**HYBRID SOLUTION APPROACH PARTICLE SWARM OPTIMIZATION AND NEWTON RAPHSON METHOD**

**Subject:** Advance Programming

|  |  |
| --- | --- |
|  | Advisor: Le Thanh Binh  Students Name: Pham Ngoc Huy, Nguyen Thanh Nam  Students ID: 20424004, 20424006 |

**TABLE OF CONTENTS**

[1 TASK DISTRIBUTION 3](#_Toc190696430)

[2 INTRODUCTION 3](#_Toc190696431)

[2.1 Particle Swarm Optimizations (PSO) 3](#_Toc190696432)

[2.1.1 What is PSO? 3](#_Toc190696433)

[2.1.2 Why is PSO needed? What problems does it solve? 3](#_Toc190696434)

[2.1.3 Pros and Cons of PSO: 4](#_Toc190696435)

[2.2 Newton Raphson Method 4](#_Toc190696436)

[2.2.1 What is Newton-Raphson? 4](#_Toc190696437)

[2.2.2 Why is Newton-Raphson needed? What problems does it solve? 4](#_Toc190696438)

[2.3 Hybrid Pso With Newton Raphson 5](#_Toc190696439)

[2.3.1 Why combine Newton-Raphson with PSO? 5](#_Toc190696440)

[2.3.2 Benefits of Hybrid PSO-Newton-Raphson Approach 5](#_Toc190696441)

[2.3.3 Applications of Hybrid PSO-Newton-Raphson 6](#_Toc190696442)

[3 ALGORITHM (MECHANISM) 6](#_Toc190696443)

[3.1 Particle Swarm Optimization 6](#_Toc190696444)

[3.1.1 Mathematical model 6](#_Toc190696445)

[3.1.2 Mechanism 7](#_Toc190696446)

[3.2 Newton Raphson 8](#_Toc190696447)

[3.3 Extend for Hybrid Particle Swarm Optimization and Newton Raphson 9](#_Toc190696448)

[4 CODE IMPLEMENTATION 10](#_Toc190696449)

[4.1 Function Base Class 10](#_Toc190696450)

[4.2 Objective function 10](#_Toc190696451)

[4.3 Particle Representation 10](#_Toc190696452)

[4.4 Particle Swarm Optimization (PSO) 11](#_Toc190696453)

[4.5 Newton Raphson Refinement 12](#_Toc190696454)

[4.6 Hybrid PSO + Newton-Raphson 13](#_Toc190696455)

[5 REFERENCE LIST 13](#_Toc190696456)

# TASK DISTRIBUTION

In this project, I will be the main coder, and Nam will be the reporting, writing and checking the plagiarism. This project aims to get the idea of Particle Swarm Optimization and Newton Raphson Method in optimization problem. In the addition, we can find out the way to collab two of these methods to get the most efficiency in solving problems. There are limitations around the explore and exploit the algorithm in programming with C++, because we are new with this. Some codes and test cases, we were using ChatGPT to provide and fix the error.

# INTRODUCTION

## Particle Swarm Optimizations (PSO)

Optimization is the process of finding the best possible values for specific parameters of a given system to fulfill all design requirements while minimizing cost, time, or other constraints. Many real-world problems, such as engineering design, financial modeling, and machine learning, require efficient optimization techniques. This document discusses the Particle Swarm Optimization (PSO) algorithm, the Newton-Raphson method, and their hybridization to improve performance.

### What is PSO?

Particle Swarm Optimization (PSO) is a population-based solution (global search in optimization term) stochastic optimization technique inspired by the social behavior of bird flocking or fish schooling. Proposed by Kennedy and Eberhart in 1995, PSO optimizes a problem by iteratively improving candidate solutions concerning a given quality measure (GeekforGeeks, 2025).

