Parallelising Quantum-Behaved Particle Swarms

ST444: Statistical Computing

LENT TERM 2019

Authors
22146
13185
14974
20751

Contents

1 Introduction

Particle Swarm Optimization (PSO) is a gradient-free heuristic optimization algorithm first proposed in a seminal paper by Eberhart & Kennedy (1995). The algorithm searches for a function's global minimum over a continuous search space using simple rules derived from the observed behaviours of social organisms navigating a complex environment; we refer to such rules as swarm intelligence (SI) rules. The aim of this project is threefold:

1. We formally introduce the PSO algorithm as well as a recent variant, the Quantum-behaved Particle Swarm Optimization (QPSO) algorithm, and show how these can be implemented in Python.
2. We briefly revisit the death of Moore's law, Waldrop (2016), and therefore motivate parallel computing as a crucial tool in modern statistical computing.
3. We analyze the performance of parallelised and non-parallelised implementations of PSO and QPSO in Python across a range of test functions.

The report is structured as follows: section 2 formally introduces the PSO algorithm, section 3 formally introduces the QPSO algorithm, section 4 discusses some practical concerns regarding parallelisation and implementation of our algorithm, section 5 describes the actual implementation of the PSO and QPSO algorithms in Python, section 6 discusses the performance of the algorithms for different test functions, and section 7 offers some concluding remarks.

2 Particle Swarm Optimisation

2.1 Motivation

The algorithm originally proposed by Eberhart & Kennedy (1995) was motivated by early computer simulations of birds flocking, in particular, Reynolds (1987). In the PSO algorithm, particles in a swarm represent potential solutions. At each iteration, particles update their position based on their own best solution for the function's minimum as well as the swarm's best solution. PSO was in part inspired by earlier genetic optimization algorithms, however, the latter presents two main advantages over the former. Firstly, interactions within the group enhance, rather than detract from, progress towards the solution. Secondly, the algorithm has 'memory': knowledge of good solutions is retained by all particles, and particles which 'fly past' good solutions are 'tugged back'.

2.2 Technical workings

When implementing the PSO algorithm to find the minimum of a function $f : \mathbb{R}^D \to \mathbb{R}$ using a swarm of N_p particles, each particle $i \in N_p$ updates its position $P_i(t+1)$ position according to its previous position $P_i(t)$ and its current 'velocity' $V_i(t+1)$ as shown below:

$$P_i(t+1) = P_i(t) + V_i(t+1) \tag{1}$$

$$V_i(t+1) = \omega V_i(t) + c_1 r_1 (pbest_i(t) - P_i(t)) + c_2 r_2 (gbest(t) - P_i(t)) \tag{2}$$

PSO variables key	
ω	inertia weight: balances global and local exploitation
c_1	cognitive parameter: importance given to *pbest*
r_1	$\sim_{\text{i.i.d.}} \mathcal{U}(0,1)$
c_2	social parameter: importance given to *gbest*
r_2	$\sim_{\text{i.i.d.}} \mathcal{U}(0,1)$
$pbest_i(t)$	$\arg\min_{k=1,..t}[f(P_i(t))], \quad i \in \{1,2,...,N_p\}$
$gbest(t)$	$\arg\min_{i=1,...,N_p,k=1,...,t}[f(P_i(k))]$

Equations (1) and (2) highlight the main advantages of the PSO algorithm over competing algorithms: the algorithm relies on simple paradigms, and for sensible choices of N_p is computationally inexpensive. The PSO algorithm makes few assumptions about the function being optimized: the function may be non-convex, may have many local minima, and it is not necessary for the gradient to exist. However, for functions with local minima, there is no guarantee of convergence to the global minimum. The cost of each iteration t is of order $\mathcal{O}\left(\sum_{i=1}^{N_p} cost(f(P_i(t)))\right)$, which is bounded from above by $\mathcal{O}(c \cdot N_p)$ if the cost of evaluating f is bounded. Hence, for a fixed number of iterations I the algorithm has computational complexity $\mathcal{O}(Ic \cdot N_p) = \mathcal{O}(N_p)$; of course, the cost of evaluating f will depend on its dimensions.

