Assignment 6 Report

AEROSP 588 – Dr. Joaquim Martins

University of Michigan

November 18, 2025

Austin Leo Thomas

I. Problem 6.1

A particle swarm optimization (PSO) algorithm was developed in Python for use in this investigation into the efficacy of gradient-free optimizers. This algorithm was put to use in solving several optimization problems with its results being compared to other gradient-free and -based algorithms.

When provided with bounds on the design space (note: not constraints), the PSO algorithm randomly generated $10n_x$ points within the bounded design space, where n_x is the number of design variables (i.e. the dimensionality of the problem). This selection for swarm size was based on the heuristic principle that the population size for an evolutionary algorithm should be at least one order of magnitude greater than the size (i.e. dimensionality) of the problem [1].

At each iteration, each point was updated according to a step size defined by Eq. (7.29) in *Engineering Design Optimization* [1]. Three parameters dictated this step size definition: α, the inertial parameter; β, the memory parameter; and γ, the social influence parameter. The terms in Eq. (7.29) associated with these parameters control the tendency of a given point to: continue moving in the same direction (inertial), turn towards the most optimal point it has been at (memory), and turn towards the best point the swarm as a whole has been at (social influence). In this manner, each point steps towards increasingly optimal locations in the design space, when considered over many iterations.

The heuristic principle for selection of α is that $\alpha \in [0.8,1.2]$ [1]. The middle value in this range was selected for this investigation, so as to provide a balanced inertial term; thus $\alpha = 1$ for all steps in the PSO algorithm.

To introduce a stochastic element to the algorithm, β- and γ-values are varied randomly and uniformly across their ranges $\beta \in [0, \beta_{\max}]$ and $\gamma \in [0, \gamma_{\max}]$, where these maximum values typically exist in the bounds of $\beta_{\max} \in [0,2]$ and $\gamma_{\max} \in [0,2]$ [1]. As a heuristic principle, the maximum values typically lie towards the upper end of this range, so as to allow all points to converge more readily to potential minima. As such, for the purposes of this investigation, $\beta_{\max} = 2$ and $\gamma_{\max} = 2$.

Defining the convergence criteria for gradient-free algorithms can be difficult. For the PSO algorithm developed here, the primary test of convergence was a check on the value of the current best point's function value and the previous best point's function value. The criteria requires that, for three subsequent iterations, the current swarm optimum value and the previous swarm optimum have a relative error between on another of $< 1 \cdot 10^{-5}$. Speaking plainly, this requires that the previous best point is essentially identical to the current best point; if such behavior is detected three times in a row, it is considered reasonable to assume that the system has converged. This is a common heuristic for determining convergence of many types of gradient-free algorithms [1]. As a failsafe, iteration and function call limits were also implemented, as testing showed that the algorithm could quickly run awry.

One optional element of a PSO algorithm is a limit on the maximum step size; such a limit can prevent a particle from shooting off in enormous steps due to influence from the memory or social influence terms, or due to a compounding inertial term. A heuristic principle is to set the maximum step size equal to one percent of the smallest breadth of the bounded design space [1]. That is to say, if the bounded design space is imagined as a hyperrectangle, the maximum allowable step size for any particle is set as one percent of the shortest dimension of that hyperrectangle. This principle was used to limit the maximum step size in the PSO algorithm implemented here.

Since this algorithm carries stochastic elements, in all cases of its implementation in this investigation, its results were subjected to statistical analysis. That is, a statistical sample was obtained by running the algorithm multiple times, and all results were asses for their mean and standard deviation values. As a heuristic principle, the traditionally-accepted minimum sample size required to obtain a normal distribution is $n = 30$. At all instances throughout this investigation, the sample size used for generating the statistical distribution is noted. Unfortunately, due to the inefficient nature of PSO algorithms, obtaining a full set of $n = 30$ samples was not always possible. In such instances, the somewhat bold assumption that a set of $n = 10$ samples would offer a statistically viable set was made.

