# SIMION's Simplex optimizer applied to electrode's potential and geometry

**Pierre Chauveau**
Ganil

pierre.chauveau@ganil.fr

Simion user meeting, GANIL, 11/06/2015

## Introduction

This short paper describes the SIMION 8.1 built-in *Nelder-Mead* (or *downhill simplex*) optimization routine and its use to optimize the potentials applied to a given set of electrodes, the geometrical parameters (dimensions, position) of those electrodes or the beam conditions. This method can find a close to optimal solution even if there are several local minima in the optimization goal function, which is likely in many-electrodes systems. A simple system of a 90 electrostatic deflector composed of two bended blades and equipped with two Einzel lenses (Fig.1) will be studied as an example throughout this paper.

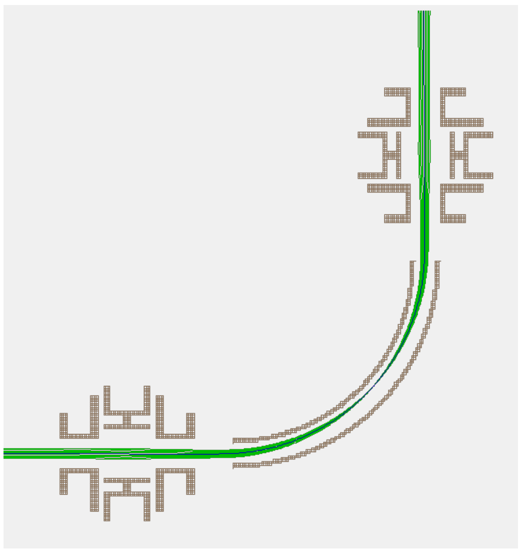
Figure 1: SIMION geometry of the blade-deflector

## The Nelder-Mead algorithm

This algorithm is derived from the Simplex optimization method, the main difference being that Nelder-Mead can be used with non linear problems. This algorithm consists in minimizing the size of a polygon in the $n$-dimensions space of parameters to converge on a certain set of parameters. For $n$ input parameters (the potential of an electrode, the diameter of a rod ...), one should choose $n+1$ sets of parameters $S_1, \ldots, S_{n+1}$, so that the vectors $(S_1S_2, \ldots, S_1S_{n+1})$ form a base of the space of parameters.

***Tip:*** *the SIMION Nelder-Mead optimizer will always require* $2n$ *parameters :* $x_1, \ldots, x_n, \Delta x_1, \ldots, \Delta x_n$ *and construct* $n+1$ *sets* $\{(x_1, x_2, \ldots, x_n), (x_1 + \Delta x_1, x_2, \ldots, x_n), (x_1, x_2 + \Delta x_2, \ldots, x_n), \ldots\}$.

The algorithm follow those steps:

1. Sort $S_0, \ldots, S_{n+1}$ by increasing value of the optimization goal function $f$. $S_{n+1}$ is then the least optimal solution.
2. Calculate $S_0$ the barycenter of $S_1, \ldots, S_n$.
3. Calculate $S_r = S_0 + (S_0 - S_{n+1})$ the reflection of $S_{n+1}$ with respect to $S_0$.
4. If $f(S_r) < f(S_1)$ (best set so far), calculate $S_e = S_0 + 2(S_0 - S_{n+1})$, replace $S_{n+1}$ with the best of $S_r$ and $S_e$ and restart at step 1.
5. Else if $f(S_1) < f(S_r) < f(S_n)$, just replace $S_{n+1}$ with $S_r$ and restart at step 1.
6. Else if $S_r > S_n$, calculate $S_c = S_0 + \frac{1}{2}(S_0 - S_{n+1})$. If $S_c < S_{n+1}$ replace $S_{n+1}$ with $S_c$ and restart at step 1.
7. Else replace all $S_i$, $i > 1$ with $S_1 + \frac{1}{2}(S_i - S_1)$.

***Tip:*** *in SIMION the algorithm stops when all the sets fit within a sphere of a certain radius.*

## Potential optimization

**I. The potentials:** first the user must chose the potentials to optimize. In the case of our deflector we chose the potential of the outer deflecting blade and the potentials of the middle electrode of the two Einzel lenses, all the other electrodes being grounded.