Figure 1 shows the movement of a single particle during one iteration of the algorithm. The extent to which particles explore new regions in the search space, 'value' their own experience, and 'value' the swarm's experience can be controlled by adjusting ω, c_1, and c_2 respectively. For example, setting $c_2 = 0$ results in particles exploring the search space with no 'knowledge' of the swarm's best estimate of the function's minimum. Zhang et al. (2015) note that it is common practice to set an upper bound for the velocity parameter to prevent particles from flying out of the search space.

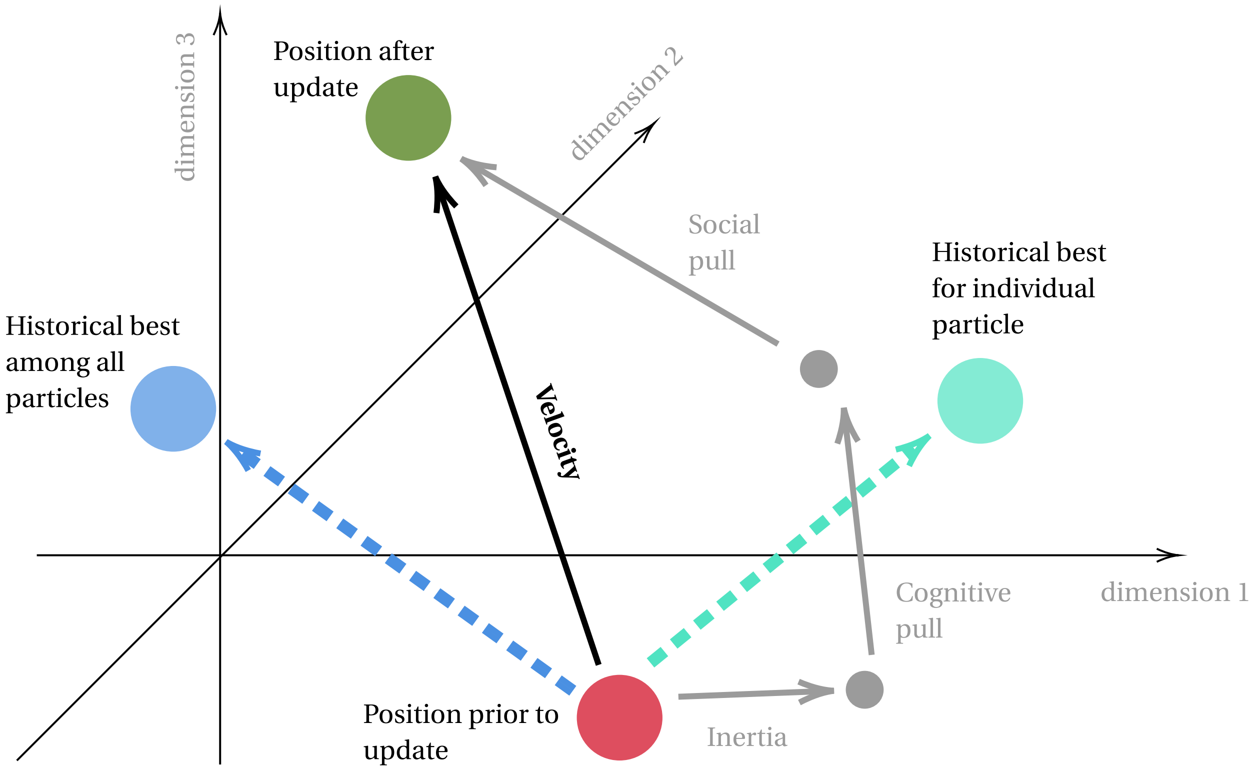

Figure 1: behaviour of a single particle in a swarm

2.3 Applications and variants

As a result of its flexibility and ease of implementation, the algorithm has been applied to problems ranging from training neural networks to classify cancer-causing genes Saraswathi et al. (2011) to forecasting wind power production Mandal et al. (2014). While the original algorithm was modeled on the behaviour of birds, many PSO variants using SI based on other animals have been developed. The wolf pack algorithm of Yang et al. (2007) mimics the tendency for wolves to separate into sub-groups when hunting. The whale algorithm of Mirjalili & Lewis (2016) mimics the 'bubble-net' feeding strategy used by humpback whales, and when implemented leads to particles spiralling in on the best solution.

