nousystems inc
ANALYSIS | SYNTHESIS | INSIGHT

7047 Old Madison Pike, Suite 305
Huntsville, AL 35806
Phone: (256)327-5541

August 3, 2022

TO: Distribution

FROM: David Reynolds

SUBJECT: Particle Swarm Optimization

1 Introduction

The need to solve for the global minimum of a multidimensional function frequently arises in mathematics and engineering. There are many different ways to solve for a minimum of a function, including gradient-based solvers that take small steps in the direction of the negative gradient. There are also symbolic solvers that solve for extrema to find when the gradient is zero. For some applications, the following ways are adequate to solve for the global minimum of a function; however, both of the following methods have shortcomings. If a function is complicated enough it may have many local minimums and would cause the gradient solvers to solve for a local minimum rather than a global minimum. Furthermore, if a function is complicated or not expressible in closed form, it may be impossible to set its derivative equal to zero and solve for the variable and therefore, impossible to find the global minima using symbolic methods.

2 Formulation

In the difficult cases described above, particle swarm optimization (PSO) can be particularly useful. Particle swarm optimization uses intelligent swarms of particles, also known as agents, to identify the global mimium of a function using an iterative method. Every particle has a position x^i, velocity v^i, and a personal best p^i. The swarm has a single position known as the global best p^g that has the "best" or lowest value out of all the particles and is computed with every iteration. The velocity update from one iteration to the next is based on three factors; current velocity, personal best, and global best. This is best shown by the velocity update equation 1 where k is the current time step.

$$v_{k+1}^i = wv_k^i + c_1r_1(p_k^i - x_k^i) + c_2r_2(p_k^g - x_k^i) \tag{1}$$

The w term in the above equation, also known as the inertia term, is a constant that determines how much of an effect the current velocity has on the next velocity. The c_1 term, also known as the cognitive term, is the effect its personal best has on the next velocity. Finally, the c_2 term, also known as the social term, is the effect the swarms best have on the next velocity. The r_1 and r_2 terms are random variables from zero to one and change for every particle and iteration. Once the velocity is known the position update is calculated using equation 2.

