0.1 Demo Analysis

Table 1: Parameter settings

Parameter	ω_{min}	ω_{max}	$c_1 = c_2$	steps	`clamp_pos`	`nhood_size`
Value	0.7	0.3	1.496	100,000	periodic	5

The present value for ω_{min} is set as `PSO_INERTIA` or 0.7. The strategy implemented is `PSO_W_LIN_DEC`, which decreases the inertia weight. The settings control the degree of descent. `nhood_size` denotes the number of informers for each particle, which in this case is 5.

The demo was primarily analysed to examine the degree of premature convergence present in the algorithm itself. The tests were informed by the work on previous examinations of these specific benchmark functions [**?**]. The program was executed for the purpose of testing using the `test.sh` bash file. Ten runs of each algorithm were performed for data collection. The `output.dat` files obtained were converted to `*.csv` for analysis.

0.1.1 Search ranges

Table 2: Search ranges for each function

Function	Range
Ackley	$-32.8, 32.8$
Sphere	$-100, 100$
Rosenbrock	$-2.048, 2.048$
Griewank	$-600, 600$

The Ackley function has a global minimum of $f = 0$ where $x = (0, 0, ..., 0)$ though it has many minor local minima.

0.1.2 Analysis

The first element will be to check the error achieved against the number of iterations of the algorithm required to achieve the desired margin of error. The second element of this analysis is to check the desired fitness level against the different levels actually achieved across different dimensions, `dim`. The mean fitness is the figure to look for. Note, mean time is quoted out of 25 runs. The standard deviation is the standard deviation between the mean values for each run, as explained below.

- **Best fitness:** This is the best fitness solution achieved by each benchmark function as represented by the minimum error level across all runs.

- **Mean fitness:** This is the average figure for the fitness achieved over all runs by each benchmark function.
- **Poorest fitness:** The least best fitness level achieved across all runs by each benchmark function.
- **Standard deviation:** As mentioned, this is the standard deviation between the mean levels of fitness achieved in each individual run, aggregated for all runs.
- **Mean time:** The average time taken for each run by each benchmark function.

Table 3: Results for benchmark functions with dimensions $N = 20$, size = 18.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**2.6375E+00**	**1.2000E+01**	**0.0000E+00**	**4.1120E-01**
Mean fitness	**5.3458E+00**	1.2620E+03	**7.0764E+02**	**1.0834E+01**
Poorest fitness	**1.9595E+01**	4.4634E+04	**5.3914E+03**	**4.3180E+02**
Standard deviation	**1.1635E-01**	**4.0254E+01**	1.5574E+02	**1.022E+00**
Mean time (ms)	120.40	71.20	108.40	130.80

Table 4: Results for benchmark functions with dimensions $N = 30$, size = 20.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**3.276E+00**	2.6000E+01	**0.0000E+00**	**6.573E-01**
Mean fitness	**5.8233E+00**	**1.6646E+03**	3.1518E+03	**1.4016E+01**
Poorest fitness	**1.9640E+01**	7.9735E+04	**1.0195E+04**	**7.1847E+02**
Standard deviation	**1.7158E-01**	**1.4613E+02**	3.2299E+02	**7.6188E-01**
Mean time (ms)	270.40	160.00	240.80	290.40

Table 5: Results for benchmark functions with dimensions $N = 50$, size = 24.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**4.0217E+00**	**7.8000E+01**	3.8887E+03	**8.9071E-01**
Mean fitness	**6.931E+00**	**2.6504E+03**	1.4254E+04	**1.8769E+01**
Poorest fitness	**1.9800E+01**	1.4625E+05	**1.8408E+04**	**1.245E+03**
Standard deviation	**2.9957E-01**	**1.4614E+02**	1.9429E+03	**1.1489E+00**
Mean time (ms)	741.20	434.40	696.40	785.60

Table 6: Results for benchmark functions with dimensions $N = 70$, size = 26.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**4.9272E+00**	**1.4500E+02**	1.9924E+04	**9.9511E-01**
Mean fitness	**7.9936E+00**	**3.7159E+03**	2.4221E+04	**2.5144E+01**
Poorest fitness	**1.9873E+01**	2.0908E+05	**2.7458E+04**	**1.8833E+03**
Standard deviation	**2.8402E-01**	**2.4331E+02**	1.9516E+03	**1.2648E+00**
Mean time (ms)	1445.20	851.60	1394.80	1538.80

