than 20,000 random restarts, the numerical optimizations were repeated with the second-best Mitchell number, M = 2, for another 50,000 restarts. (Other alternatives were to change the seed of the pseudorandom number generator and repeat the optimizations; or continue with restart numbers = 50,001…175,000.)

The command for the re-runs in a Command Prompt window was:

```
for %n in (270, 290, 298, 300) do ThomsonX.exe %n 2 6
```

These gave the following results:

N	Restarts	Time (s)	Solution @ $2 \cdot 10^{-14}$	Best Solution
270	8997	5956.58	34000.6484970195	34000.6484970195
290	6740	5497.04	39323.4100620376	39323.4100620376
298	13707	12371.41	41561.9207687966	41561.9207687965
300	39398	34979.94	42131.2641372455	42131.2641372455

Table 8. Rerun the Difficult Cases of the Multithreaded C Program (M = 2, MaxIter = 50,000)

The N = 300 case still required too many restarts (although the best numerical solution was found to be the same as before), therefore we ran the optimizations a third time, with the Mitchell number set to 4, but with that the known best local optimum was not found. At restart number 32386, (time = 27957.66) the local optimum was f = 42131.2793895808.

An older PC, with a quad-core processor was also used to perform numerical optimizations, in a 4-thread configuration. This PC runs 35% slower than our main, hex-core PC. Below are the results:

N	Restarts	Time (s)	Solution @ $2 \cdot 10^{-14}$	Best Solution
301	45153	51307.84	42417.3566298396	42417.3566298396
302	10025	11574.88	42704.4937584799	42704.4937584797
303	6886	7903.16	42992.5289668784	42992.5289668779
304	22859	26516.71	43281.5133649637	43281.5133649635
305	14785	17452.28	43571.5504536891	43571.5504536883
306	6938	8243.37	43862.6041879979	43862.6041879973
307	8530	10286.99	44154.5477059508	44154.5477059508
308	1376	1686.37	44447.4530563597	44447.4530563589
309	2714	3353.79	44741.5642838072	44741.5642838065
310	25326	31267.19	45036.4985722226	45036.4985722226
311	1139	1438.44	45332.4021055013	45332.4021055013
312	48767	61794.20	45629.3138040044	45629.3138040044
313	38645	49296.83	45927.1925470310	45927.1925470308
314	35275	45749.97	46225.9797575720	46225.9797575720
315	43844	57447.88	46525.8415464685	46525.8415464685
316	48055	63514.43	46826.5884028806	46826.5884028806
317	20123	26889.41	47128.3647566270	47128.3647566270
318	31743	42413.26	47431.0560200437	47431.0560200437
319	9808	13314.52	47734.8137639723	47734.8137639723
320	4665	6384.48	48039.4689301657	48039.4689301648
321	9507	13222.53	48345.1636460244	48345.1636460243
322	7427	10394.39	48651.7621823984	48651.7621823976

Table 9. Results of the 4-threaded C Program (M = 3, MaxIter = 50,000)

With more points, the number of local optima increases rapidly. As expected, in the range of N = 301…322, there were many more: nine N values, where the numerical optimization found the best results after more than 20,000 restarts. To improve the chances that the found local optima is the global optimum, we repeated the optimizations with another 125,000 restarts (iter = 50,001…175,000):

```
for /l %n in (300, 1, 322) do ThomsonX.exe %n 3 6 175000 50001
```

This gave the results shown in Table 4.

Still, at these large N values our confidence is low, that the found best local solutions are actually global optima. Many more random restarts seem to be needed, with the estimates shown in Table 6.