$$x^i_{k+1} = x^i_k + v^i_{k+1} \tag{2}$$

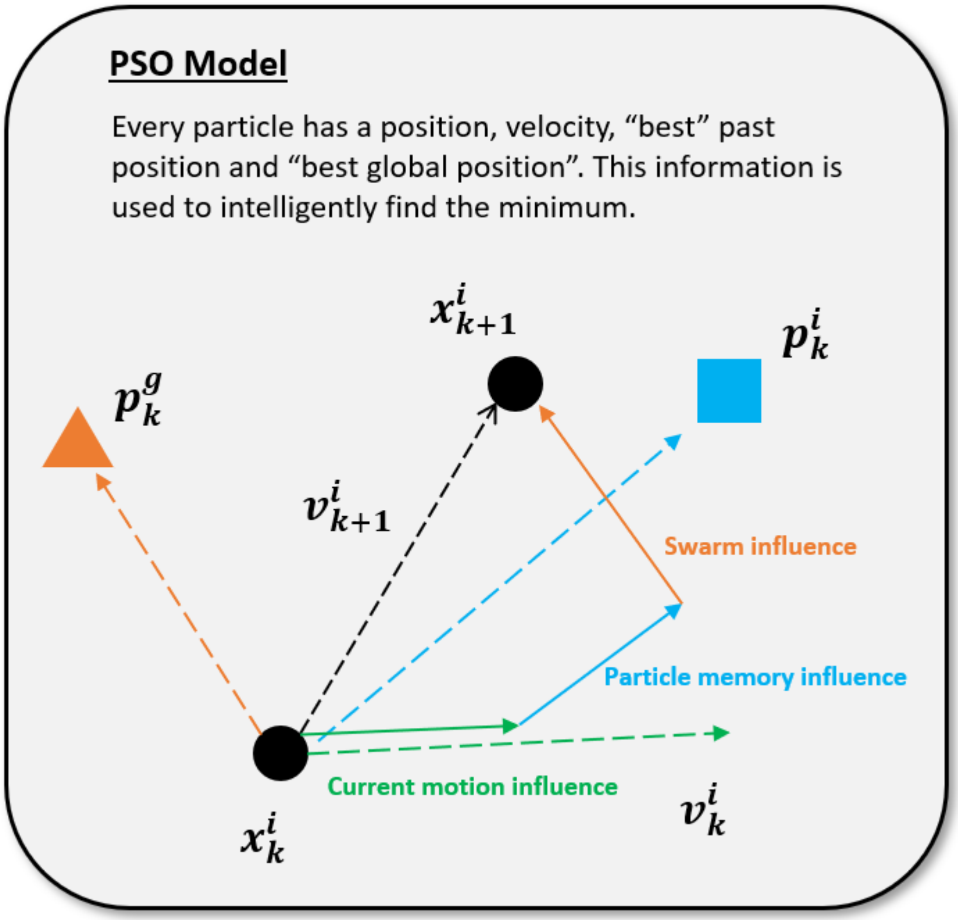

PSO Model
Every particle has a position, velocity, "best" past position and "best global position". This information is used to intelligently find the minimum.
x^i_{k+1}
p^i_k
p^g_k
v^i_{k+1}
Swarm influence
Particle memory influence
Current motion influence
x^i_k
v^i_k

Figure 1: Visualization of position and velocity update formula

The next step is to find some values for w, c_1 and c_2. One way would be to assign them some arbitrary values. However, some values may cause the particle swarm optimizer to be unstable. This is the problem Clerc and Kennedy solved in their paper titled, "The Particle Swarm—Explosion, Stability, and Convergence in a Multidimensional Complex Space". [1] In this paper, they came up with a way to generate values for w, c_1 and c_2 that guarantees stability. To do this let,

$$\phi = \phi_1 + \phi_2 \quad \phi \geq 4 \tag{3}$$

$$\chi = \frac{2\kappa}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|} \quad 1 \geq \kappa \geq 0 \tag{4}$$

Then,

$$w = \chi \tag{5}$$

$$c_1 = \chi\phi_1 \tag{6}$$

$$c_2 = \chi\phi_2 \tag{7}$$

As long as $\phi > 4$ and $0 < \kappa < 1$ the PSO is guaranteed to be stable. The variable kappa can be thought of as how much the PSO searches a space. The higher kappa the more it searches, however the slower it converges. If a search space is highly nonlinear then a kappa closer to 1 is recommended. ϕ_1 is similar to c_1 in that it is how much the social component affects the particles. ϕ_2 is similar to c_2 in how much the cognitive component affects the particles. Once all the constants are initialized, the algorithm follows the flow chart shown below.