3 Quantum Particle Swarm Optimisation

3.1 Motivation

The QPSO algorithm first introduced by Sun, Feng & Xu (2004), and was motivated by an analysis of convergence properties of traditional PSO algorithms carried out by Clerc & Kennedy (2002). In classical mechanics, a particle's movement through space is determined by a position vector $P_i(t)$ and a velocity vector $V_i(t)$ (see L.H.S. of figure 2). In quantum mechanics, however, the idea of 'trajectory' is meaningless, since the *uncertainty principle* ensures a that particle's position and momentum may never be jointly known (see Heisenberg (1985)). Instead, a particle may be depicted by a complex valued quantum wave function, $\psi(x,t)$. Born (1927) noticed that the square of the absolute value of the wave function can be interpreted as a probability distribution function, and so $\psi(x,t)$ can be thought of as a probabilistic model for a particle's most likely locations (see R.H.S of figure 2).

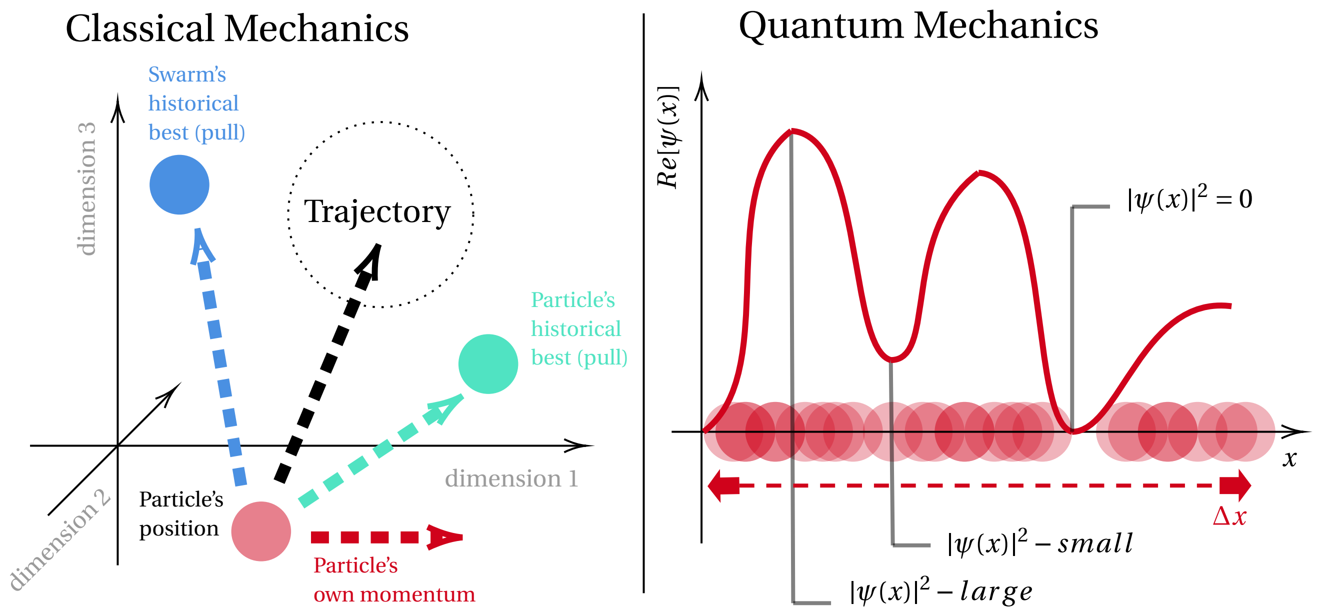

Figure 2: movement of a Newtonian particle (L.H.S.) versus a quantum-bit (R.H.S.)

Sun et al. Fang et al. (2010) adapt the traditional PSO algorithm by treating each particle in a swarm as a quantum-bit (qbit). In keeping with quantum mechanics each qbit's exact location is not known with certainty. Rather, at each iteration particles are updated according to their most likely location as given by $\left|\psi(x,t)\right|^2$.