Another nuance of PSO algorithms is the case where particles find themselves outside the original design space bounds. Since the problems tackled here are artificially bounds (that is, they are in fact defined over an infinite design space), no precautions were taken to prevent particles from leaving the bounds of the design space or to correct their path back towards the defined design space. In such cases, the issue was understood to be self-correcting, since design space bounds were selected to surround the known minima. In such cases where particles existed the bounds of the design space, the memory and social influence terms in Eq. (7.29) would quickly turn the particle back towards the bounded space.

II. Problem 6.2

The algorithm was first set loose on the bean function, defined by Eq. (D.4) in *Engineering Design Optimization* [1]. The bean function is a two-dimensional function with an analytic minima of $f(x^*) \cong 0.09194$ at the optimum point $\sim x^*(1.21314, 0.82414)$.

To solve this problem, the PSO algorithm was run to generate for $n = 30$ samples, each with distinct starting particle swarms within the design variable bounds $x_1 \in [-3,3]$ and $x_2 \in [-3,3]$. The results of this statistical study, including the relative percent error in the results when compared to the analytic solution, are shown in Table (I).

Table I
PSO Results: Bean Function

Result	Mean	Std. Dev.	Error
x_1^*	1.21355	$\pm 1.3 \cdot 10^{-3}$	0.033%
x_2^*	0.82437	$\pm 1.5 \cdot 10^{-3}$	0.028%
$f(x^*)$	0.09195	$\pm 1.3 \cdot 10^{-5}$	0.011%
f-calls	21,507	$\pm$20,863	-

Examination of the results shown in Table (I) reveal fairly accurate values, at the cost of a regrettably high average number of function calls required for convergence. However, the accuracy of these results is far

from machine precision, with absolute errors in the optimum design variable and objective function values on the order of 10^{-4}. Given the high computational cost of this approach, it is difficult to consider the results of the study to be favorable toward the algorithm's efficacy.

Though a statistical approach is the correct methodology for analyzing the results of any stochastic optimization algorithm, it is useful to explore the best result from this statistical study, as it illustrates the high level of efficiency that the PSO algorithm is capable of when the stochasticity works to its favor. Here, the "best" result is considered to be that which meets the convergence criteria with the minimal number of function evaluations – not necessarily the sample which is the most accurate. The results of this best sample are shown in Table (II).

Table II
PSO Best Result: Bean Function

Result	Value	Error
x_1^*	1.21589	0.227%
x_2^*	0.82212	0.245%
$f(x^*)$	0.09200	0.065%
f-calls	940	-

Examination of the results shown in Table (II) reveal that, when compared to the overall distribution of the sampled set, the best solution converges much more rapidly. With only 940 function calls required for convergence, the result indicates a highly computationally efficient solution. However, examination of the optimal design variables and function values reveals that the cost of this efficiency is the accuracy of the result itself: the relative errors in the converged results here are roughly an order of magnitude greater than the relative errors in the statistical average result. The absolute errors in the optimal design variable values are on the order of 10^{-3} while the absolute error in the optimal function value is on the order of 10^{-4}. While the result is impressive from an efficiency perspective, the inaccuracy in the solution does not bolster a strong case for the overall efficacy of the PSO algorithm.

To further understand the algorithm, and to understand how such an inaccurate solution was converged to so rapidly, the particle swarm was plotted in the design space at several iterations of the best solution defined above. These plots, generated for iteration values $k = 0, 10, 25,$ and 47, are shown in Fig. (1) through (4), respectively.