9.2. The Multithreaded C Program Code

```
/* ---- Thomson.c: Random restart numerical optimization for the Thomson problem
|       using the NLopt library (version 2.7.1), built with CLion IDE
| by Laszlo Hars, 12/16/2021
|
| Written for Windows 10, C standard: ISO C17
|
| Need 3 files: nlopt.dll, nlopt.lib (built with CLion, Release/Debug) and nlopt.h (source of nlopt)
| Copy release/debug versions of nlopt.dll and nlopt.lib to release/debug project directories
|
| VS: %USERPROFILE%\source\repos\Thomson\x64\<Release/Debug>
| Module- and project-target must match (x64) drop-down on VS 2nd menu line
| Project Properties: Linker/Input/Additional Dependencies add $(OutDir)nlopt.lib;
|
| CLion: add 3 lines to the end of the default CMakeLists.txt ->
|   add_library(NLopt SHARED IMPORTED)
|   set_target_properties(NLopt PROPERTIES IMPORTED_IMPLIB "nlopt.lib")
|   target_link_libraries(<project> PUBLIC NLopt)
|
| Spherical points are represented by their Stereographic Projected images on the [u,v] plane
| P[-2] = (0,0,1); North Pole, hard coded (projected to infinity)
| P[-1] = (X,0,Z); where X = sqrt(1-Z*Z). First optimization variable: x[0] <- Z
| For i = 0..N-3: P[i] projected to planar point [u,v], packed into x: x[2i+1] = u, x[2i+2] = v
\-----------------------------------------------------------------------------------------*/

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <stdint.h>
#include <windows.h>
#include <time.h>
#include <signal.h>

#define _USE_MATH_DEFINES
#include <math.h>
#include "nlopt.h"                              // copied to project directory from nlopt-2.7.1\src\api

// Globals
#define MAXTHREADS 33
#define CONW 53                                 // console sizes
#define CONH 70

typedef double* VEC;
HANDLE mtxi, mtxo;                              // input/output mutexes for exclusive access

int n, N = 0, N2;                               // parameters
int MITCHELL, threadcount;
int lines, iter, iterX;                         // #info lines printed, restart number/stop
uint32_t seed = 0;                              // seed for the xTEA hash
const double eps = 1e-15, pwc = .5 * M_SQRT1_2; // tolerance, scaling
double mf = INFINITY, *mx;                      // the current best solutions
clock_t cl0;                                    // clock at start
double tosec = 1. / CLOCKS_PER_SEC;             // scale clock to second

double f0(const double* x, VEC mem) {           // Objective function w/o gradients
    VEC X = mem, Y = X + N2, Z = Y + N2, w = (VEC)x;
    double q, s, u, v, x1, z1;

    for (int i = 0; i < N2; ++i) {              // inverse stereographic projection
        u = *++w; v = *++w;
        q = 2.0 / (u * u + v * v + 1.);
        X[i] = q * u;
        Y[i] = q * v;
        Z[i] = 1 - q;
    }
```

```
    z1 = x[0]; x1 = sqrt(1. - z1 * z1);         // special point P[-1]
    s = sqrt(1. / (1. - z1));                   // distance(P[-1],P[-2])

    for (int i = 0; i < N2; ++i)                // distances of P[i] to P[-1], P[-2]
        s += sqrt(1. / (1. - Z[i])) + sqrt(1. / (1. - x1 * X[i] - z1 * Z[i]));

    for (int i = 0; i < N2 - 1; ++i)            // distances(P[i],P[j])
        for (int j = i + 1; j < N2; ++j)
            s += sqrt(1. / (1. - X[i] * X[j] - Y[i] * Y[j] - Z[i] * Z[j]));

    return s + s;                               // total energy scaled up by 2sqrt(2)