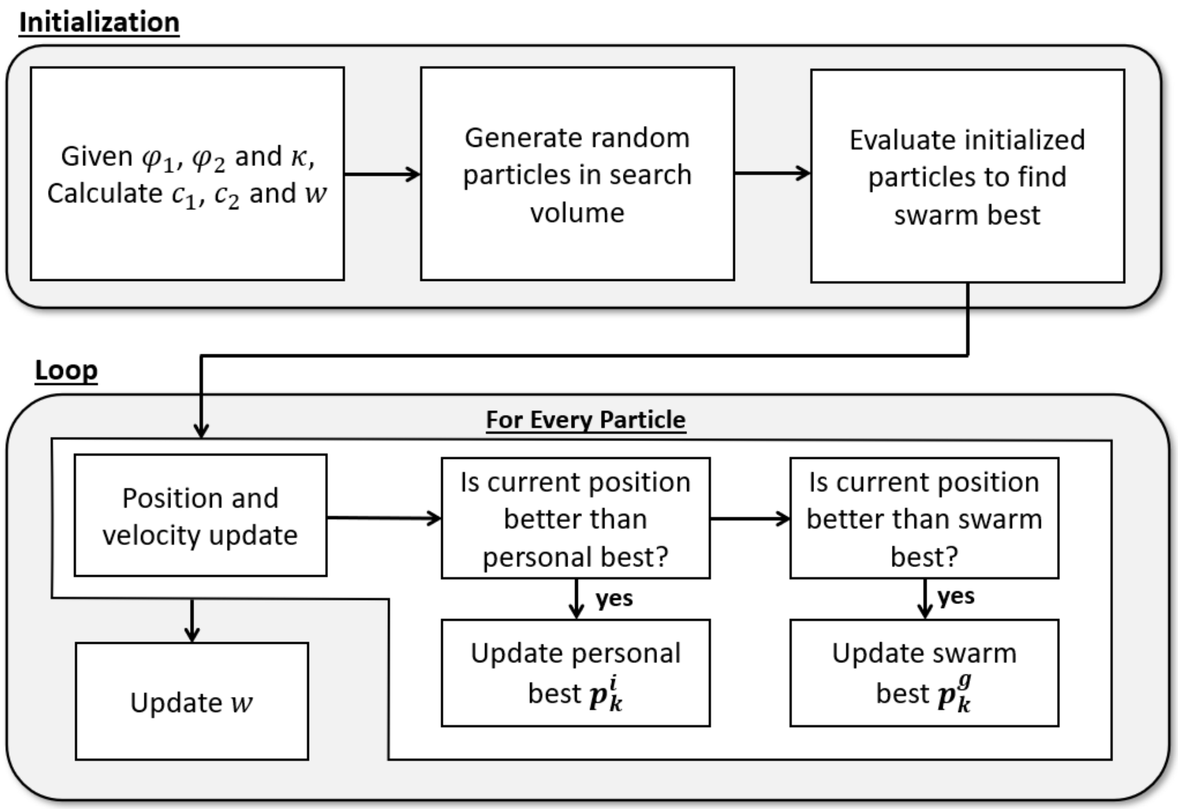

Initialization
Given φ_1, φ_2 and κ, Calculate c_1, c_2 and w
Generate random particles in search volume
Evaluate initialized particles to find swarm best
Loop
For Every Particle
Position and velocity update
Is current position better than personal best?
Is current position better than swarm best?
yes
yes
Update personal best p_k^i
Update swarm best p_k^g
Update w

Figure 2: Flow chart of particle swarm algorithm

One term that is not explained in the flowchart is the w update. As a reminder, w is the inertia term and is how much the current velocity affects the next velocity. The update multiplies the current value of w by a number close, but not equal to, 1 such as 0.985. The reason to update w can be shown experimentally as shown in figure 3.

As shown in figure 3 the PSO with w-damp is approximately 26 orders of magnitude better than with no w-damp in this example. One intuitive way of thinking of the reduction of w is

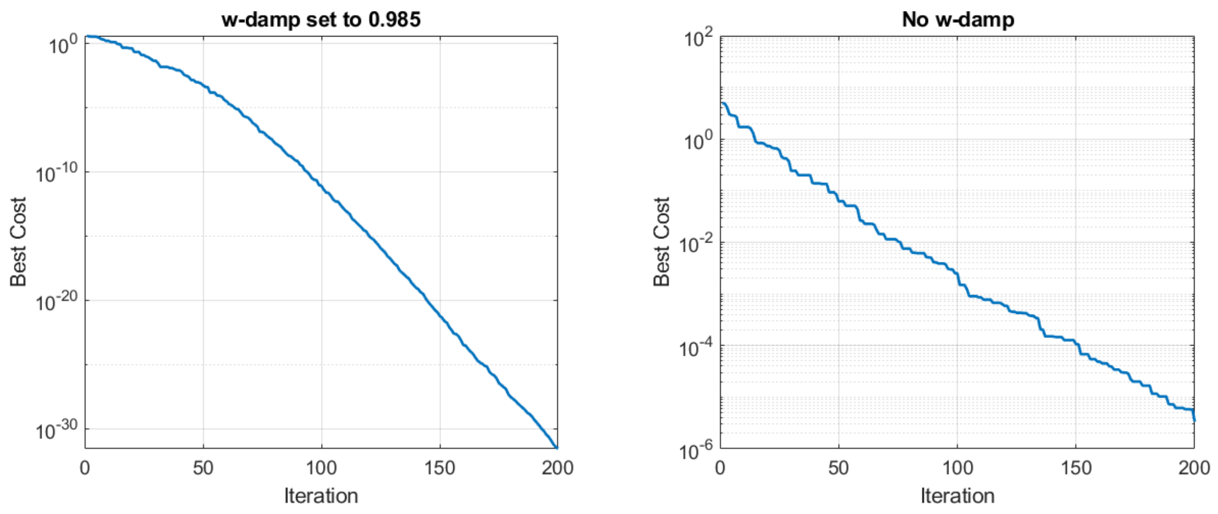


Figure 3: Comparison of the addition of the w damp term with 10-dimensional normal function with 100 particles and 200 iterations. The best cost of the no w-damp is $3.35 \cdot 10^{-6}$ and the best cost with a w-damp set to 0.985 is $2.81 \cdot 10^{-32}$

that it removes some of the "Kinetic Energy" of the particle. As the algorithm iterates, it becomes less reliant on its own velocity and more on its personal and global best. This causes the average velocity of the swarm to decrease and converge to a solution.