Examination of Fig. (1) and (2) reveal the general trend of the algorithm. Fig. (1) shows the initial state of the particle swarm: for the two-dimensional problem, 30 particles were randomly generated in the bounded design space. After 10 iterations of the PSO algorithm loop, the particle swarm then took on the arrangement shown in Fig. (2). Glancing

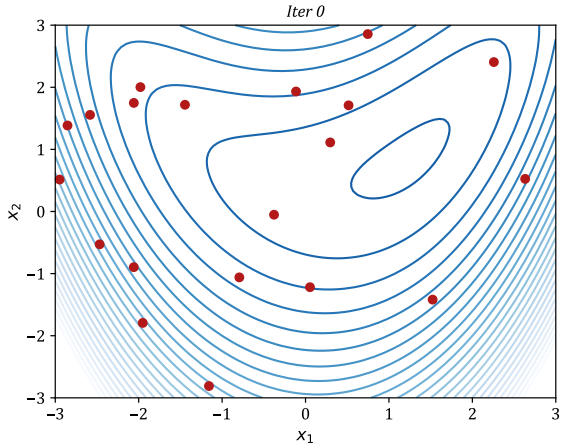

Figure 1. Particle swarm for "best" bean function solution at k = 0*. This is the randomly-generated swarm created prior to running the particle swarm optimization loop.*

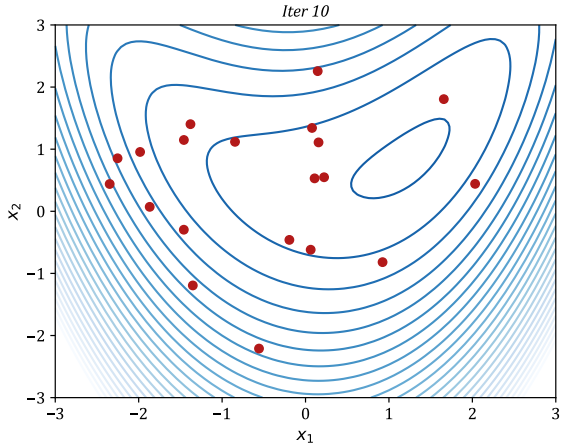

Figure 2. Particle swarm for "best" bean function solution at $\mathrm{k} = 10$.

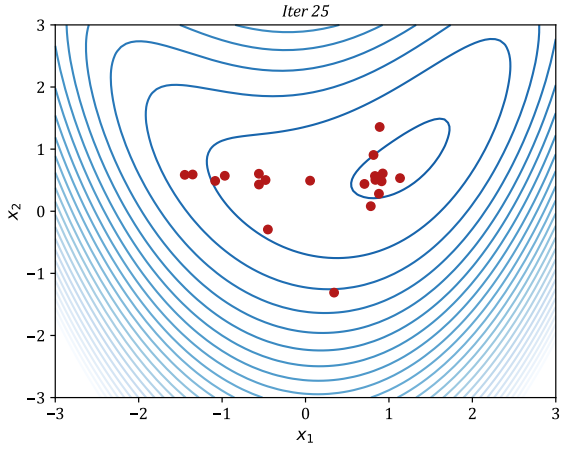

Figure 3. Particle swarm for "best" bean function solution at $\mathrm{k} = 25$.

between these two plots it is clear that, over the course of the first ten iterations, the particles collectively migrated nearer to the analytic minimum.

Examination of Fig. (3) begins to reveal some interesting behavior. Firstly, it is readily observed that the particles on the left began to form a straight-line path towards the cluster of particles nearer to the analytic minima.

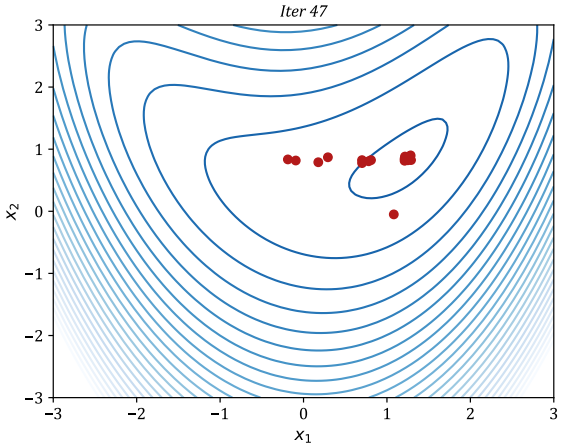

Figure 4. Particle swarm for "best" bean function solution at $\mathrm{k} = 47$. *This is the final particle swarm configuration at the iteration in which the convergence criteria was met.*