}

// OBJECTIVE FUNCTION (total potential energy)
double f(unsigned n, const double* x, double* grad, void* mem) {
    if (!grad) return f0(x, (VEC)mem);

    VEC X = (VEC)mem, Y = X + N2, Z = Y + N2, Xu = Z + N2, Xv = Xu + N2,
        Yu = Xv, Yv = Yu + N2, Zu = Yv + N2, Zv = Zu + N2;
    VEC w = (VEC)x, g = grad, h, gg;
    int i, j;
    double p, q, r, s, t, u, v;

    for (i = 0; i < N2; ++i) {                  // Inverse Stereographic Projection
        u = *++w; v = *++w;
        q = 2.0 / (u * u + v * v + 1.);
        X[i] = q * u;
        Y[i] = q * v;
        Z[i] = 1 - q;

        q *= q;                                 //  partial derivatives
        Xu[i] = t = .5 * (v*v - u*u + 1.) * q;  // dX/du
        Xv[i] = -u * v * q;                     // dX/dv = dY/du
        Yv[i] = q - t;                          // dY/dv
        Zu[i] = u * q;                          // dZ/du
        Zv[i] = v * q;                          // dZ/dv
    }

    double z1 = x[0], x1 = sqrt(1. - z1 * z1);
    t = 1. / (1. - z1); s = sqrt(t);
    grad[0] = t * s;

    for (i = 0; i < N2; ++i) {                  // distances, gradients to special points
        t = 1. / (1. - Z[i]);                   // t, p, q, r are scaled up by 2sqrt(2)
        s += p = sqrt(t);
        t *= p;

        r = 1. / (1. - x1 * X[i] - z1 * Z[i]);
        s += q = sqrt(r);
        r *= q;

        *++g = t * Zu[i] + r * (x1 * Xu[i] + z1 * Zu[i]);
        *++g = t * Zv[i] + r * (x1 * Xv[i] + z1 * Zv[i]);

        grad[0] += r * (-z1 / x1 * X[i] + Z[i]);
    }

    for (i = 0, g = grad + 1, h = gg = g + 1;  // distances between normal points, gradients
        i < N2 - 1;
        ++i, g += 2, h = gg = g + 1) {
        for (j = i + 1; j < N2; ++j) {
            t = 1. / (1. - X[i] * X[j] - Y[i] * Y[j] - Z[i] * Z[j]);
            s += q = sqrt(t);
            t *= q;

            *g += t * (Xu[i] * X[j] + Yu[i] * Y[j] + Zu[i] * Z[j]);
            *h += t * (Xv[i] * X[j] + Yv[i] * Y[j] + Zv[i] * Z[j]);
            *++gg += t * (Xu[j] * X[i] + Yu[j] * Y[i] + Zu[j] * Z[i]);
            *++gg += t * (Xv[j] * X[i] + Yv[j] * Y[i] + Zv[j] * Z[i]);
```

```
        }
    }

    return s + s;                               // total energy and gradients are scaled up by 2sqrt(2)
}

uint64_t h(uint32_t u, uint32_t v, uint32_t w) {// xTEA cipher based hash
    const uint32_t KEY[4] = { 0x622E76FA^seed, 0x0FE5D146, 0xC82AC7E0, 0x27CD25C7-seed };
    for (int i = 0; i < 32; ++i) {
        u += ((v << 4 ^ v >> 5) + v) ^ (w + KEY[w & 3]);
        w += 0x9E3779B9;
        v += ((u << 4 ^ u >> 5) + u) ^ (w + KEY[(w >> 11) & 3]);
    }
    return (uint64_t)u << 32 | v;
}


void srnd(uint32_t i, uint32_t j, uint32_t k, VEC x, VEC y, VEC z) {  // spherical pseudorandom points
    double t, u, s;
    t = ldexp((double)h(i, j, k), -63) * M_PI;  // uniform in [0,2pi)
    u = ldexp((double)h(k, i, j), -63) - 1.;    // uniform in [-1,1)
    s = sqrt(1. - u * u);

    *x = s * cos(t);
    *y = s * sin(t);
    *z = u;
}

void initU(int iter, VEC x) {                   // Uniform pseudorandom initial spherical points
    double X, Y, Z;
    VEC w = x;
    x[0] = sin(M_PI_2 * ldexp((double)h(iter, 0x5ea899db, 123), -63) - 1.); // z1
    for (uint64_t i = 0; i < N2; ++i) {
        srnd(iter, i, 9, &X, &Y, &Z);
        *++w = X / (1 - Z);