Table 7: Results for benchmark functions with dimensions $N = 100$, size = 30.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**6.0542E+00**	**4.2000E+02**	**2.9383E+04**	**1.1134E+00**
Mean fitness	**9.4993E+00**	6.0538E+03	3.5019E+04	**3.2981E+01**
Poorest fitness	**1.9894E+01**	**3.0353E+05**	**4.5821E+04**	**2.7054E+03**
Standard deviation	**4.1190E-01**	4.3564E+02	3.6994E+03	**1.3156E+00**
Mean time (ms)	2908.00	1819.60	2831.20	3194.00

Table 8: Results for benchmark functions with dimensions $N = 500$, size = 54.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**1.4433E+01**	5.7015E+04	1.9829E+05	**4.8773E+01**
Mean fitness	**1.6645E+01**	1.3874E+05	2.1792E+05	**3.0480E+02**
Poorest fitness	**1.9972E+01**	1.5935E+06	2.2998E+05	1.4501E+04
Standard deviation	**6.1797E-01**	2.7717E+04	6.9748E+03	**7.1154E+01**
Mean time (ms)	147818.00	95558.40	74689.60	134804.80

Table 9: Results for benchmark functions with dimensions $N = 1000$, size = 73.

Function	Ackley	Sphere	Rosenbrock	Griewank
Best fitness	**1.7250E+01**	3.2513E+05	4.3102E+05	**7.3685E+02**
Mean fitness	**1.8593E+01**	5.7175E+05	4.5025E+05	1.6362E+03
Poorest fitness	**1.9993E+01**	3.2470E+06	4.6860E+05	2.9311E+04
Standard deviation	**3.2277E-01**	1.1247E+05	1.0289E+04	**3.2838E+02**
Mean time (ms)	560022.80	419196.80	299853.20	590098.00

Figure 1: Standard deviations of the means illustrate the break in consistency with the Rosenbrock benchmark.