**II. The optimization goal function:** if this function depends on the output beam (transmission, emittance gain, beam position, direction ...), one should import the file *testplanelib.lua* located in *SIMION_directory/examples/test_plane* in the current optimization folder and use it to determine beam characteristics on a stop plane (see first code segment below). In the case of our deflector, we will try to maximize the transmission rate and minimize the angle between the beam direction and the line axis as well as the distance between the beam mean position and the line axis. The goal function will be calculated as $s = A*(1-Tr)^a + B*|\theta_{beam} - \theta_{axis}|^b + C*|X_{beam} - X_{axis}|^c$, with $A$, $a$, $B$, $b$, $C$ and $c$ being coefficient to adjust according to the importance given to each parameter (see second code segment). This part is quite arbitrary and the goal function should therefore be chosen with caution. In our example, the starting potential of the two Einzel lenses and the deflecting blade were 1200V, 1200V and 545V respectively. For this set the value of the goal function was 25.9411 (no physical meaning).

***Tip:*** *in case of ion crash, the position and speed of this ion will be stored in memory the same way as if it reached a stop plane. This can make the goal function rise but also decrease. In the last case, the "optimal solution" in terms of goal function could be to crash a part of the beam or even all of it on the walls. Thus if maximal transmission is wanted, the goal function should never be created in a way that a crashed beam leads to a better result.*

**III. The optimizer:** it has to be declared before the potentials in the *lua* file. *Flym*, *initialize_run* or *init_p_value* (or even *fast_adjust* if all the potentials are time dependent) are viable segments to declare the optimizer. As stated before the optimizer will need a set of potential $V_i$, a set of variations $\Delta V_i$ and a radius of convergence $r_0$ (see third code segment). Most of the time this radius can be chosen small ($\sim 10^{-5} V_i$) as for short times of flight the optimization will be fast and for long times of flight (traps, cyclotrons ...) a high precision will be needed. The optimizer will keep working throughout several runs, until the $n+1$ last sets fits within a sphere of radius $r_0$. At the end of each run (in *terminate_run*) or at the beginning of the next one, the potentials to apply to the electrodes should be changed to those given by the optimizer (still in third code segment). Finally, the optimizer will not make the simulation re-run by itself. Stopping and re-running the simulation as well as feeding the optimizer with the goal function have to be done at each step within the *lua* code.

***Tip:*** *the variables "start", "step" and "minradius" are reserved by the optimizer and should not be used for another purpose (a loop for example) as it might cause the code to crash.*

**IV. Choice of the starting point:** the user should perform a few runs without optimizing to find a working set (without all ions crashing on the walls) and to find out the sensitivity of each parameter. When choosing the set of variations $\Delta V_i$, one should keep in mind that a too small variation will slow down the beginning of the optimization, but a large one could induce a dramatic voltage variation causing the beam to crash on the walls. Generally, a few tries allow us to find a compromise.

***Tip:*** *if the user want to explore a certain parameter in a certain direction (increase/decrease), he can "force" the optimizer to start in this direction by putting a high variation on this parameter and low variations on the others.*

**V. The Input beam:** using a gaussian beam would be nice to optimize our system under realistic conditions. Unfortunately, the gaussian beam generated in SIMION are random beam with a gaussian distribution of probability, which means that each run will have a slightly different input beam. This introduce chance into the optimization and threaten its repeatability. The user can instead generate a gaussian beam and save it to use it in each run or simply define arithmetic sequences for positions/angles of the ions so that the emittance of this beam is roughly the same as for an equivalent gaussian beam.

***Tip:*** *in general, there should be no randomly generated parameters* ***inside*** *the optimization loop. Everything not directly modified by the optimizer should remain unchanged. The only exception would be if the user want to introduce simulated annealing (not described in this paper).*

**VI. Multiple optimizations and randomizing:** with increasing complexity, the number and density of local minima increase as well. The chance of the optimizer converging to a non absolute minimum can be high with 3 or more potentials, especially with small initial variations. The simplest method to find a close to absolute minimum is to finish an optimization, randomize the optimal set within a certain range ($\pm 1\%$ works fine) and to restart a new optimization with these new parameters (see fourth code segment). Here the randomization is done between two optimization and not within one, it does not affects the repeatability of the result. The process can be repeated as many times as needed.