3 Examples

3.1 Example 1

As an example, let the function we want to minimize be the normal function defined below where k is the number of dimensions.

$$f(x_k) = \sqrt{\sum_{i=1}^{k} x_i^2} \tag{8}$$

This function can be thought of as the distance from a point to the origin in any number of dimensions. This is a good function to test the PSO because of its simplicity and its trivial solution (x = 0). Furthermore, the dimension of the function can be changed to see how the number of dimensions affects convergence. The results of the PSO show an exponential decrease in error as shown in figure 4.

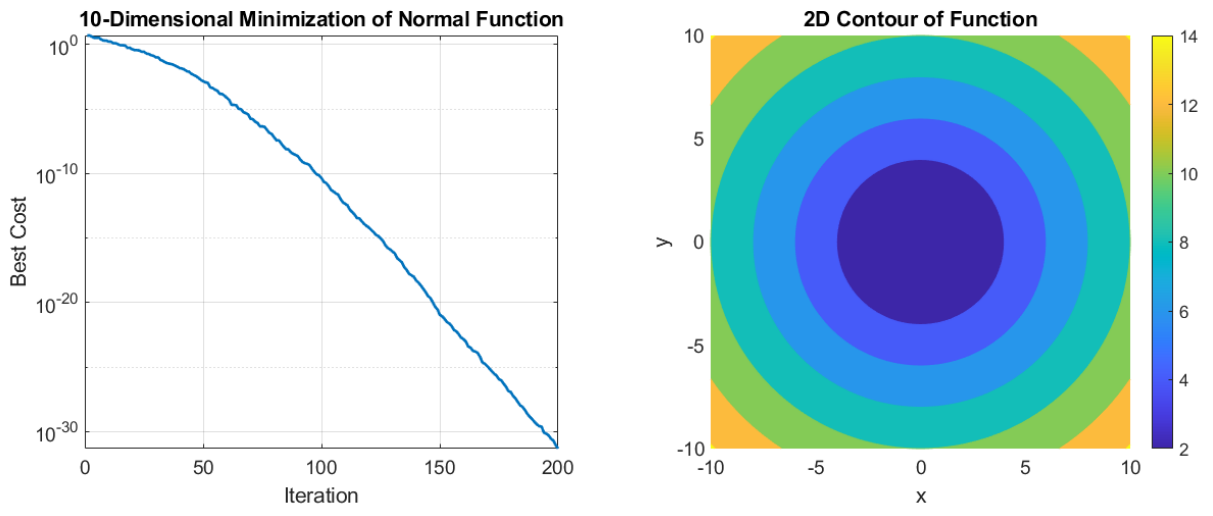


Figure 4: Left plot shows best cost as a function iteration with 100 particles and 200 iterations minimizing 10-Dimensional normal function. Right plot shows 2D contour of normal function

3.2 Example 2

Another benchmark of the PSO is Shaffer's f6 function defined below.

$$f(x,y) = 0.5 + \frac{sin^2(\sqrt{x^2+y^2}) - 0.5}{(1 + 0.001 \cdot (x^2+y^2))^2} \tag{9}$$

The global minimum of this function is located at (0,0); however it may be difficult to solve for this numerically. The PSO, with 100 particles and 100 iterations, finds the exact solution to within MATLAB floating point precision and returns an error of 0 after 60 iterations.