Viewed in light of Eq. (7.29), such behavior can be understood to be driven first by the social influence term and then by the inertial term, all while the memory term has no effect on the step. Along this imaginary path, each particle is constantly descending: as such, the memory term is equal to zero, since each particles historic best point is its current point. The dominating effect is then the social influence term, which drives all particles in the same direction; over several iterations this directional influence is compounded by the inertial term. With this considered, the tendency of the points far from the swarm's best point to drive directly towards the swarm's best point is very logical.

Another interesting observation from Fig. (3) is that the cluster of points near the analytic optimum are in fact slightly offset from the true minima location: this is made obvious by glancing between Fig. (3) and (4), as a noticeable shift in the particle clusters in each plot can be seen. This reveals a downside of the PSO algorithm: if the social influence

term is too dominant in defining the step size and direction, particles may all tend towards the swarm's best point even if this point is not the function optimum. Since all stochasticity in the PSO algorithm is involved with the magnitude of the step rather than perturbing the step direction, particles have little influence to explore the design space asides from the effect of the inertial term. In this sense, only by particles overshooting the swarm's best point, or by the best point itself shifting, are better points found.

In this instance, the function has no local minima, and as such convergence to the global minima is almost inevitable: here, it is probable that the inertial effect of the far-left particles in Fig. (3) caused these particles to overshoot the swarm's best point and discover the true function optimum. If this algorithm were applied to a function with many minima, however, it is likely that the social influence term might cause the algorithm to converge to a local minima due to a lack of thorough exploration of the design space.

With this basic case considered, attention was then paid to the effect of noise on the efficacy of the PSO algorithm. The bean function was perturbed by a noise function defined by a Gaussian distribution centered around zero with some standard deviation σ. Initially, this noise parameter was set at a small value of $\sigma = 10^{-4}$. Every time the bean function was evaluated, its function output was perturbed by a random noise pulled from this Gaussian distribution.

This noisy bean function was optimized using both the PSO algorithm and a conjugate gradient method using a strong-Wolfe line search (referred to here as the CG algorithm). For the gradient-based CG algorithm, the elements of the gradient were perturbed, individually and independently, by a value from the same Gaussian distribution. Since the PSO algorithm is stochastic, and the noise generation method is likewise stochastic, a statistical treatment was offered to both the gradient-free and gradient-based approaches. To add stochasticity to the gradient-based approach, the initial guess was generated randomly and uniformly within the bounded design space for each sample. For both methods, $n = 10$ samples were collected. The results of this exploration are shown in Table (III).

Table III
Noisy Bean Function Results: $\sigma = 10^{-4}$

Method	Value	Mean	Std. Dev
PSO	x_1^*	1.21380	$\pm 6.1 \cdot 10^{-3}$
	x_2^*	0.82389	$\pm 6.3 \cdot 10^{-3}$
	$f(x^*)$	0.09197	$\pm 5.7 \cdot 10^{-5}$
	f-calls	93,288	$\pm$68,772
CG	x_1^*	1.21328	$\pm 7.4 \cdot 10^{-4}$
	x_2^*	0.82398	$\pm 8.9 \cdot 10^{-4}$
	$f(x^*)$	0.09196	$\pm 8.5 \cdot 10^{-5}$
	f-calls	287,174	$\pm$3,498

Examining Table (III) reveals that the PSO algorithm actually performed better than the CG algorithm for this noisy bean function. Both methods converged to very similar results, with absolute errors in the optimum design variables on the order of 10^{-4} and absolute errors in the optimum function values on the order of 10^{-5} for both methods. The PSO algorithm, however, required roughly one-third of the functional calls that the CG algorithm utilized. This can best be understood by the fact that, for small noise values, the PSO algorithm is largely unaffected so long as step sizes are sufficiently large. The CG algorithm, however, relies on gradients which may now have sign errors near the

optimal value. Since the convergence criteria for the CG algorithm relies on the gradient falling below a certain value, noise in the gradient becomes increasingly troublesome for defining convergence.