3.2 Technical workings

When implementing the QPSO algorithm with a swarm of N_p qbits, at each iteration qbits update their most likely position $P_i(t)$ according to the value of a local attractor $A_i(t)$ and what we will call a local exploration variable $L_i(t)$:

$$P_i(t+1) = A_i(t) + (-1)^{C_i(t)} L_i(t) \cdot \log u_i(t) \tag{3}$$

$$A_i(t) = \varphi_i(t) \cdot pbest(i,t) + (1-\varphi_i(t)) \cdot gbest(t) \tag{4}$$

$$L_i(t) = 2\beta \cdot \left|pbest_i(t) - P_i(t)\right| \tag{5}$$

QPSO variables key	
$C_i(t)$	coin toss variable: $\sim_{\text{i.i.d}}$ *Bernoulli* (0.5)
$u_i(t)$	$\sim_{\text{i.i.d.}} \mathcal{U}(0,1)$
$\varphi_i(t)$	$\sim_{\text{i.i.d.}} \mathcal{U}(0,1)$
β	contraction-expansion coefficient
$pbest_i(t)$	$\arg\min_{k=1,..t}[f(P_i(t))] \quad i \in \{1,2,...,N_p\}$
$gbest(t)$	$\arg\min_{i=1,...,N_p,k=1,...,t}[f(P_i(k))]$

The parameter β may be tuned to control the convergence rate of the algorithm. Additionally, the value of $L_i(t)$ is important for convergence. Sun, Xu & Feng (2004) introduce the idea of 'mainstream thought', through which qbits cannot converge to the function's minimum independently, but are instead forced to account for the locations of other qbits in the swarm. Defining $pbest_{(i,j)}(t)$ to be the j-th entry in $pbest_i(t)$, equation (6) defines the 'mainstream thought point':

$$mbest(t) = \left(\frac{1}{N_p}\sum_{i=1}^{N_p}\left\{pbest_{(i,1)}(t)\right\},...,\frac{1}{N_p}\sum_{i=1}^{N_p}\left\{pbest_{(i,D)}(t)\right\}\right)^T \tag{6}$$

Intuitively, each entry in the *mbest(t)* vector represents the mean best coordinate in that dimension among all particles. In the improved QPSO algorithm, the local exploration variable is therefore defined to be: $L_i(t) = 2\beta \cdot |mbest(t) - P_i(t)|$. For the remainder of the report, this improvement will be used.

3.3 Comparison to traditional PSO algorithms

The main advantage of QPSO over PSO is its simplicity: whereas in the original algorithm it was necessary to select values for a cognitive parameter c_1, a social parameter c_2 and an inertia weight ω, when using QPSO it is sufficient to choose the value of β. It is difficult to overstate the gains from this simplification. Indeed, some researchers have suggested solving a separate optimization problem to select c_1 and c_2.

The added simplicity of QPSO does not necessarily come at an unreasonable price. When testing QPSO, PSO, and Levy PSO on eight high dimensional test functions Fang et al. (2010) found that QPSO was outperformed by a PSO variant only once, on a Rastrigin-type function $f(\mathbf{x}) = \sum_{i=1}^{\dim \mathbf{x}} x_i \sin\sqrt{|x_i|}$. The idea that simplifying the original PSO algorithm can lead to improved performance is consistent with other recent PSO variants. Kennedy (2003) introduces the bare-bones PSO, in which the velocity and position updating rules are replaced by random draws from a parametric distribution.

4 Practical Considerations When Implementing PSO and QPSO

4.1 Rational for parallelisation

The recent drive to parallelise optimisation algorithms is motivated by the 'death' of Moore's law. Moore's Law is a rule of thumb stating the number of transistors on a chip roughly doubles every two years; accordingly, processor performance has tended to grow at the same exponential rate. Now, this was true for a long time but does not seem to be the case anymore. Making the transistors smaller is not as easy as it once was. By 2020 it is projected that microprocessors will have circuit features that are only 10 atoms in length; at this scale processor performance will become unreliable, as the behaviour of electrons will be governed by quantum mechanics which is inherently uncertain. Additionally, simply making microprocessors larger is not feasible either, as a larger processor increases the likelihood of faults occurring during the manufacturing process which will make the chip non-functional. For example, i5 and i7 processors are in fact the same physical chip, the difference between the two is that i5 chips have minor defects which prevent them from making use of hyperthreading.

As more data becomes available the processing power required to analyse it naturally increases. However, as Moore's law fails the growth in the performance of individual chips beings to slow. This motivates the use of parallel computing. Four processors working at a certain speed can accomplish the same work in the same time as one processor that is four times as powerful according to Waldrop (2016). Say we parallelise a piece of code over 8 CPUs, one CPU acts as a master and the rest as workers. The master distributes tasks and gathers the information needed between iterations, while the workers do the calculations. There are two main methods of doing this: synchronous parallelisation and asynchronous parallelisation. Synchronous tasks are carried out in the same order as assigned, preventing the program from continuing until all tasks are completed. In contrast, asynchronous will do tasks out of order, and will not lock the program. Thus, asynchronous tends to be faster but is problematic if order or communication matter.