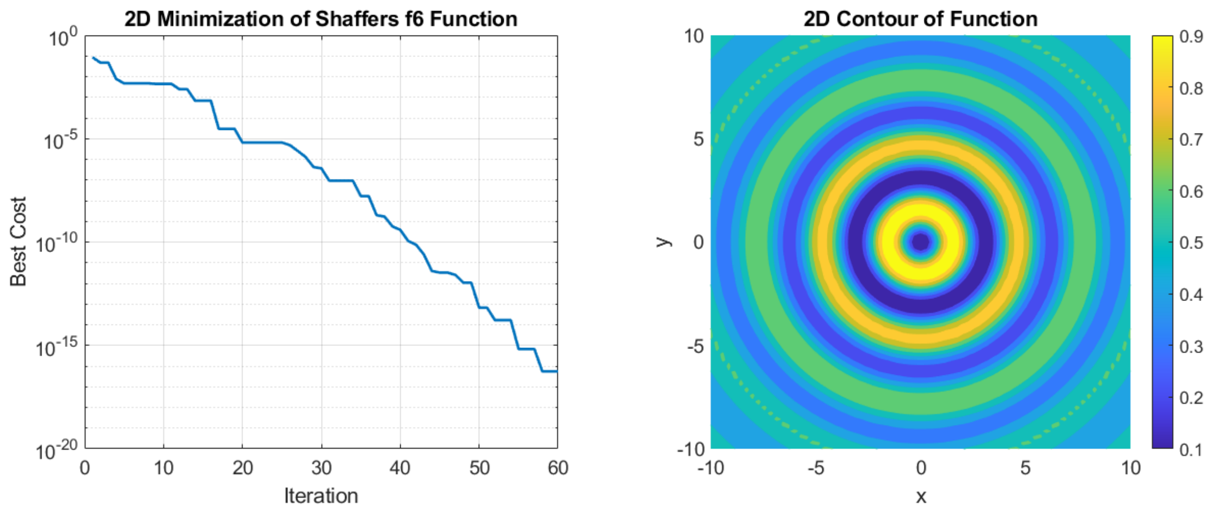


Figure 5: Left plot shows best cost as a function iteration with 100 particles and 100 iterations minimizing 2-Dimensional Shaffer's function. Right plot shows 2D contour of Shaffer's function

3.3 Example 3

In this final example, a more applied problem is conducted. Imagine there are a set of 2D points in a "sensor" reference frame and a "global" reference frame. The only information given is the x and y position of the points from the sensor and the elevation of a pointer, located at the origin of the global reference frame. The objective is to find a rotation matrix and a translation vector that translates from the sensor reference frame to the global reference frame. The transformation is defined by the formula below where $\vec{p'}$ is a 2D point in the sensor reference R is a rotation matrix and $\vec{b}$ is a translation vector.

$$\vec{p} = R\vec{p'} + \vec{b} \tag{10}$$

From the formula above the transformation can be defined by three numbers b_x, b_y and θ where b_x and b_y define a translation vector and θ defines the rotation matrix.

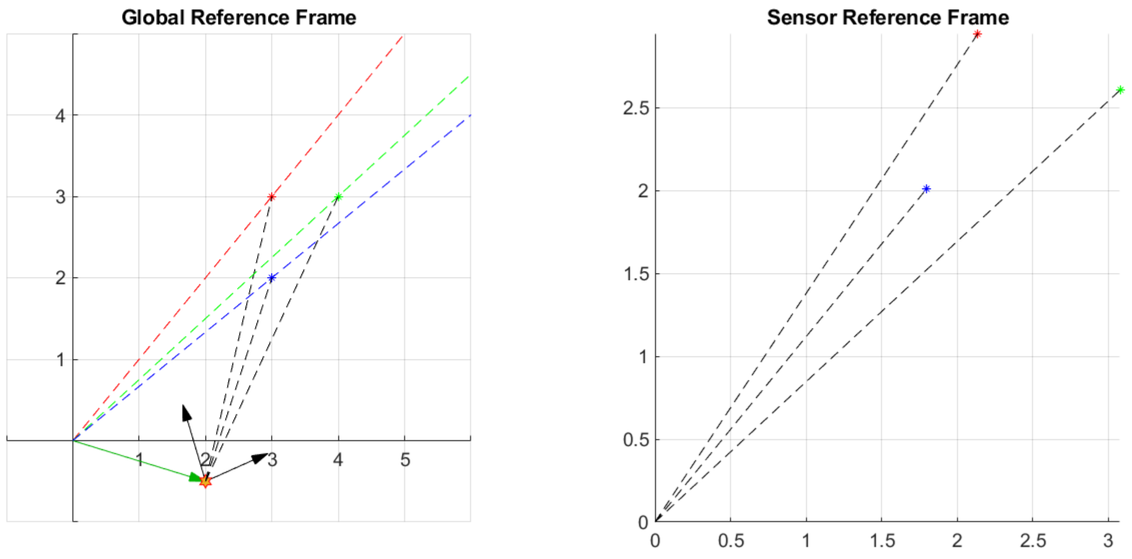


Figure 6: Left plot shows global reference frame and right plot shows sensor reference frame

With three points, as shown in the figure, there is one exact solution. In the example above, that solution is at $x = 2$,$y = \frac{1}{2}$ and $\theta = 20°$. Intuitively, this makes sense because all points align with the dashed line. The dashed line represents the angle the point makes in the global reference frame. For example, the red point (upper point) was measured by the pointer at 45 degrees. This means that the red point must lie along a point that is at a 45 degree angle. The same must be true for all other points. Any other choice of b_x, b_y and θ will make the points lie off the line and is likely not the solution. Therefore, there is a cost associated with a transformed point not being close to the elevation line (dashed line). The associated cost is the average distance of the normal vector from the point to the elevation line. A visual representation of the error can be found in figure 7.