To further explore the notion that the PSO algorithm may prove more robust than the CG algorithm when faced with noise, the same test was run for increasingly large noise parameters. Since the addition of a noise perturbation directly to the function evaluation negates any meaning of the optimum function value, it was not considered. Additionally, since the convergence criteria of both algorithms became impossible to meet with the introduction of large noise values, both algorithms were limited to 200,000 function evaluations; as such, both methods were forced to be equally computationally expensive. The results of this study are shown in Table (IV).

Examination of Table (IV) reveals that, event for rather significant noise levels, both functions performed well. For noise on the order of 10^{-3} and 10^{-2}, absolute errors in design variable values were on the order of 10^{-3} for both methods. For noise on the order of 10^{-1}, absolute errors in design variable values were on the order of 10^{-2} for both methods. Only for noise on the order of unity did the methods deviate: here, the relative error in the PSO results were 23.4% and 28.1%, for x_1 and x_2, respectively, while the relative error in the CG results were much lower, at 6.8% for x_1 and 9.0% for x_2. The standard deviations in all cases did not raise cause for concern that any distributions were non-Gaussian.

In light of these results it is seen that, for an equal number of function evaluations, the PSO algorithm does not out-perform the CG algorithm even for large noise levels. For sufficiently large noise levels, in fact, the PSO algorithm becomes highly ineffective at finding a correct solution. At middling noise levels, both methods perform comparably for an equivalent computational cost. This indicates that, even when allowed many function evaluations, both methods are fundamentally limited in their accuracy when faced with high levels of noise. However, the result from Table (III) remains: for small perturbations in the function value, the computational cost of the PSO algorithm is much less than the CG algorithm, despite comparable results.

Table IV

Noisy Bean Function Results: $\sigma = 10^{-3}$			
Method	Value	Mean	Std. Dev
PSO	x_1^*	1.21234	$\pm$0.01402
	x_2^*	0.82530	$\pm$0.01324
CG	x_1^*	1.21293	$\pm$0.00156
	x_2^*	0.82464	$\pm$0.00174
Noisy Bean Function Results: $\sigma = 10^{-2}$			
Method	Value	Mean	Std. Dev
PSO	x_1^*	1.2111	$\pm$0.02329
	x_2^*	0.82684	$\pm$0.03701
CG	x_1^*	1.21343	$\pm$0.00815
	x_2^*	0.82312	$\pm$0.00338
Noisy Bean Function Results: $\sigma = 10^{-1}$			
Method	Value	Mean	Std. Dev
PSO	x_1^*	1.2228	$\pm$0.030928
	x_2^*	0.82296	$\pm$0.042767
CG	x_1^*	1.20844	$\pm$0.03028
	x_2^*	0.82615	$\pm$0.03200
Noisy Bean Function Results: $\sigma = 10^{0}$			
Method	Value	Mean	Std. Dev
PSO	x_1^*	0.92966	$\pm$0.33635
	x_2^*	0.59239	$\pm$0.35701
CG	x_1^*	1.13104	$\pm$0.18638
	x_2^*	0.74963	$\pm$0.18495

Next, the efficacy of the PSO algorithm was evaluated when faced with a discontinuous function. A perturbation was added to the bean function as defined by Eq. (7.31) in *Engineering Design Optimization* [1]. This generated a checkered function which rose and fell by units of four in a grid across the design space. The minima of this checkered bean function was the same as the unaltered bean function.

The same methodology as was applied for the noise investigation was applied here, except no restriction was placed on the number of function evaluations: the methods were left to converge by their own criteria. The results are shown in Table (V).