### Why is PSO needed? What problems does it solve?

PSO is widely used in global optimization problems, including engineering design, machine learning, neural network training, and operational research. It is particularly effective in solving:

* Multimodal optimization problems
* Non-differentiable or discontinuous functions (non-linear and linear programming)
* Large-scale parameter optimization
* Complex search spaces where gradient-based methods struggle, and take long time processing

### Pros and Cons of PSO:

|  |  |
| --- | --- |
| **Pros:** | **Cons:** |
| * Easy to implement and requires minimal parameter tuning * Suitable for high-dimensional and non-convex problem * Can escape local minima due to its stochastic nature * Requires no gradient information | * May converge prematurely to suboptimal solutions * Performance depends on parameter selection (e.g. inertia weight, cognitive, and social components) * Slower convergence for high-precision solutions compared to gradient-based methods |

## Newton Raphson Method

### What is Newton-Raphson?

The Newton Raphson Method, also known as the Newton Method, is one of the most common methods for finding the root of nonlinear univariate functions. Starting with the initial guess of the root, denoted as x0, and iteratively improves this guess using a formula that involves the derivative of the function (GeekforGeeks, 2025).

### Why is Newton-Raphson needed? What problems does it solve?

* **Refining Approximate Solutions:**
  + Optimization algorithms (like Particle **Swarm Optimization, Genetic Algorithms, Gradient Descent**) often get close to an optimal solution but do not refine it precisely.
  + Newton-Raphson **fine-tunes** solutions by making rapid corrections based on the function's slope.
* **Improving Global Optimization (Hybrid Approaches):**
  + **PSO and Genetic Algorithms** are good at **global exploration** but may struggle with fine-tuning.
  + Newton-Raphson **accelerates local convergence**, making hybrid approaches more efficient.

## Hybrid Pso With Newton Raphson

### Why combine Newton-Raphson with PSO?

While PSO is excellent for global search, it suffers from slow convergence in fine-tuning solutions. Newton-Raphson, on the other hand, converges quickly but is highly dependent on a good initial guess and can fail in complex landscapes.

### Benefits of Hybrid PSO-Newton-Raphson Approach

* **Global and Local Optimization synergy**
* PSO efficiently explores the search space to locate a promising region
* Newton-Raphson refines the solution with rapid local convergence
* **Global and Local Optimization synergy**
* PSO efficiently explores the search space to locate a promising region
* Newton-Raphson refines the solution with rapid local convergence
* **Improved Convergence Speed**
* PSO prevents premature convergence to suboptimal solutions.
* Newton-Raphson accelerates convergence once near the optimal solution.
* **Increased Accuracy**
* PSO provides a good starting point for Newton-Raphson, reducing the risk of divergence
* Newton-Raphson ensures high precision in the final optimization step.
* **Robustness**
* The hybrid approach can handle complex, multimodal functions better than either method alone.
* **Applications of Hybrid PSO-Newton-Raphson**
* Electrical power system optimization
* Machine learning hyperparameter tuning
* Engineering design with nonlinear constraints
* Financial modeling with risk constraints

### Applications of Hybrid PSO-Newton-Raphson

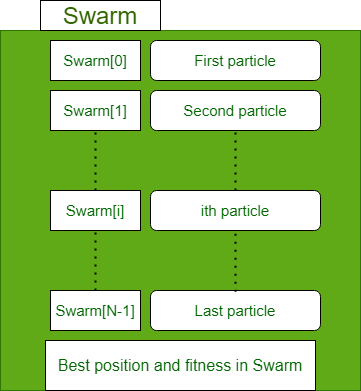
* Electrical power system optimization
* Machine learning hyperparameter tuning
* Engineering design with nonlinear constraints
* Financial modeling with risk constraints

# ALGORITHM (MECHANISM)

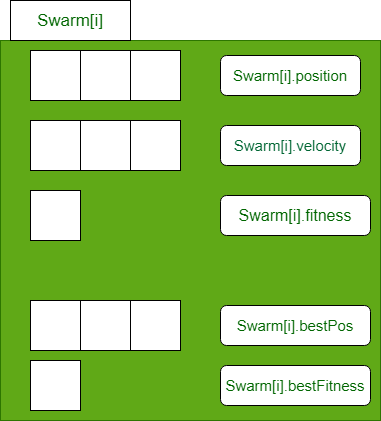
## Particle Swarm Optimization

### Mathematical model

* Each particle in particle swarm optimization has an associated position, velocity, fitness value
* Each particle keeps track of the particle\_best\_Fitness\_value, particle\_bestFitness\_position
* A record of global\_bestFitness\_position and global\_bestFitness\_value is maintained