4.2 Stopping criteria for SI algorithms

In order to prevent the algorithm from running for longer than necessary, stopping criteria can be used to set an early breaking point. While we still included a maximum number of iterations ($maxIterations$) of 1000 so that the code will end even if the stopping criteria is not met; the use of stopping criteria can greatly reduce the computational time of the algorithm. There are a number of different criteria which are studied in-depth within Zielinski & Laur (2007), of which we will focus on two: distribution-based criteria called $MaxDistQuick$, and improvement-based criteria called $ImpBest$.

Distribution-based criteria focus on how spread-out the various points are according to Zielinski &

Laur (2007). If most of the points are clustered close to each other, then this implies that the algorithm has converged. These types of stopping criteria are generally the most reliable indicator of convergence. $MaxDistQuick$ stops iterations when the best p% particles are close enough because it is assumed that most particles will aggregate when the algorithm has converged. However, in practice, we found this is hard to execute. For some test functions, the stopping criteria did not trigger even after the 10000th iteration, though algorithms had already converged. In other test functions, the criteria were triggered far too early. Figure 4 may explain the first problem associated with the poor performance of $MaxDistQuick$: Even at the end of a large number of iterations of the PSO algorithm, only a small group of particles are around the position of global minimum since others get stuck at local minima. As a result, the distance between most of the particles is still very large.

A different category of stopping criteria called improvement-based criteria is based on a threshold for improvement in the particles. If they improve less than the threshold, then the algorithm ends. Specifically, we used $ImpBest$ which looks at the improvement in the global best value, $f(gbest)$, found so far. If the improvement is less than the threshold for a certain number of rounds, then the algorithm stops. Unfortunately, for evolutionary algorithms such as the PSO, these are considered unreliable. This is because evolutionary algorithms do not tend to improve in a given pattern: they may improve very little, and then suddenly improve a lot. To overcome this, we used a low threshold and a high number of iterations to wait for an improvement. We set the tolerance to be 10^{-9} and the number of iterations for there to be no change better than this to be 20 iterations. In doing so, we found this particular stopping criterion gave the best results; performance of the stopping rule was similar to other successful rules discussed in Zielinski & Laur (2007).

4.3 Keeping particles within the search area

For each function, we are only interested in finding a global minimum within a specified search area. This might be for a number of reasons. Some of the functions cannot be evaluated outside of these bounds. Other functions may have a global minimum outside of the search area which is not of interest to the particular application which we are performing the optimization for. Additionally, we may have prior knowledge about a region where the global minimum occurs, meaning that any searching outside of the area would be a waste of computational power.

However, for both PSO and QPSO, it is possible that some of the particles may still move outside of the search area, even if we initialise all of them within the area. To prevent this from occurring, we implemented a modified method of updating the particle's position. Instead of just moving the particle, we first calculate where it would move to. If the new position is within the bounds, we allow the particle to move there. If the new position is outside of the bounds, we keep the particle at its previous position.

5 Implementing PSO and QPSO in Python

The two algorithms have been implemented in Python which is described below in pseudo-code in Algorithm 1 and Algorithm 2. For the implementation, the suggestions by Zhang et al. (2015) for PSO and Fang et al. (2010) for QPSO were followed. This was done both using parallel computing and non-parallel computing. We chose to parallelise using the CPU on a server hosted by Google Cloud. To parallelise the ‘multiprocesses’ package was used in Python.

5.1 Pseudocode

5.1.1 Pseudocode for PSO implementation

Before we began, we chose the parameter values based on a paper by Jiang et al. (2007). They suggest that the values of $\omega = 0.715$, $c_1 = c_2 = 1.7$ will help increase the explorability and thus help prevent the algorithm from getting caught at local minima. In comparison to some other values which they tested that were based on other papers, these values generally meant that the algorithm took a higher number of iterations to converge, but that it had a higher success rate of finding the global minimum.