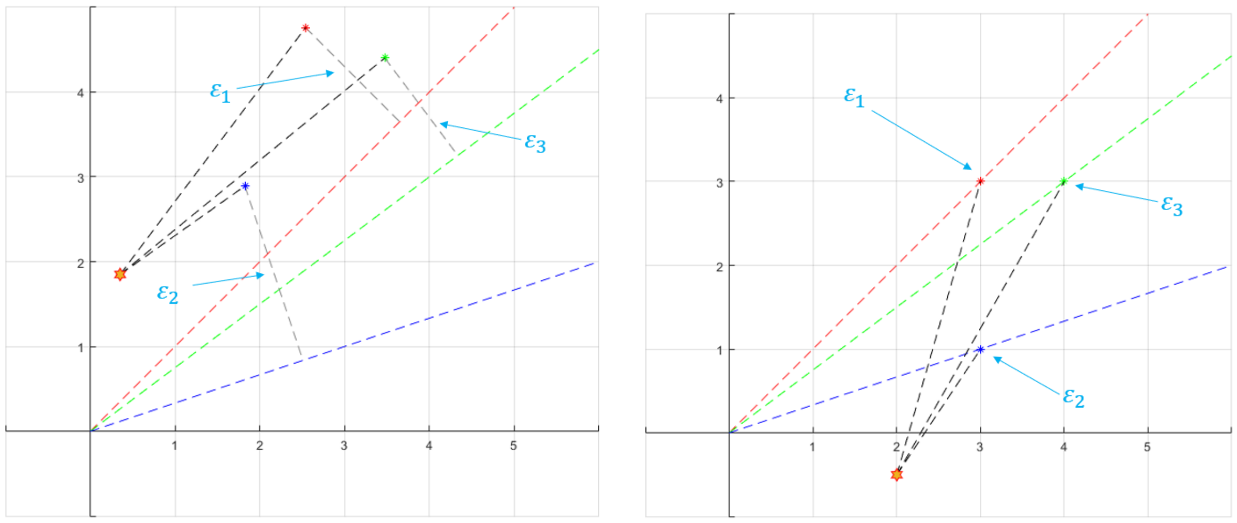
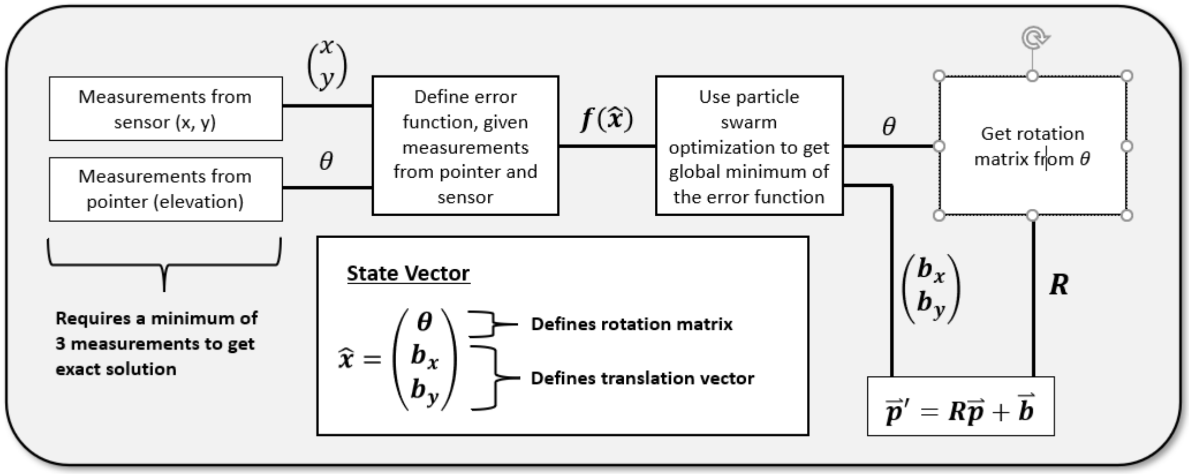


Figure 7: Left plot shows random value of b_x, b_y and θ and has associated cost of approximately 1.74. The plot on right shows a solution of b_x, b_y and θ and associated cost of zero. Both plots are in the global reference frame.