**Figure 1** Data structure to store Swarm population [1]



**Figure 2** Data structure to store ith particle of Swarm [1]

**Parameters of problem:**

* Number of dimension (d)
* Lower bound (minx)
* Upper bound (maxx)

**Hyperparameters of the algorithm:**

* Number of particles (N)
* Maximum number of iterations (max\_iter)
* Interia (w)
* Cognition of particle (c1)
* Social influence of swarm (c2)

### Mechanism

* **Initialized the Swarm:**
  + Generate an initial population of N particles with random positions and velocities
  + Set the initial best-known position for each particle and the overall best position of the swarm
  + Calculate fitness function, to have the first global best fitness base.
* **Select Hyperparameters:**
  + Define inertia weight , cognitive coefficient , and social coefficient .
* **Iterate Through the Optimization Process:**
  + **Update Velocity:** Adjust the velocity of each particle using the PSO velocity equation.

Where:

|  |  |
| --- | --- |
|  | Velocity of the particle |
|  | Position of the particle |
| and | Random numbers |
|  | Personal best of the particle |
|  | Global best |
| and | Acceleration Coefficients |
|  | Inertia weight |

* + **Update Position:** Move each particle based on its velocity.
  + **Boundary Handling:** If a particle moves outside the defined range, clip its position within allowed limits.
  + **Update Best Positions:** Check if the new position of a particle is better than its previous best. If so, update its best-known position.
  + **Update Global Best:** If any particle achieves a better fitness than the global best, update the global best position.
* **Apply Newton-Raphson Refinement:**
  + Once PSO has identified a promising solution, apply the Newton-Raphson method to refine the result for faster and more precise convergence.
* **Return the Best Solution Found by the Swarm.**

## Newton Raphson

The method approximates the function near a point - using Taylor series expansion:

Where:

is the estimated root of the function

is the value of the equation at  estimated root

is the value of the first order derivative of the equation or function at

To find the root of , we assume that . Solving for , we get the standard Newton-Raphson formula as:

The derivative here we using the Newton forward difference formula:

## Extend for Hybrid Particle Swarm Optimization and Newton Raphson

* **Initialize Swarm**
  + Each particle has a position , velocity , and fitness value
  + Assign random initial values for position and velocity
* **Update Particles** (PSO Update Step)
  + Update velocity (same as PSO above)
  + Update position (same as PSO above)
  + Evaluate fitness and update personal/global best
* **Apply Newton-Raphson for Refinement (on best solutions)**
  + Select top-k particles (e.g., those near )
  + Apply Newton-Raphson to refine (same as Newton-Raphson above)
  + If improvement is significant, update and possibly
* **Repeat Until Convergence**
  + Continue iterating PSO steps with Newton-Raphson refinements until stopping criteria (max iterations, error threshold...) are met.

# CODE IMPLEMENTATION

## Function Base Class

Define an abstract function interface for:

* Evaluating functions values
* Computing numerical derivatives using Newton Forward Difference

|  |
| --- |
| **<Function.h>**  #ifndef *FUNCTION\_H*  #define *FUNCTION\_H*  #include <vector>  #include <functional>  using namespace std;  class *Function* {  protected:  *function*<double(const *vector*<double>&)> func;      double h;  public:  *Function*(*function*<double(const *vector*<double>&)> *f*, double *step\_size* = 1e-6);      virtual double *evaluate*(const *vector*<double>& *x*) const;      virtual double *derivative*(const *vector*<double>& *x*, int *dim*) const;      virtual *~Function*() {}  };  #endif  **<Function.cpp>**  #include "Function.h"  using namespace std;  Function::*Function*(function<double(const vector<double>&)> f, double step\_size)      : *func*(f), *h*(step\_size) {}  double Function::*evaluate*(const *vector*<double>& *x*) const {      return *func*(x);  }  double Function::*derivative*(const *vector*<double>& *x*, int *dim*) const {  *vector*<double> xh = x;      xh[dim] += h;      return (*func*(xh) - *func*(x)) / h;  } |