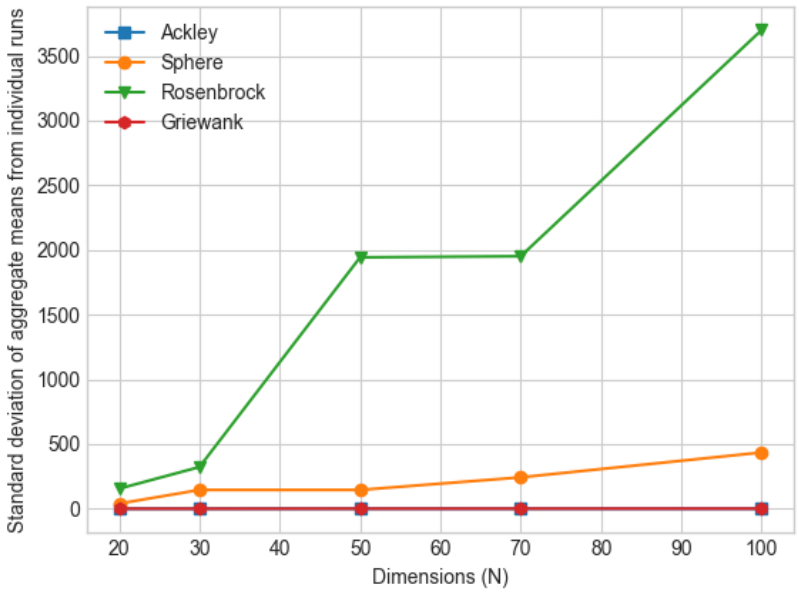
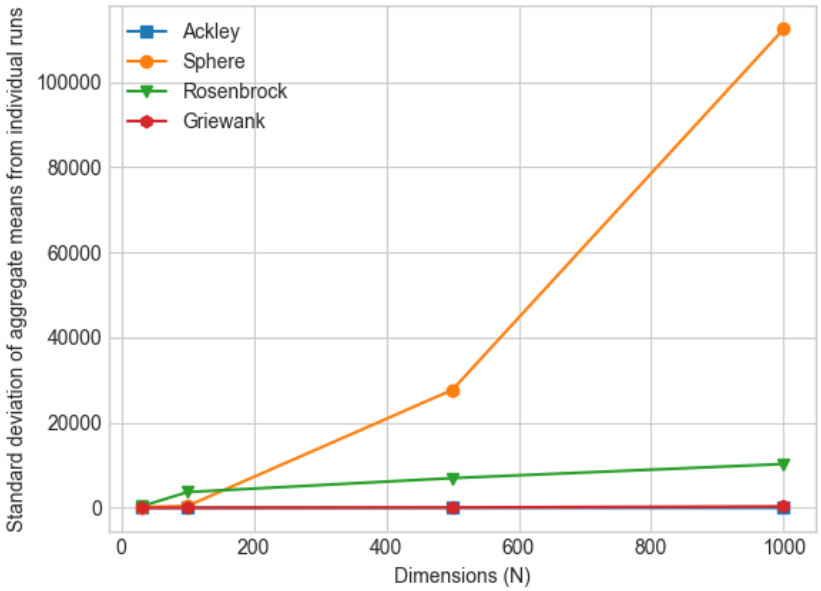
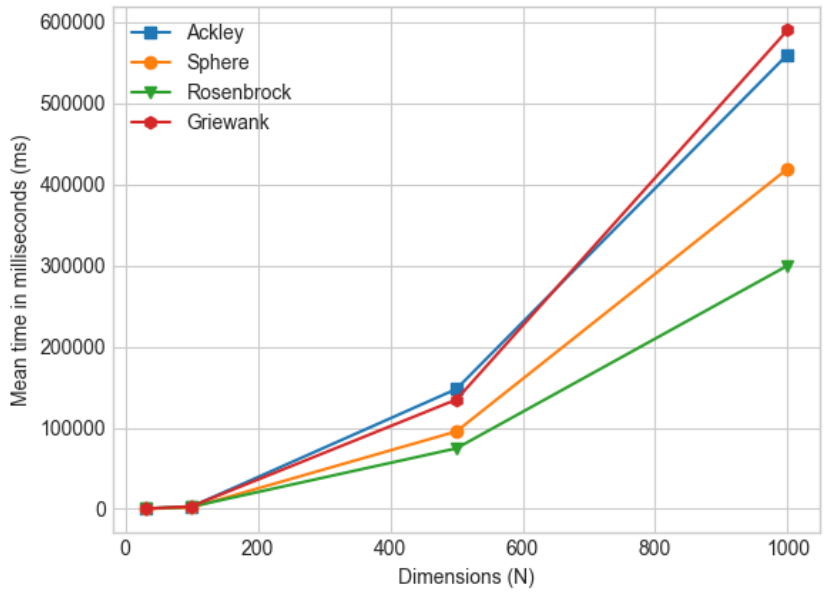


Figure 2: Standard deviations of the means up to $N = 1000$.

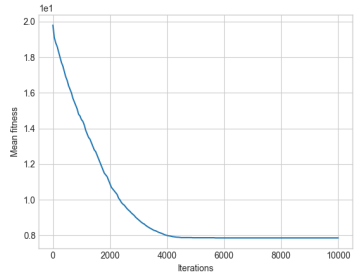


0.1.3 Timing Graphs

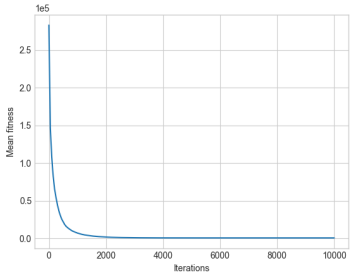
Figure 3: Average time per run for each function.