Having a defined cost function, we can now apply the PSO to this problem. The problem of finding a translation vector and rotation matrix has been transformed to a problem of minimizing a 3-dimensional function. When the minimum of the function is found, it represents the transformation from the sensor reference frame to the global reference frame. The full steps to find the rotation matrix and translation vector are shown in the flow chart below.



Measurements from sensor (x, y)
Measurements from pointer (elevation)
Requires a minimum of 3 measurements to get exact solution
$\binom{x}{y}$
θ
Define error function, given measurements from pointer and sensor
$f(\hat{x})$
Use particle swarm optimization to get global minimum of the error function
θ
Get rotation matrix from θ
$\binom{b_x}{b_y}$
R
$\vec{p}' = R\vec{p} + \vec{b}$
State Vector
$\hat{x} = \begin{pmatrix}\theta \\ b_x \\ b_y\end{pmatrix}$
Defines rotation matrix
Defines translation vector

Figure 8: Calibration flow chart

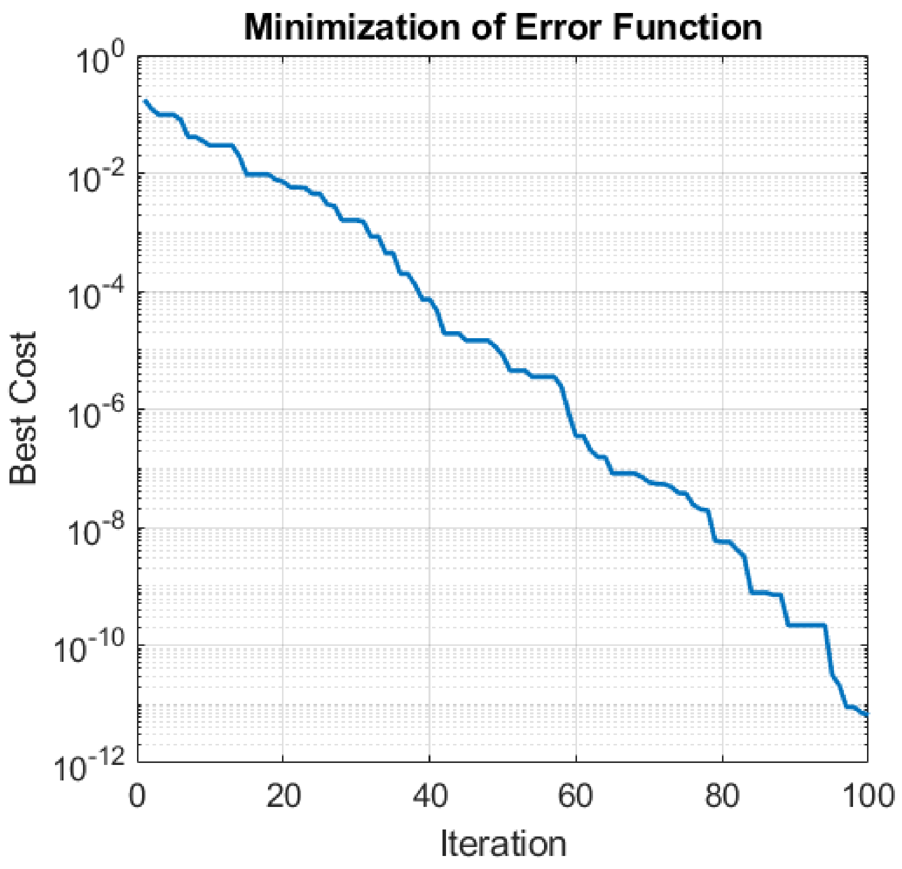


Figure 9: The results of the optimization with 100 particles and 100 iteration had a final cost of $6.55 \cdot 10^{-12}$

The PSO preforms well with an error of $6.55 \cdot 10^{-12}$ after 100 iterations as shown in the figure above. This example was a two dimensional problem with three degrees of freedom; however it would only take slight modifications for the three dimensional case. In the three dimensional case there would be six degrees of freedom; three to define a translation vector and three to define a rotation matrix.