## Objective function

Defines the specific function to be optimized:

* The default of objective function here is the minimize. In case, if you want to change to the maximize, you can update the “return sum;” into “return -sum”. The maximize result will be behind the “-“<minus operator>.

|  |
| --- |
| **<ObjectiveFunction.h>**  #ifndef *OBJECTIVEFUNCTION\_H*  #define *OBJECTIVEFUNCTION\_H*  #include "Function.h"  class *ObjectiveFunction* : public *Function* {  public:  *ObjectiveFunction*();  };  #endif  **<ObjectiveFunction.cpp>**  #include "ObjectiveFunction.h"  using namespace std;  *ObjectiveFunction*::*ObjectiveFunction*() : *Function*([](const *vector*<double>& *x*) {      double sum = 0.0;      for (double val : *x*)          sum += val \* val;      return sum;  }) {} |

## Particle Representation

Each **Particle** stores:

* **Position** (current solution candidate).
* **Velocity** (update direction).
* **Personal best solution** and fitness score.

|  |
| --- |
| **<Particle.h>**  #ifndef *PARTICLE\_H*  #define *PARTICLE\_H*  #include <vector>  #include "Function.h"  using namespace std;  class *Particle* {  public:  *vector*<double> position;  *vector*<double> velocity;  *vector*<double> bestPos;      double fitness;      double bestFitness;  *Particle*(int *dim*, double *minx*, double *maxx*, const *Function*& *func*);  };  #endif  **<Particle.cpp>**  #include "Particle.h"  #include <cstdlib>  Particle::*Particle*(int dim, double minx, double maxx, const Function& func) {      position.*resize*(dim);      velocity.*resize*(dim);      bestPos.*resize*(dim);      for (int d = 0; d < dim; d++) {          position[d] = minx + ((double) *rand*() / RAND\_MAX) \* (maxx - minx);          velocity[d] = ((double) *rand*() / RAND\_MAX) \* (maxx - minx) \* 0.1;          bestPos[d] = position[d];      }      fitness = func.*evaluate*(position);      bestFitness = fitness;  } |

## Particle Swarm Optimization (PSO)

Implements global search by updating particles’ velocity and positions.

|  |
| --- |
| **<pso.h>**  #ifndef *PSO\_H*  #define *PSO\_H*  #include <vector>  #include "Particle.h"  using namespace std;  class *PSO* {  protected:      int N, DIM, max\_iter;      double minx, maxx, w, c1, c2;  *vector*<*Particle*> swarm;  *vector*<double> bestPosSwarm;      double bestFitnessSwarm;      const *Function*& func;      void *initializeSwarm*();  public:  *PSO*(int *n*, int *dim*, int *iter*, double *minX*, double *maxX*, double *inertia*, double *cognitive*, double *social*, const *Function*& *function*);  *vector*<double> *optimize*();  };  #endif  **<PSO.cpp>**  #include "pso.h"  #include <iostream>  #include <limits>  #include <cstdlib>  using namespace std;  PSO::*PSO*(int n, int dim, int iter, double minX, double maxX, double inertia, double cognitive, double social, const Function& function)      : *N*(n), *DIM*(dim), *max\_iter*(iter), *minx*(minX), *maxx*(maxX), *w*(inertia), *c1*(cognitive), *c2*(social), *func*(function) {      bestPosSwarm.*resize*(DIM);  }  void PSO::*initializeSwarm*() {  *srand*(*time*(0));      bestFitnessSwarm = numeric\_limits<double>::*max*();      for (int i = 0; i < N; i++) {  *Particle* *p*(DIM, minx, maxx, func);          if (p.fitness < bestFitnessSwarm) {              bestPosSwarm = p.position;              bestFitnessSwarm = p.fitness;          }          swarm.*push\_back*(p);      }  }  *vector*<double> PSO::*optimize*() {  *initializeSwarm*();      for (int iter = 0; iter < max\_iter; iter++) {          for (auto& particle : swarm) {              for (size\_t d = 0; d < particle.position.*size*(); d++) {                  double r1 = (double) *rand*() / RAND\_MAX;                  double r2 = (double) *rand*() / RAND\_MAX;                  particle.velocity[d] = w \* particle.velocity[d] +                                         c1 \* r1 \* (particle.bestPos[d] - particle.position[d]) +                                         c2 \* r2 \* (bestPosSwarm[d] - particle.position[d]);                  particle.position[d] += particle.velocity[d];              }              particle.fitness = func.*evaluate*(particle.position);              if (particle.fitness < particle.bestFitness) {                  particle.bestFitness = particle.fitness;                  particle.bestPos = particle.position;              }              if (particle.fitness < bestFitnessSwarm) {                  bestFitnessSwarm = particle.fitness;                  bestPosSwarm = particle.position;              }          }          cout << "Iteration: " << iter << " Best fitness: " << bestFitnessSwarm << endl;      }      return bestPosSwarm;  } |