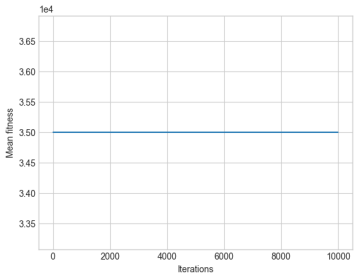
0.1.4 Convergence Graphs



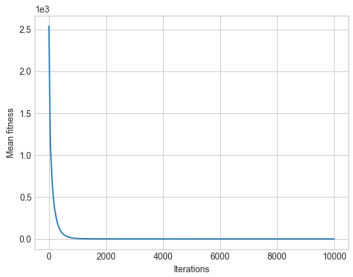
(a) Ackley



(b) Sphere



(c) Rosenbrock



(d) Griewank

Figure 4: Convergence graphs for each function at $N = 100$ dimensions.

0.2 Discussion of observations

0.2.1 Ackley

In terms of timing, Ackley is fine with 20 and 30 dimensions, but then begins to decay, coming second only to Griewank. That being said, the mean fitness levels achieved are consistently the best out of the benchmark set and even the poorer fitness levels cannot compare with even the better Rosenbrock or Sphere ones in the higher dimensions. The standard deviation between the mean best fitness levels achieved per run (see Figure 1) are also comparatively the lowest.

0.2.2 Sphere

The Sphere function is dependable in lower dimensions and doesn't decay too much in terms of time. This function does however achieve the poorest fitness levels once N is greater than 30. Whilst the means achieved do not deviate as much as those achieved by the Rosenbrock function, it is nonetheless prone to break in terms of deviations earlier.

0.2.3 Rosenbrock

Rosenbrock is not well behaved in the higher dimensions in spite of its better performance measures at the lower dimensions. A quick median fitness analysis for Rosenbrock at $N = 30$ is 0.77712, so it converges very quickly in lower dimensions. Time decay takes hold when $N = 70$, and so it does not achieve the correct fitness levels afterwards.

0.2.4 Griewank

Griewank decayed the most in terms of timing. Less deviation between the average time of each run for each function due to higher computational load and the sort mechanism in memory is the case here, since the function, whilst able to achieve a consistent fitness level, can also encounter much poorer fitness levels when in the process of evaluating.

Remark. *The next step will be to run these same tests in parallel to see what the performance improvement is by adjusting the number of processors. With the path planning problem, the serial evaluation will look at how increasing the number of particles affects performance, and thus with the parallel evaluation of that application, the same assessment will take place, but with the spread of N particles across multiple processors. The appendix will be further developed to accommodate more graphical analysis that cannot be included in the main body of the report.*

0.3 Increment and Problem Dimensionality

```
1 //Calculate swarm size based on dimensionality
2 int pso_calc_swarm_size(int dim) {
3   int size = 10. + 2. * sqrt(dim);
4   return (size > PSO_MAX_SIZE ? PSO_MAX_SIZE : size);
5 }
```

Listing 1: Function for calculating appropriate swarm size

All of the demo runs were performed with swarm sizes that were less than 100, with the highest, $N = 1000$ containing a swarm size of 73. An automatic calculation of the swarm size was performed by the function shown above, which was used in each case. The classical thinking on the population size is that the best performance typically occurs in populations with smaller sizes, hence the restricted allowance for incrementation with the problem dimensions. However, it has been argued recently that the smaller sizes selected to evaluated different optimisation problems may indeed be too small [**?**].

The authors of this paper have argued that the best results are typically obtained in the 70-500 size range. This is not a problem for the results obtained above as the range of sizes is incremented sufficiently, and evaluation of the impact of higher dimensions versus the increased population size is beyond the scope of this analysis. Nonetheless, it is evident that the impact of higher dimensions likely outweighs any influence a population size beyond 100 would have, as can be seen by the breakdown in performance of the Rosenbrock benchmark.