        *++w = Y / (1 - Z);
    }
}

void initM(int iter, VEC x, VEC mem) {          // Mitchell pseudorandom initial points
    int i, j, k;
    double dmin, dmax, t, vx, vy, vz,
        vminx, vminy, vminz,
        vmaxx, vmaxy, vmaxz;
    VEC X = mem, Y = X + N, Z = Y + N;
    X[0] = 0; Y[0] = 0; Z[0] = 1;               // N points P[0]..P[N-1], including specials
    Z[1] = sin(M_PI_2 * ldexp((double)h(iter, 0x5ea899db, 123), -63) - 1.);
    X[1] = sqrt(1 - Z[1] * Z[1]); Y[1] = 0;

    for (i = 2; i < N; ++i) {
        dmax = 2.0;
        for (k = 0; k < MITCHELL; ++k) {        // GLOBAL MITCHELL
                dmin = -2.0;                    // cos(distance), decreasing
            srnd(iter, i, k, &vx, &vy, &vz);
            for (j = 0; j < i; ++j) {
                t = vx * X[j] + vy * Y[j] + vz * Z[j];
                if (dmin < t) {                 // min(distance) = max(t)
                    dmin = t;
                    vminx = vx; vminy = vy; vminz = vz;
                }
            }
            if (dmax > dmin) {                  // max(min(dist)) = min(dmin)
                dmax = dmin;
                vmaxx = vminx; vmaxy = vminy; vmaxz = vminz;
            }
        }
        X[i] = vmaxx; Y[i] = vmaxy; Z[i] = vmaxz;
    }
    x[0] = Z[1];
```

```
    VEC w = x;
    for (i = 2; i < N; ++i) {
        *++w = X[i] / (1 - Z[i]);
        *++w = Y[i] / (1 - Z[i]);
    }
}

VEC dalloc(int sz) {                            // allocate memory for doubles
    VEC p = (VEC)malloc(sz * sizeof(double));
    if (p) return p;
    else {
        printf("MALLOC ERROR\n");
        exit(-11);
    }
}

DWORD WINAPI Optim(LPVOID Param) {              // THREAD FUNCTION
    nlopt_opt opt;                              // declare the optimization object
    opt = nlopt_create(NLOPT_LD_LBFGS, n);      // set the algorithm and dimensions
    nlopt_srand(0);                             // fix seed for reproducible behavior

    VEC x = dalloc(n);                          // thread-specific optimization variables
    VEC mem = dalloc(8 * N2);                   // ... and scratchpad
    nlopt_set_min_objective(opt, f, (void*)mem);
    nlopt_set_ftol_rel(opt, eps);

    nlopt_set_lower_bounds1(opt, -INFINITY);
    nlopt_set_lower_bound(opt, 0, -1.);
    nlopt_set_upper_bounds1(opt, +INFINITY);
    nlopt_set_upper_bound(opt, 0, +1.);

    double minf;                                // result of optimization
    int it, RC;                                 // iteration number, return code

    for (;;) {                                  // MAIN OPTIMIZATION LOOP ------------
        WaitForSingleObject(mtxi, INFINITE);
        it = iter++;                            // global iteration number to work on, increment
        ReleaseMutex(mtxi);

        if (it > iterX) break;                  // all done

        initM(it, x, mem);                      // uniform or Mitchell random starting points
        RC = nlopt_optimize(opt, x, &minf);     // optimization process...

        WaitForSingleObject(mtxo, INFINITE);    // update/print current iteration number, time
        printf("\x1b[GIter#%6i,  time =%9.2f", it, (clock() - cl0) * tosec);

        if (RC > 0 && minf < mf) {
            mf = minf;                          // update/print current best solution
            printf(", f = % #16.15g\n", minf * pwc);