For actually implementing PSO, we start by randomly initialising the positions of particles $P_i(0)$ with a uniform distribution across each dimension d over a certain range defined by the bounds of the function. We also define the best location of each particle $pbest_i(t)$ so far (by default the starting position) and calculate the value for each $f(pbest_i(t))$, and define the global best $gbest(t)$ as the position with the lowest value so far and calculate the value of that $f(gbest(t))$. Next we randomly initialise the velocity $V_i(0)$ for the movement of each particle i in each dimension d. This is based on a uniform distribution between the negative range of bounds in that dimension and the positive range of bounds in that dimension.

Next is the algorithm itself. Within each iteration t, we calculate the new velocity for each particle $V_i(t+1)$ and update the particle's new location $P_i(t+1)$, given it stays within the previously defined boundaries. This was done dimension by dimension in our final implementation in Python. Finally, we update our best positions and the best values for each particle ($pbest_i(t)$ and $f(pbest(t))$) and overall ($gbest(t)$ and $f(gbes(t))$). At the end of the iteration, we check if the stopping criterion is met. Once the algorithm finishes, we then return the best position and the minimum of the function. The pseudocode for the implementation can be seen in Algorithm 1.

Algorithm 1 Particle Swarm Optimization

```
1:  procedure PSO(function, numParticles, bounds, parameters, maxIterations)
2:      Initialise $P_{i,d}(0) \sim_{\text{i.i.d.}} \mathscr{U}(lbound_d, upbound_d) \forall i \in 1,...,N_p; d \in 1,...,D$
3:      Initialise $V_{i,d}(0) \sim_{\text{i.i.d.}} \mathscr{U}(-boundrange_d, boundrange_d) \forall i \in 1,...,N_p; d \in 1,...,D$
4:      Initialise $pbest_i(0)$, $f(best_i(0))$, $gbest(0)$, $f(gbest(0))$ as described above
5:      $t \leftarrow 0$                                                    ▷ Iteration counter
6:      while $t < maxIterations$ do
7:          for $i < numParticles$ do
8:              Set $r_1$, $r_2 \sim_{\text{i.i.d.}} \mathscr{U}(0,1)$
9:              Calculate $V_i(t+1)$ using equation 2
10:             if $P_{i,d}(t) + V_{i,d}(t+1)$ is within bounds then
11:                 Set $P_{i,d}(t+1) = P_{i,d}(t) + V_{i,d}(t+1)$
12:             Update $pbest_i(0)$, $f(best_i(0))$, $gbest(0)$, $f(gbest(0))$
13:         $t \leftarrow t+1$
14:         Check stopping criterion
15:     return $gbest(t)$, $f(gbest(t))$
```

5.1.2 Pseudocode for QPSO implementation

Similarly to above, for QPSO we start by randomly initialising the positions of each particle $P_i(0)$ in each dimension d based on a uniform distribution over the boundaries of the function. We also define the best location of each particle $pbest_i(t)$ so far (by default the starting position) and calculate the value for each $f(pbest_i(t))$, and define the global best $gbest(t)$ as the position with the lowest value so far and calculate the value of that $f(gbest_i(t))$. Then we build a proposal matrix, x_i which is initialised as $P_i(0)$. This matrix stores the proposed movement, which is made if it is an improvement from the particles last location.

For each iteration t of the algorithm, we start by calculating the mean of best position coordinates $mbest$ for each dimension d, and also calculating the β value. β decides the distance of the movement. We set our β as suggested by Fang et al. (2010): β linearly decreases from 1 to 0.5 as the iterations go from 0 to the maximum iterations. We think this is a feasible approach as we start by making large steps and as the iterations continue, we expect to be closer to the global minimum so we can make smaller steps. Now, we move each particle one by one by using the equations in section 3.2 and use that as a proposal $x_i(t+1)$ for the next position $P_i(t+1)$. If this proposed location $x_i(t+1)$ is better then our current location $P_i(t+1)$ and within the bounds, we move our particle. Lastly, we update the particle's best position $pbest_i(t)$ and global best position $gbest(t)$ and the corresponding values $f(pbest_i(t))$ and $f(gbest_i(t))$. At the end of the iteration, we check if the stopping criterion has been met. Once the algorithm finishes, we then return the best position and the minimum of the function. The pseudocode for the implementation can be seen in Algorithm 2.