0.4 Profiling

Profile was conducted using a mixture of Vampir, Score-P, and `gprof`, the latter of which was used for both the demo code and serial application. An example of how the profiling was run is given as

```
gprof -b ./prog ackley > analysis.txt
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 80.75      0.92     0.92        1     0.92     1.14  pso_solve
 12.29      1.06     0.14   200040     0.00     0.00  pso_ackley
  7.02      1.14     0.08    10001     0.00     0.00  inform
  0.00      1.14     0.00    10001     0.00     0.00  calc_inertia_lin_dec
  0.00      1.14     0.00    10001     0.00     0.00  inform_ring
  0.00      1.14     0.00        4     0.00     0.00  pso_matrix_new
  0.00      1.14     0.00        1     0.00     0.00  elapsed_time
  0.00      1.14     0.00        1     0.00     0.00  end_timer
  0.00      1.14     0.00        1     0.00     0.00  init_comm_ring
  0.00      1.14     0.00        1     0.00     0.00  parse_arguments
  0.00      1.14     0.00        1     0.00     0.00  print_elapsed_time
  0.00      1.14     0.00        1     0.00     0.00  pso_calc_swarm_size
  0.00      1.14     0.00        1     0.00     1.14  pso_demo
  0.00      1.14     0.00        1     0.00     0.00  pso_settings_free
  0.00      1.14     0.00        1     0.00     0.00  pso_settings_new
  0.00      1.14     0.00        1     0.00     0.00  start_timer
```

As evidenced by the `gprof` profiling of the demo code, the key sections that the parallel implementation will seek to improve will include the `pso_solve` algorithm elements, the topological functions and related inform functions (`inform_ring`, `inform_random`, and `inform_global`), and the elements that deal with the allocation of matrices.

As well as these functions, the evidence from the profiling suggests that the calculation of the inertia weight should be sped up in light of the call count on that function (`calc_inertia_lin_dec`).

0.5 Serial Path analysis

```
1
2  /* Path options */
3  int inUavID = 0;
4  double inStartX = 70.0;
5  double inStartY = 70.0;
6  double inEndX = 136.0;
7  double inEndY = 127.0;
8  double inStepSize = 1;
9  double inVelocity = 2;
10 double inOriginX = 0;
11 double inOriginY = 0;
12 double inHorizonX = 200;
13 double inHorizonY = 200;  // 70
14 int waypoints = 5;
15
16 /* PSO parameters */
17 double pso_c1 = -1.0;
18 double pso_c2 = -1.0;
19 double pso_w_max = -1.0;
20 double pso_w_min = -1.0;
21 int pso_w_strategy_select = -1;
22 int pso_nhood_size = -1;
23 int pso_nhood_topology_select = -1;
24
25 int pso_w_strategy = -1;
26 int pso_nhood_topology = -1;
```

Listing 2: Path parameter settings and additional PSO settings

In addition to the parameter settings for the demo analysis, the serial parameter settings are shown above. The previous settings that were used for the highest range during the demo analysis were left unchanged and are exactly as shown in that table.

Note: increasing the number of particles through the `PopSize` variable will break the `gsl` generator and lead to a memory leak, so this will be removed. The correct way in this instance will be to adjust the dimensions via the multiplier in the code, as the size of the swarm is conditional on the number of dimensions. This gives us the best of both worlds in the sense that we can test for the effects of an increased number of dimensions as well as an increase in the size of the swarm.

0.5.1 Number of obstacles per run

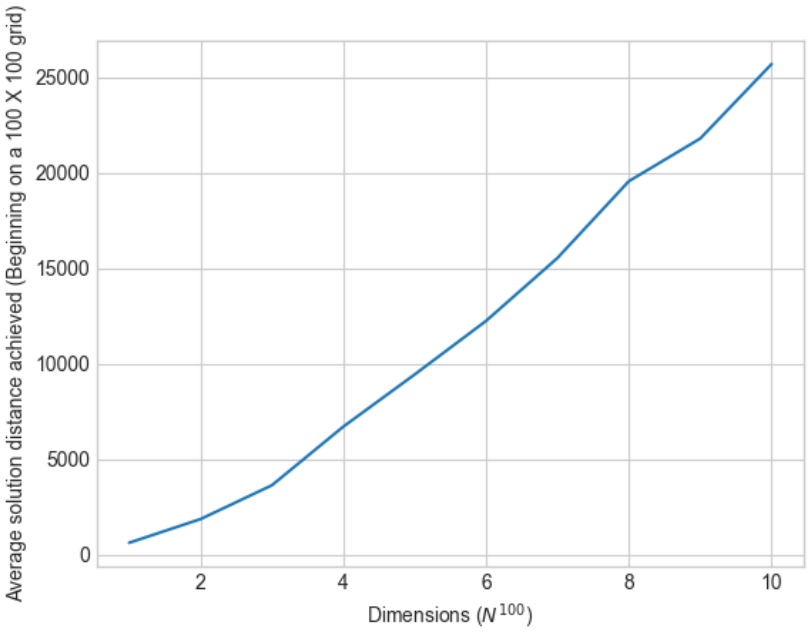
This merits a discussion as the informed observer would think that a random generation of obstacles would give different results no matter how many dimensions each run was conducted in. This is not the case as tests were carried out with a map containing pre-existing obstacles and there were no measurable effects on timing found.