            lines += 1;                         // 1 line finalized
            for (int i = 0; i < n; ++i) mx[i] = x[i];
        }
        ReleaseMutex(mtxo);
    }

    free(x); free(mem);                         // free thread memory
    nlopt_destroy(opt);

    return 0;
}

void sig_handler(int signo) {                   // set STOP at Ctrl-C
    if (signo != SIGINT) return;
    WaitForSingleObject(mtxi, INFINITE);
        printf("\x1b]0;%45s- - -  S T O P P I N G  - - -\x07","");
        iter = iterX+1;                         // signal STOP to threads
    ReleaseMutex(mtxi);
}
```

```
int main(int argc, char* argv[]) {
    DWORD dwMode = 0;                           // enable virtual terminal sequences
    HANDLE hOut = GetStdHandle(STD_OUTPUT_HANDLE);
    GetConsoleMode(hOut, &dwMode);
    SetConsoleMode(hOut, dwMode | ENABLE_VIRTUAL_TERMINAL_PROCESSING);

    SMALL_RECT consz = { 0, 0, CONW, CONH };    // Console size < cmd Defaults! Sizes will be 1 larger!
    SetConsoleWindowInfo(hOut, 1, &consz);
    SetConsoleScreenBufferSize(hOut, (COORD) { CONW, CONH });

    CONSOLE_SCREEN_BUFFER_INFO buff_info;
    CHAR_INFO buff[CONW][CONH] = { {32,0} };    // Initialize with spaces of default attributes

    N = argc > 1 ? atoi(argv[1]) : 0;
    MITCHELL = argc > 2 ? atoi(argv[2]) : 3;
    threadcount = argc > 3 ? atoi(argv[3]) : 6;
    iterX = argc > 4 ? atoi(argv[4]) : 50000;
    iter  = argc > 5 ? atoi(argv[5]) : 0;
    if (argc < 2 || N < 3 || MITCHELL < 1 || threadcount < 1 || threadcount > 33 || iterX < 1) {
        printf("Command line arguments:\n\t#Points(3..)\n\tMITCHELL(3: 1..)\
               \n\t#Threads(6: 1..33)\n\tMaxIter(50000)\n\tStartIter(0)");
        exit(1);
    }
    printf("N = %d MITCHELL = %d Threads = %d (Ctrl-C stops)\n\n", N, MITCHELL, threadcount);
    N2 = N - 2;
    n = N + N - 3;
    lines = 3;                                  // 3 lines of header written
    mx = dalloc(n);                             // the current best solution

    if (signal(SIGINT, sig_handler) == SIG_ERR) // register signal handler for Ctrl-C
        printf("\nCannot catch SIGINT\n");

    cl0 = clock();
    clock_t clockcapt = cl0 + 300 * CLOCKS_PER_SEC; // 5 min in the future

    char fname[50];
    sprintf(fname, "Thomson-%03i-%02i-%02i.txt", N, MITCHELL, threadcount);
    FILE* fptr = fopen(fname, "r");
    if (fptr) {
        printf("File %s already exists", fname);
        fclose(fptr);
        exit(2);
    }
    fptr = fopen(fname, "w");

    DWORD ThreadId[MAXTHREADS];
    HANDLE ThreadHandle[MAXTHREADS];
    mtxi = CreateMutex(NULL, 0, L"mutexI");
    mtxo = CreateMutex(NULL, 0, L"mutexO");
    if (mtxi == NULL || mtxo == NULL) {
        printf("\x1b[?1049lCreateMutex failed: %d\n", GetLastError());
        exit(-1);
    }

    if (iter >= iterX) {                        // (re)compute 1, print point coordinates
        Optim(NULL);                            // perform optimization with the given iter
        printf("Coordinates in file %s\n", fname);
                                                // write coordinates to file
        fprintf(fptr, " N = %d, M = %d, I = %d, OPT = %#16.15g\n", N, MITCHELL, iter-1, mf*pwc);
        fprintf(fptr, "% 3.14f % 3.14f % 3.14f\n", 1., 0., 0.);
        fprintf(fptr, "% 3.14f % 3.14f % 3.14f\n", sqrt(1.-mx[0]*mx[0]), 0., mx[0]);