Applied to our example the potential optimization made the goal function drop to 0.8322, with a precision in beam position 6 times better and a precision in beam direction more than 100 times better than before the optimization. The MR-ToF-MS PILGRIM (Fig.2) and its injection has been optimized with this method. The resolving power of this mass spectrometer has vastly increased from a few thousand in the earliest configurations to more than $520.10^3$ nowadays.

## Geometric optimization

All the previous statements also apply to geometry optimization. The major difference is that the voltage change is very fast, while changing geometry requires to build new potential maps, which can be in the range of seconds (simple 2D files) to hours (3D geometry

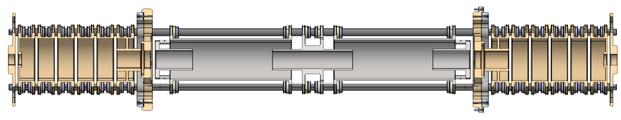

Figure 2: The MR-ToF-MS PILGRIM : a linear ion trap with high resolving power.

with many electrodes or fine mapping). Therefore, it might not be the best solution to optimize potentials and geometric parameters in the same loop as the laters would slow down the formers (the opposite is in principle possible too but is extremely uncommon).

**I. The *gem* file:** the parameters to optimize have to be declared in the header and called within the file. The code shown in the fifth segment generate the two blades of the deflector. In the main *lua* file, a routine must pass these parameters to the *gem* file, generate a potential array (PA) and refine it (see sixth code segment).

***Tip:*** *the user has to make sure he takes advantage of every symmetry available, as each symmetry not stated in the gem file will multiply the refining time by two.*

***Tip:*** *the option surface = fractional should be checked in the potential array declaration in the gem file (see fifth code segment) to be able to deal with lengths which are not an integer multiple of the binning.*

**IIa. Geometry sweep:** a first simple way to make a pseudo optimization is to choose one geometric parameter and to change it following an arithmetic sequence. A full voltage optimization should be made between each change of geometry. At the end of each optimization, the potentials should be reset. The user can sweep each geometric parameter and then, looking at the sensitivity of the goal function to each parameter, can decide which one should be modified to minimize this function. This method uses no proper algorithm (no optimizer) but has been proven effective on a multi-directions deflector developed at GANIL. It is merely a more "user-friendly" way to manually optimize the geometry.

**OR**

**IIb. Dual simplex:** it is possible to use two intricated simplex optimizers on both the potentials and the geometry. For each step of the geometry optimization, one can perform a full potential optimization. This method is slower than the previous one as there are more geometry refines. Nevertheless, this method puts an actual optimizer on the geometry and the results are likely to be better.

**OR**

**IIc. Unique simplex:** we can also apply one simplex algorithm to both the geometry and the potentials. This would probably help to converge faster to a local minimum in terms of iterations, but not necessarily in terms of computation time. One should keep in mind that the refining time induced by a change of geometry can be much longer than the time needed to perform a full potential optimization in the case of a fine 3D geometry. With a unique simplex solution, a new geometry would have to be refined on each step of the optimization, potentially leading to longer computation time than the former dual simplex solution.

**III Comparison:** if a high precision is wanted regardless of the computation time, methods *b* or *c* would be the best choice. It is also possible to apply the randomize method at the end of those optimizations on both the potentials and the geometry. Method *a* is faster, simpler though less precise, and allows the user to "map" the geometric parameter space and possibly find several good configurations, which give us some degrees of freedom for the following engineering phase.

In method *a* and *b*, it should be decided what to do with the potentials after changing the geometry as those can be retained or reset.

- If the potentials are retained, the beam will be less likely to crash on the wall just after the geometry change.
- If the potential are reset, the user can afford to randomize the potential and perform several potential optimization on each geometry as the potential starting point for a new geometry would not depend of a non repeatable last set of potential.