0.5.2 Testing

Testing was conducted on the provided map `sample_map_Doorways.txt` to check for the influence on obstacles on the timing of the initial runs and it was found that they had no real measurable effect. The same can be said for the effect on the error. This is likely due to the low number of obstacles included with the map; this number would have to be increased if there are to be any noticeable effects, but this is beyond the scope of this report.

NOTE: below should be changed to a stacked bar graph upon completion of the parallel tests.

Figure 5: Average solution distance.



Remark. *Will upgrade to include parallel timing.*

Figure 6: Timing across runs.

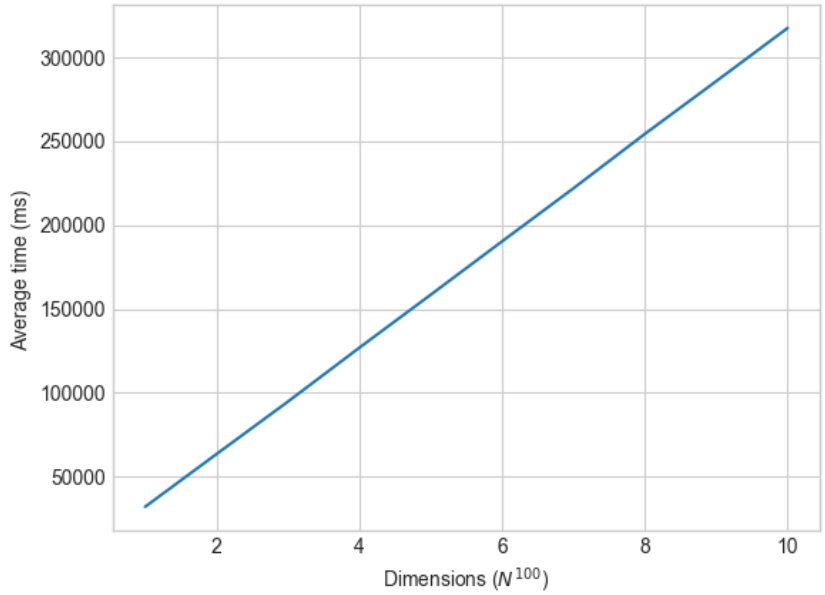


Figure 7: Convergence graph.

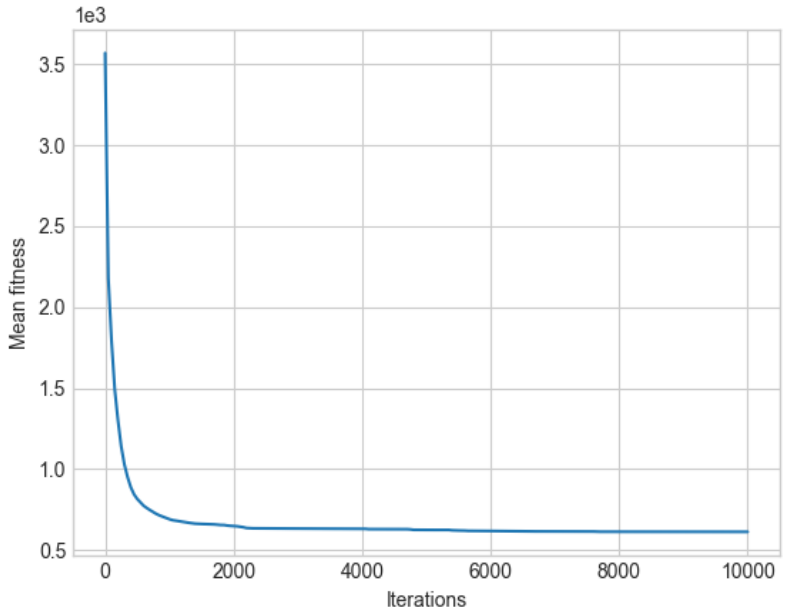


Table 10: Results across different $N^1$00.