Table V
Checkered Bean Function Results

Method	Value	Mean	Std. Dev
PSO	x_1^*	1.21303	$\pm 3.0 \cdot 10^{-4}$
	x_2^*	0.82398	$\pm 3.8 \cdot 10^{-4}$
	$f(x^*)$	0.09195	$\pm 1.1 \cdot 10^{-6}$
	f-calls	38,120	$\pm$19,873
CG	x_1^*	1.27252	$\pm$0.17734
	x_2^*	0.94171	$\pm$0.35276
	$f(x^*)$	0.27513	$\pm$0.54955
	f-calls	41,498	$\pm$122,035

Examination of Table (V) reveals that there is finally a case where the PSO algorithm is definitively better than the CG algorithm. The relative errors in the PSO results are 0.009%, 0.019%, and 0.011% for x_1^*, x_2^*, and $f(x^*)$, respectively. The same results for the CG algorithm are 4.89%, 14.3%, and 199%, respectively. The PSO algorithm was also able to accomplish this superior result in marginally fewer function evaluations.

Further examination of the standard deviations of the statistical distributions reveals the possible cause of this outcome. The distributions of all values associated with the PSO result are much tighter than those associated with the CG result. This indicates that outliers may be unduly influencing the outcome of the CG study, or that the distribution of results for the CG study is not Gaussian. Given the methodology behind the strong-Wolfe line search which the CG algorithm implements, it is highly likely that the CG algorithm repeatedly got stuck in the local minima which developed at points of discontinuity in the checkered function. Such behavior would explain the larger distribution and the large number of function evaluations employed by the CG algorithm. Regardless of the cause, it is clear that, when dealing with highly discontinuous functions such as this, a gradient-free method such as the PSO algorithm is potentially superior to a gradient-based method such as the CG algorithm.

Finally, an effort was made to compare both methods in the setting of a three-dimensional problem. The function to be minimized is expressed in Eq. (1). It's analytic minima is $f(x^*) = 0$ at $x^*(0,0,0)$.

$$f(x) = |x_1| + 2|x_2| + x_3^2 \qquad (1)$$

The same methodology as the previous two studies was applied again; the results are shown in Table (VI). Examination of these results reveals that, as function dimensionality increases, the efficacy of the PSO algorithm may decrease dramatically. While the CG algorithm had no issue converging to the correct solution, correct to machine precision, the PSO algorithm yielded design variables correct to only two decimal places. While the

Table VI
Eq. (1) Optimization Results

Method	Value	Mean	Std. Dev
PSO	x_1^*	0.00115	$\pm$0.00311
	x_2^*	-0.0003	$\pm$0.00081
	$f(x^*)$	0.00295	$\pm$0.00358
	f-calls	103,005	$\pm$116,797
CG	x_1^*	$8 \cdot 10^{-32}$	$\pm 4 \cdot 10^{-31}$
	x_2^*	$2 \cdot 10^{-32}$	$\pm 3 \cdot 10^{-31}$
	$f(x^*)$	$5.5 \cdot 10^{-5}$	$\pm$0.00016
	f-calls	390,295	$\pm$29,677

PSO algorithm used significantly less function evaluations to arrive at its solution, such computational efficiency means very little when accuracy is severely compromised. This brief investigation reveals that the PSO method may begin to deteriorate as problem dimensionality increases, as instances arise such as this where the convergence criteria are met despite a poor result.

III. Problem 6.3

To further explore the effects of problem dimensionality on the efficacy of the PSO algorithm, several methods were employed to tackle to n-dimensional Rosenbrock function, represented in Eq. (D.3) in *Engineering Design Optimization* [1].