        for (int i = 1, j = 2; j < n; i += 2, j += 2) {
            double q = 2.0 / (mx[i]*mx[i] + mx[j]*mx[j] + 1.);
            fprintf(fptr, "% 1.14f % 1.14f % 1.14f\n", q*mx[i], q*mx[j], 1.-q);
        }

        fclose(fptr);                           // clean up
        CloseHandle(mtxi);
```

```
        CloseHandle(mtxo);
        return 0;                               // exit
    }

    for (int i = 0; i < threadcount; ++i) {
        ThreadHandle[i] = CreateThread( // create thread, returns its identifier
            NULL,           // default security attributes
            0,              // default stack size
            Optim,          // thread function
            NULL,           // address of parameters for thread function (int, struct..)
            0,              // default creation flags
            ThreadId + i    // address of the thread function
        );
        if (ThreadHandle[i] == NULL) {
            printf("\x1b[?1049lCreateThread [%d] failed: %d\n", i, GetLastError());
            exit(-2);
        }
    }

    for (; iter <= iterX; ) {
        if (clock() > clockcapt) {              // time to save snapshot to disk
            clockcapt += 300 * CLOCKS_PER_SEC;  // next snapshot in 5 minutes

            WaitForSingleObject(mtxo, INFINITE);// read console buffer
                GetConsoleScreenBufferInfo(hOut, &buff_info);
                SMALL_RECT ReadRegion = {buff_info.srWindow.Left, buff_info.dwCursorPosition.Y - lines+1,
                                         buff_info.srWindow.Left + CONW, buff_info.dwCursorPosition.Y+1 };
                ReadConsoleOutput(hOut, *buff, (COORD){CONH,CONW}, (COORD){0,0}, & ReadRegion);
            ReleaseMutex(mtxo);

            rewind(fptr);                       // console info overwrites file
            for (int r = 0; r <= lines; ++r) {
                if (buff[r][0].Char.AsciiChar == 32) continue;  // skip lines starting with SPACE
                for (int c = 0; c < CONW; ++c)
                    fprintf(fptr, "%c", buff[r][c].Char.AsciiChar);
                fprintf(fptr, "\n");
            }
            fflush(fptr);                       // data to disk from buffer
        }
        Sleep(99);                              // shortest sleep time ~= 16 ms
    }
                                                // wait for all threads to finish (~join)
    WaitForMultipleObjects(threadcount, ThreadHandle, 1, INFINITE);
    for (int i = 0; i < threadcount; ++i)
        CloseHandle(ThreadHandle[i]);           // close the thread handles
    CloseHandle(mtxi);
    CloseHandle(mtxo);

    printf("   ----- stopped ------\n");
    GetConsoleScreenBufferInfo(hOut, &buff_info);
    SMALL_RECT ReadRegion = { buff_info.srWindow.Left, buff_info.dwCursorPosition.Y - lines,
                              buff_info.srWindow.Left + CONW, buff_info.dwCursorPosition.Y };
    ReadConsoleOutput(hOut, *buff, (COORD) { CONH, CONW }, (COORD) { 0, 0 }, & ReadRegion);
    rewind(fptr);
    for (int r = 0; r < lines; ++r) {
        if (buff[r][0].Char.AsciiChar == 32) continue;  // skip lines starting with SPACE
        for (int c = 0; c < CONW; ++c)
            fprintf(fptr, "%c", buff[r][c].Char.AsciiChar);
        fprintf(fptr, "\n");
    }
    fclose(fptr);

    // printf("\nPress ENTER to exit\n"); getchar();
    return 0;
}
```