The second solution explores multiple local minima and would be very interesting with the "mapping" behaviour of method *a*, especially for close-to-walls beam for which the algorithm can give very different result depending on if the beam partly crashes on the walls or not. For method *b*, resetting the potentials at each step would only allow us to map the geometric parameter space along the optimization path (highly irregular). Only the potentials of the last geometry should be randomized, which can be done after the end of the geometric optimization.

Methods *b* and *c* have been applied to our example, making the goal function drop again to 0.0001 with a precision in beam direction 1000 times better than with a simple potential optimization. Method *a* has been used to optimize the geometry of a 90-degree 4-ways beam deflector (Fig.3). The goal of this optimization was to bend an ion beam while conserving its properties. The output beam of the optimized deflector was showing deviation of 0.0017°in direction, 0.59 in time-of-flight bunchwidth and 0.10 $\pi$.mm.mrad for an input parallel beam.

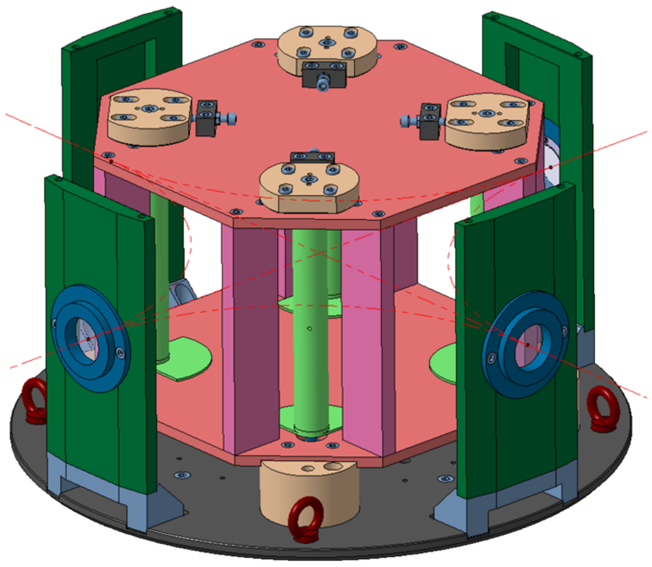

Figure 3: A 4-ways deflector designed to bend the beam with minimal alteration in angle, position and time-of-flight dispersion.

## Conclusion

The Nelder Mead algorithm is a very powerful tool which can save time and enhance a lot the quality of the design in a pre-engineering phase. Finding the absolute minimum is not guaranteed but finding several close-to absolute local minima is. This algorithm can be used to optimize potentials, geometric parameters, beam shape, and pretty much everything that the user can think of, as long as he pays enough attention to the creation of the goal function. Cross optimizations are also possible and give good results. Finally the main drawback of all numerical method, the likeliness to miss the absolute minimum, can be compensated with this algorithm by re-run, randomize then re-run or even simulated annealing.

## LUA Code

## Segment 1: test plane

```
local TP = simion.import 'testplanelib.lua'


-- call the "create" function in the testplanelib.lua file.
-- defining the stop plane of the simulation
local test1 = TP(220,280,0, 0,1,0,
        function()

                -- write tof and position to matrices
                number_of_ions = number_of_ions + 1
                tof[ion_number] = ion_time_of_flight
                px[ion_number] = ion_px_mm
                v[1][ion_number] = ion_vx_mm
                v[2][ion_number] = ion_vy_mm
                dx[ion_number] = 1000.*math.atan(ion_vx_mm/ion_vy_mm)
                ion_splat = 1
end
)


function segment.tstep_adjust()

        test1.tstep_adjust()

end


-- dealing with ion splats on the walls
function segment.other_actions()

        test1.other_actions()
        if ion_splat ~= 0 then
        -- count ions splatted
        ions_splat = ions_splat + 1
                -- if ions splat somewhere in the trap
                if ion_py_mm > 281 or ion_py_mm < 279 then
                        bad_splat = bad_splat + 1
                        print(bad_splat,'splat on mirrors')
                        number_of_ions = number_of_ions + 1
                        -- fill missing values with some numbers
                        tof[ion_number] = 0.
                        px[ion_number] = 0.
                        v[1][ion_number] = 0.
                        v[2][ion_number] = 0.
                        dx[ion_number] = 0.
                end
        end
end
```