## Newton Raphson

## Newton-Raphson to find Root:

* The default epsilon is 1e-6 and the iteration for the loop of the newton Raphson will be 100. If you want to customize, you can change it also.

|  |
| --- |
| **<newtonraphson.h>**  #ifndef *NEWTONRAPHSON\_H*  #define *NEWTONRAPHSON\_H*  #include "Function.h"  using namespace std;  class *NewtonRaphsonCalculation* {  protected:  *Function*& func;      double epsilon;  *vector*<double> xn;  public:  *NewtonRaphsonCalculation*(*Function*& *f*, const *vector*<double>& *initial\_guess*, double *epsilon\_*);  *vector*<double> *findRoot*();  };  #endif  **<newtonraphson.cpp>**  #include "newtonraphson.h"  #include <cmath>  #include <iostream>  using namespace std;  NewtonRaphsonCalculation::*NewtonRaphsonCalculation*(Function& f, const vector<double>& initial\_guess, double epsilon\_)      : *func*(f), *xn*(initial\_guess), *epsilon*(epsilon\_) {}  *vector*<double> NewtonRaphsonCalculation::*findRoot*() {  *vector*<double> xn1 = xn;      bool converged = false;      int max\_iterations = 100;      for (int iter = 0; iter < max\_iterations && !converged; iter++) {          converged = true;          for (size\_t i = 0; i < xn.*size*(); i++) {              double fx = func.*evaluate*(xn);              double dfx = func.*derivative*(xn, i);              if (*abs*(dfx) < 1e-6) continue;              double new\_xi = xn[i] - fx / dfx;              if (*abs*(new\_xi - xn[i]) > epsilon)                  converged = false;              xn1[i] = new\_xi;          }          xn = xn1;      }      return xn1;  } |

## Hybrid PSO + Newton-Raphson

Combines **PSO global search** and **Newton-Raphson local refinement**.

|  |
| --- |
| **<hybridPSONewtonRaphson.h>**  #ifndef *HYBRID\_PSO\_NEWTON\_H*  #define *HYBRID\_PSO\_NEWTON\_H*  #include "pso.h"  #include "newtonraphson.h"  using namespace std;  class *HybridPSONewton* {  private:  *PSO* pso;  *Function*& func;      double epsilon;  public:  *HybridPSONewton*(int *psoParticles*, int *dimensions*, int *psoIterations*, double *minX*, double *maxX*,                      double *inertia*, double *cognitive*, double *social*, *Function*& *function*, double *newtonEpsilon*);  *vector*<double> *optimize*();  };  #endif  **<hybridPSONewtonRaphson.cpp>**  #include "hybridPSONewtonRaphson.h"  #include <iostream>  using namespace std;  HybridPSONewton::*HybridPSONewton*(int psoParticles, int dimensions, int psoIterations, double minX, double maxX,                                   double inertia, double cognitive, double social, Function& function, double newtonEpsilon)      : *pso*(psoParticles, dimensions, psoIterations, minX, maxX, inertia, cognitive, social, function),  *func*(function), *epsilon*(newtonEpsilon) {}  *vector*<double> HybridPSONewton::*optimize*() {  *vector*<double> bestPosition = pso.*optimize*();      double bestValue = func.*evaluate*(bestPosition);      cout << "PSO Best Fitness: " << bestValue << " at Position: ";      for (double val : bestPosition) cout << val << " ";      cout << endl;  *NewtonRaphsonCalculation* *newtonSolver*(func, bestPosition, epsilon);  *vector*<double> refinedPosition = newtonSolver.*findRoot*();      double refinedValue = func.*evaluate*(refinedPosition);      cout << "Newton-Raphson Refined Value: " << refinedValue << " at Position: ";      for (double val : refinedPosition) cout << val << " ";      cout << endl;      return refinedPosition;  } |