N	1	2	3	4	5	6	7	8	9	10
Obstacles (avg)	0	19	99.6	264.8	548	804.4	1284	2058	2034.8	2812.4
Solution distance (avg)	611.78	1854.10	3628.08	6699.95	9435.24	12243.68	15541.26	19564.87	21802.40	25707.28
Avg error	6.6852E+02	2.0838E+03	3.9828E+03	7.1107E+03	9.8900E+03	1.2818E+04	1.6163E+04	2.0074E+04	2.2412E+04	2.6416E+04
Best	4.3074E+02	1.3755E+03	2.1111E+03	5.0095E+03	7.2606E+03	8.1969E+03	1.2458E+04	1.6955E+04	1.9115E+04	2.1504E+04
Worst	4.2178E+03	8.866E+03	1.4031E+04	1.9283E+04	2.3186E+04	2.9204E+04	3.3699E+04	3.8305E+04	4.3142E+04	4.7813E+04
Standard deviation	1.1201E+02	3.3027E+02	6.8822E+02	8.2608E+02	1.0447E+03	1.2427E+03	1.5363E+03	1.1984E+03	1.4494E+03	1.5479E+03
Time (average,ms)	31956	63414.8	94690	126811.2	158425.2	190279.6	221928	254361.2	285816.4	317517.2

0.6 PSO Topologies

Each of the following topologies is used to determine the matrix strategy used to update the neighbouring particles upon running the algorithm. It should be noted that the testing phases for the serial and demo versions specified the global topology as the default for each test. Upon completion of the parallel version, the each topology will be tested with $N = 100$ and compared with the resulting parallel implementation for different numbers of processes.

Both the random and ring topologies are governed by a general inform function, which copies `pos_b` of each particle to `pos_nb` of the matrix itself. This takes the principle of the best informer and applies it to the general function in `pso_solve`. The global inform function is unconnected to this function and is the default function for this purpose.

The idea will be to both initialise and execute the inform communication within a singular function for both global, ring, and random topologies to avoid the communication overhead associated with the initialising and execution of each function separately. This is where `MPI_Cart_create` will be employed to create three separate topologies for such a purpose.

0.6.1 Global

In this case, all particles are drawn to the best position `pos_b` at that particular iteration and copy the contents of `pos_b` to the next best position, `pos_nb`.

0.6.2 Ring

This is a fixed topology. The array is reset and informer particles are chosen, with the diagonal being set to 1. The adjacent particles inform each other in a ring fashion.

0.6.3 Random

An average number of particles is chosen to act as informers, with a random integer generated to denote the topology to be chosen. Each particle informs itself in this case at the beginning, with particle i informing particle j thereafter.

0.7 Parallel Implementation

0.8 MPI functions

0.8.1 `MPI_calc_inertia_lin_dec`

The first function organises the linearly descending inertia weights between different processes to speed up the computation of the appropriate weighting for each iteration. The goal is to ensure that - for example, given 4 processes - that the computation is always 4x ahead.

The function utilises a combination of `MPI_Comm_split` and `MPI_Alltoall` to first create a new communicator to based on the row order and to split the communicator on that basis. The original rank is used for ordering. What happens then is that the value gets stored in `val[i]` and distributed among all processes through `MPI_Alltoall`. As it runs, the program accounts for a total of $\omega * 4$ weights for a program run across 4 processes in parallel. This will need to be adjusted so that the program calculates an individual weight, but to make it available to all processes.

0.8.2 MPI topologies

The general inform function will be cut out due to the ability of `MPI_Cart_shift` and related functions to enable efficient communication between processes, which will be mapped with the individual values for `pos_b` and `pos_nb`. These two variables will feature prominently in the new topological set-ups given their link to delivering a solution.