## Segment 2: calculating the goal function

```
-- in : function segment.terminate_run()


-- a bit of analysis
means[1] = Stat.array_mean(tof)
means[2] = Stat.array_mean(px)
means[3] = Stat.array_mean(v[1])
means[4] = Stat.array_mean(v[2])
means[5] = Stat.array_mean(dx)
Tr = (number_of_ions-bad_splat)/number_of_ions

-- GOAL function
s = abs(means[2]-EL2x)*10+abs(means[5])*10+(bad_splat/number_of_ions)*100

-- this put the calculated goal function into the optimizer.
optV:result(s)
```

## Segment 3: setting the optimizer and changing the potentials

```
-- called on PA initialization to set voltages.
-- starting/refreshing the potential optimizer
function segment.init_p_values()

    if not optV then  -- is first run (opt not defined)
        -- Create new optimizer using current adjusted voltages.
        optV = SimplexOptimizer {
        start = {V2,V3,V4},
        step = {step_V2,step_V3,step_V4},
        minradius = 1E-1}
    end

    -- Initialize values for this run.
    if Vopt == true then
        V2,V3,V4 = optV:values()
    end

    -- sets electrode voltage
    adj_elect[1] = V1
    adj_elect[2] = V2
    adj_elect[3] = V3
    adj_elect[4] = V4

end
```

## Segment 4: randomizer

```
-- in : function segment.terminate_run()

local Stat = require "simionx.Statistics"
-- randomizer
math.randomseed( os.time() )
local randomtable
randomtable = {}
-- defines a random number between -1 and 1
for i = 1,10 do
        randomtable[i] = math.random(0,2000)/1000 -1
end
-- if the optimization is over randomize the last set
-- and relaunch the optimizer
if optV:running() == false then
        optV = nil
        if nVopt < max_optV then
                nVopt   = nVopt + 1
                V2 = V2 + step_V2*randomtable[1]
                V3 = V3 + step_V3*randomtable[2]
                V4 = V4 + step_V4*randomtable[3]
                optV = SimplexOptimizer{
                start = {V2,V3,V4},
                step = {step_V2,step_V3,step_V4},
                minradius = 1E-1}
        else
                nVopt   = nVopt + 1
        end
end
sim_rerun_flym = 1
-- when set to 1, delete all trajectories from screen and memory
```

## Segment 5: geometry file

```
pa_define(270,285,1,p,surface=fractional, n, e, 1., 1., 1.)


# -- angle of the blades
# local th = _G.th or 90.


Locate(120.,150.,0,1)
 { Elect(1)
   {
    Fill{
           Within {Circle(0.,0., 95, 95) Box(0., 100., 100., -100.)}
           Notin{Circle(0.,0., 92, 92)}
           Notin{locate(0,0,0,1,0,$(th),0){box(0.,120.,120.,-120.)}}
       }
   }
 }

 Locate(120.,150.,0,1)
 { Elect(4)
   {
    Fill{
           Within {Circle(0.,0., 108, 108) Box(0., 110., 110., -110.)}
           Notin{Circle(0.,0., 105, 105)}
```

```
            Notin{locate(0,0,0,1,0,$(th),0){box(0.,120.,120.,-120.)}}
        }
    }
 }
```

## Segment 6: *gem* file reading in *lua* file.

```
-- Loading Geometric Parameters
local function LGP(EL2x,EL2y,th)

  -- convert GEM file to PA# file.
  -- pass variable to GEM file.
  _G.EL2x = EL2x
  _G.EL2y = EL2y
  _G.th = th

  local pa = simion.open_gem('Def2D.gem'):to_pa()
  pa:save('Def2D.pa#')
  pa:close()

  -- reload in workbench
  simion.wb.instances[1].pa:load('Def2D.pa#')

  -- refine PA# file.
  simion.wb.instances[1].pa:refine{convergence=1E-7}


  print("EL2x=" .. EL2x .. ", EL2y=" .. EL2y .. ", th=" .. th)

end
```