## Main file for testing

I create this main file base on the recommendation of GPT for testing, because it provide for all class and function

|  |
| --- |
| **<main.cpp>**  #include <iostream>  #include "ObjectiveFunction.h"  #include "pso.h"  #include "newtonraphson.h"  #include "hybridPSONewtonRaphson.h"  using namespace std;  int *main*() {  *// Define objective function*  *ObjectiveFunction* objFunc;  *// Test PSO Only*      cout *<<* "=== Running PSO Only ===" *<<* *endl*;  *PSO* *pso*(10, 2, 100, -5, 5, 0.5, 1.5, 1.5, objFunc);  *vector*<double> pso\_result = pso.*optimize*();  *// Test Newton-Raphson Only (with random initial guess)*      cout *<<* "\n=== Running Newton-Raphson Only ===" *<<* *endl*;  *vector*<double> initial\_guess = {2.0, -3.0}; *// Example starting point*  *NewtonRaphsonCalculation* *newtonSolver*(objFunc, initial\_guess, 1e-6);  *vector*<double> newton\_result = newtonSolver.*findRoot*();  *// Test Hybrid PSO + Newton-Raphson*      cout *<<* "\n=== Running Hybrid PSO + Newton-Raphson ===" *<<* *endl*;  *HybridPSONewton* *hybrid*(10, 2, 100, -5, 5, 0.5, 1.5, 1.5, objFunc, 1e-6);  *vector*<double> hybrid\_result = hybrid.*optimize*();      return 0;  } |

# REFERENCE LIST

1. GeeksforGeeks (2025) *Particle Swarm Optimization (PSO) – An Overview*. Available at:<https://www.geeksforgeeks.org/particle-swarm-optimization-pso-an-overview/> (Accessed: 17 February 2025).
2. GeeksforGeeks (2025) *Newton-Raphson Method*. Available at:<https://www.geeksforgeeks.org/newton-raphson-method/> (Accessed: 17 February 2025).
3. Oh, R., Shi, Y., and Choi, J.W. (2021) 'A Hybrid Newton-Raphson and Particle Swarm Optimization Method for Target Motion Analysis by Batch Processing', Sensors, 21(6), p. 2033. doi: 10.3390/s21062033.
4. TRAN THI CAM GIANG (2020) *Cài đặt Giải Thuật Tối Ưu Bày Đàn (PSO) bằng C++ - Đạt 60PM1 Tlu*. YouTube video, 26 October 2020. Available at: <https://www.youtube.com/watch?v=nEn7fuCYQPg&t=30s>  (Accessed: 17 February 2025).
5. Ritika xRay Pixy (2020) *Particle Swarm Optimization (PSO) Algorithm Example Step-by-Step Explanation ~xRay Pixy*. YouTube video, 10 October 2020. Available at: <https://www.youtube.com/watch?v=HmDjfL3R39M&list=PLVLAu9B7VtkalMRDfFRT_Bzf45VE3OGBF>  (Accessed: 17 February 2025).
6. Code by Pixy (2020) *PSO (Particle Swarm Optimization) Example Step-by-Step*. Available at:<https://codebypixy.blogspot.com/2020/10/pso-particle-swarm-optimization-example.html> (Accessed: 17 February 2025).