The PSO algorithm was applied with a sample set of $n = 10$, as before. Other algorithms were applied with initial guesses at the fringe of the bounded design space presented to the PSO algorithm. For this problem, the bounded design space required the magnitude of all design variables to be less than or equal to 2; as such, all starting guesses had all design variables set to -2. An off-the-shelf Nelder-Mead and finite-difference solver were used (via SciPy) as well as the same self-made CG algorithm employed in *Section II*. Results for the PSO algorithm converged only up to $n = 16$ (an attempt at $n = 32$ was terminated manually after 2.5 hours); as such, only those problems are considered here. The numerical results of this investigation are shown in Tables (VII) and (VIII). For the PSO results, the mean is presented as the singular value and the standard deviation is omitted, since it is not relevant here.

Table VII
Rosenbrock Function Results: $n = 2$

Method	Result	Value
PSO	x_1^*	(0.9968202,0.9915659)
	$f(x^*)$	0.00345968
	f-calls	195,716
NM	x_1^*	(1.00001,1.00003)
	$f(x^*)$	$1.81794 \cdot 10^{-10}$
	f-calls	149
FD	x_1^*	(0.999997,0.999994)
	$f(x^*)$	$8.52092 \cdot 10^{-12}$
	f-calls	102
CG	x_1^*	(1,1)
	$f(x^*)$	$7.98856 \cdot 10^{-14}$
	f-calls	6,833

Rosenbrock Function Results: $n = 4$

Method	Result	Value
PSO	x_1^*	$(1, \dots, 1)$
	$f(x^*)$	0.00204840
	f-calls	915,042
NM	x_1^*	$(1, \dots, 1)$
	$f(x^*)$	$1.45618 \cdot 10^{-8}$
	f-calls	570
FD	x_1^*	$(1, \dots, 1)$
	$f(x^*)$	$1.81799 \cdot 10^{-11}$
	f-calls	245
CG	x_1^*	$(1, \dots, 1)$
	$f(x^*)$	$8.07991 \cdot 10^{-14}$
	f-calls	40,176

Table VIII

Rosenbrock Function Results: $n = 8$

Method	Result	Value
PSO	x_1^*	$(1, \dots, 1)$
	$f(x^*)$	$2.85529 \cdot 10^{-6}$
	f-calls	3,346,928
NM	x_1^*	$(0.92, 0.86, \dots, -0.04)$
	$f(x^*)$	3.03598
	f-calls	1,600
FD	x_1^*	$(1, \dots, 1)$
	$f(x^*)$	$2.97164 \cdot 10^{-11}$
	f-calls	684
CG	x_1^*	$(1, \dots, 1)$
	$f(x^*)$	$1.52031 \cdot 10^{-12}$
	f-calls	29,159

Rosenbrock Function Results: $n = 16$

Method	Result	Value
PSO	x_1^*	$(1, \dots, 1)$
	$f(x^*)$	$7.50098 \cdot 10^{-9}$
	f-calls	3,800,888
NM	x_1^*	$(0.56, 0.051, \dots, 0.35)$
	$f(x^*)$	139.48804
	f-calls	3,200
FD	x_1^*	$(1, \dots, 1)$
	$f(x^*)$	$3.58951 \cdot 10^{-10}$
	f-calls	2057
CG	x_1^*	$(1, \dots, 1)$
	$f(x^*)$	$9.39868 \cdot 10^{-13}$
	f-calls	29,884

Evaluation of Tables (VII) and (VIII) reveal several interesting results. The first, and most surprising, observation to be made is that for $n = 8$ and $n = 16$, SciPy's Nelder-Mead optimizer does not converge. In both instances the returned optimum point and optimum function value are very far from the analytic results of $f(x^*) = 0$ at $x^*(1, \dots, 1)$. This is interesting considering the low number of function calls utilized by these solutions. Given that the number of function calls for $n = 8$ and $n = 16$ are 1600 and 3200, respectively, it is clear that some internal stopping criteria was hit rather than an actual convergence criteria. Nonetheless, it is interesting that the SciPy tool gave up so readily when faced with a relatively simple problem. This is indicative of possible shortcomings of off-the-shelf gradient-free approaches as a whole: that is, that such methods might has little ambition to persist towards extremely costly solutions.