4 Conclusion

This technical memo only scratches the surface of particle swarm optimization. There are many variations of this algorithm that extend its capability. For example, there are gradient-based PSOs that use the gradient as extra information and PSOs that add an acceleration term to the particles. There are particle swarm optimizers that use multiple swarms that each converge to a minimum. These methods can be added to existing optimization algorithms to create a hybrid optimizer. Furthermore, the algorithm can be made more computationally efficient by writing it in C or parallelizing it on GPUs. All of the examples shown in this technical memo and their associated MATLAB code is available in \\nsi.nou-systems.com\noufs\Public\Technical Memos\code\Particle Swarm.

References

[1] Clerc, M., & Kennedy, J. (2002). The particle swarm - explosion, stability, and convergence in a multidimensional complex space. IEEE Transactions on Evolutionary Computation, 6(1), 58–73. https://doi.org/10.1109/4235.985692

[2] Kalami, M. H. [Yarpiz]. (2016, May 22). Particle Swarm Optimization in MATLAB - Yarpiz Video Tutorial [Video]. YouTube. https://youtu.be/sB1n9a9yxJk

[3] Sanyal, S. S. (2021, October 30). An Introduction to Particle Swarm Optimization (PSO) Algorithm. Analytics Vidhya. Retrieved July 18, 2022, from https://www.analyticsvidhya.com/blog/2021/10/an-introduction-to-particle-swarm-optimization-algorithm

5 Appendix

5.1 MATLAB code

```matlab
function out = particleSwarmOptimizerSimple(costFun, nParticles, nIter, lowerBound, upperBound)
    kappa = 1;
    phi2 = 2.05;
    phi1 = 2.05;
    wDamp = .985;

    nVar = length(upperBound);
    varSize = [1, nVar];

    phi = phi1 + phi2;
    chi = 2*kappa/abs(2-phi-sqrt(phi^2-4*phi));
    w = chi;
    c1 = chi*phi1;
    c2 = chi*phi2;

    maxVel = 0.2*(upperBound - lowerBound);
    minVel = -maxVel;

    %%% Initialization
    emptyParticle.pos = [];
    emptyParticle.vel = [];
    emptyParticle.cost = [];
    emptyParticle.best.pos = [];
    emptyParticle.best.cost = [];
    particle = repmat(emptyParticle, nParticles, 1); % Create Population Array

    globalBest.cost = Inf; % Initialize Glabal Best
    for ii=1:nParticles % initalize population members
        particle(ii).pos = unifrand(lowerBound, upperBound, nVar); % Generate Random Solution
        particle(ii).vel = zeros(varSize); % Initialize Velocity
        particle(ii).cost = costFun(particle(ii).pos); % Evaluation

        particle(ii).best.pos = particle(ii).pos; % Update Personal Best Posistion
        particle(ii).best.cost = particle(ii).cost; % Update Personal Best Velocity

        if particle(ii).best.cost < globalBest.cost % Update Global Best
```

```
            globalBest.pos = particle(ii).pos;
            globalBest.cost = particle(ii).cost;
        end
    end
    bestCosts = zeros(nIter, 1); % Array to Hold Best Cost
    bestPos = zeros(nIter, nVar); % Array to Hold Best Posistion
    for iter = 1:nIter
        for ii=1:nParticles
            particle(ii).vel = w*particle(ii).vel... % Update Velocity
                + c1*rand(varSize).*(particle(ii).best.pos - particle(ii).pos)...
                + c2*rand(varSize).*(globalBest.pos - particle(ii).pos);

            particle(ii).pos = particle(ii).pos + particle(ii).vel; % Update Posistion

            particle(ii).cost = costFun(particle(ii).pos); % Evaluation
            if particle(ii).cost < particle(ii).best.cost
                particle(ii).best.pos = particle(ii).pos; % Update Personal Best Posistion
                particle(ii).best.cost = particle(ii).cost; % Update Personal Best Velocity
                if particle(ii).best.cost < globalBest.cost % Update Global Best
                    globalBest = particle(ii).best;
                end
            end
        end
        bestCosts(iter) = globalBest.cost; % Saves Best Cost to Array
        bestPos(iter,:) = globalBest.pos; % Saves Position Cost to Array
        w = w * wDamp; % Sets w
    end
    bestCosts = bestCosts(1:iter-1); % Append Best Cost to Array
    bestPos = bestPos(1:iter-1,:); % Append Best Posistion to Array

    out.pop = particle;
    out.globalBest = globalBest;
    out.bestCost = bestCosts;
    out.bestPos = bestPos;
end
```