The PSO algorithm, surprisingly, converged in all cases (though the $n = 16$ solution took approximately 45 minutes to return). Indeed, the result in all four cases was quite accurate (and extremely costly). Interestingly, the accuracy of the PSO results seemed to increase with the problem dimensionality: the $n = 16$ solution yielded an optimal function value nearest to the analytic solution, and was in fact comparable to the accuracy of the off-the-shelf finite-difference method. This can almost certainly be accounted for by the extreme number of function calls required to arrive at this solution, and the fact that the convergence criteria for the PSO algorithm was not loosened for this study.

The results of the finite-difference method are not surprising, and serve as a good baseline here: the method yielded acceptable, though not perfect, results with a consistently low number of function evaluations. The conjugate-gradient method yielded the most accurate results, with an optimal function value several orders-of-magnitude smaller than the runner-up, the finite difference method. This accuracy came at the cost of an order-of-magnitude increase in computational cost, however. It should be noted, though, that the CG algorithm used here was developed by hand and is not from the SciPy package: the SciPy CG algorithm would certainly be more

efficient; as such, a comparison between the FD and CG results here is not terribly founded.

As a whole, these results reveal the limitations of gradient-free algorithms. While the gradient-based approaches easily converged to highly accurate results with relatively low computational costs, the gradient-free methods struggled: the NM algorithm failed to converge at higher n-values and the PSO algorithm yielded accurate results only after an exorbitant number of function evaluations.

To better understand the severity of these limitations, a logarithmic plot of the number of function calls required for convergence of each algorithm, as a function of problem dimensionality, was generated. This plot is shown in Fig. (5).

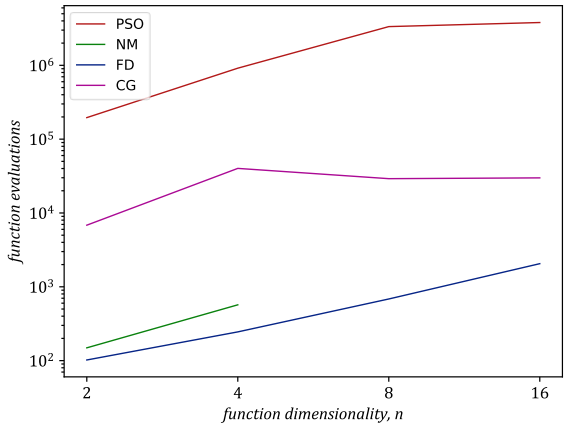

Figure 5. Relationship between the number of function calls required for convergence of each algorithm and the dimensionality of the Rosenbrock function being minimized.

Examination of Fig. (5) further reveals interesting results from this investigation. Of particular not is the fact that the trend for the CG algorithm seems to level off after the $n = 4$ solution – in fact, the two higher-dimension problems are solved by the CG algorithm with fewer function calls that the $n = 4$ case. This is likely anomalous and a result of the somewhat inefficient writing of the self-made CG code. Likewise, the PSO algorithm is observed to level off somewhat following the $n = 8$ solution. Again, this is likely anomalous and a result of inefficient programming.

One very interesting result, however, is that all four trends seemingly have the same slope: when compared to Fig. (7.1) in *Engineering Design Optimization*, it is clear that this, in theory, should not be the case [1]. In the PSO case, a small sample size may well be skewing the results for higher dimensions simply by chance. Additionally, such a small range of n-values makes it difficult to draw concrete conclusions. While the results shown in Fig. (5) are peculiar, the investigation done here simply is not of a large enough scale to rule out stochastic and anomalous effects which may be skewing the results.

IV. Disclosures

As required by the syllabus, a disclosure is included here regarding AI use on this assignment. U-M GPT was used as an aide when writing the Python scripts employed in this assignment.

V. References

[1] J. R. Martins and A. Ning, Engineering Design Optimization, Cambridge: Cambridge University Press, 2020.