Algorithm 2 Quantum Particle Swarm Optimization

```
1: procedure QPSO(function, numParticles, boundaries and maxIterations)
2:     Initialise $P_{i,d}(0) \sim_{\text{i.i.d.}} \mathscr{U}(lbound_d, upbound_d) \forall i \in 1,...,N_p; d \in 1,...,D$
3:     Initialise $pbest_i(0)$, $f(best_i(0))$, $gbest(0)$, $f(gbest(0))$ as described above
4:     Set $x_i$ as $P_i(0)$                         ▷ This is a proposal of a change to P
5:     $t \leftarrow 0$                                        ▷ Iteration counter
6:     while $t < maxIterations$ do
7:         Calculate $mbest(t)$ and $\beta$
8:         for $i < numParticles$ do
9:             Calculate $\psi_i$, $u_i \sim_{\text{i.i.d.}} \mathscr{U}(0,1)$ and $C_i$ as described above
10:            Set $x_i$ as the result of equation 3
11:            if $f(x_i) < f(P_i(t))$ then
12:                Set $P_i(t+1)$ as $x_{ij}$
13:                Update $pbest_i(0)$, $f(best_i(0))$, $gbest(0)$, $f(gbest(0))$
14:        $t \leftarrow t+1$
15:        Check stopping criterion
16:    return $gbest(t-1)$, $f(gbest(t-1))$
```

5.2 Parallelising

First, we wrote a function that performs all the updates for each particle. Next, we had to reformat the data into lists where each entry was a sub-list of all the information needed to update one individual particle. Then, the computation of each particle is divided amongst the workers which map the described function to each particle using 'starmap' from the 'multiprocessing' package. Since the workers do not communicate, then unlike in the non-parallelised versions, there will be no updates between particles. Instead, at the end of the iteration, the master will gather all the data and we can then update the global best position and value as well as the best position and value of each particle. Thus, instead of updating the global best position and value after every particle, these are updated only at the end of each iteration. Whether this will actually save substantial amount of time depends on the number of particles, dimensions, and more and will be discussed later in section 7.

6 Results from Computational Studies

6.1 Setup

To asses the relative performance of PSO and QPSO, as well as the benefits (or otherwise) of parallelisation, we tested Python code written according to section 5 on a battery of single objective test functions. In total 18 test functions were chosen from Wikipedia (2004). These included the sphere function ($f : \mathbb{R}^4 \to \mathbb{R}$) as a benchmark, as well as high dimensional functions and functions with many local minima. Each function was optimised 50 times using PSO, Parallel PSO, QPSO, and Parallel QPSO; appendix C summarises the average time to convergence, average runtime, mean absolute error (MAE), and median absolute error (MedianAE) for each function and optimisation method. In addition, appendix A shows the convergence behaviour (number of iterations plotted against absolute error) of a single run of PSO and QPSO for each function.

6.2 Environment

To avoid test results being contaminated by processes running in the background of local machines, computational studies were carried out on a Google Cloud instance. The instance was equipped with an eight-core Intel Xeon E5 v4 CPU with a base clock rate of 2.2 GHz. The instance had 32 GB of RAM, hence memory allocation was not problematic. Google Cloud was chosen as it offers the same infrastructure used by Google and YouTube, so we could expect the instance to perform reliably. The consistent performance was particularly important, as the test code took around 20 minutes to run.

6.3 Results

6.3.1 PSO versus QPSO

QPSO outperformed PSO both in terms of algorithm speed and mean absolute error (MAE) across the majority of functions. While the PSO algorithm took on average 0.27 second to run across the 18 test functions, QPSO took only 0.16 seconds. The faster run time is largely due to QPSO converging faster to the function's optimum, and hence triggering the stopping criteria after a lower number of iterations; this is particularly apparent from the plots in appendix A. For example, the PSO algorithm ran on average for 150 iterations across all functions, whereas QPSO ran for only 90. PSO's slower convergence may be due to our parameter choices: we set $c_1 = c_2 = 1.7$ and $\omega = 0.715$ as recommended by Jiang et al. (2007), however in practice these parameter should be chosen according to the properties of the function. Meanwhile, in the QPSO algorithm β was allowed to vary as the number of iterations increased, which may make the method more flexible.

6.3.2 Effects of parallelisation

Both algorithms were slower when parallelised and, unsurprisingly, the parallelised version of QPSO continued to be faster and more accurate than the parallelised version of PSO. While the point of parallelisation is to reduce time, we were parallelising very few particles in a low number of dimensions. The parallelised implementation of PSO and QPSO involve very large communication overheads. We parallelised 50 particles over eight cores with one core acting as the master, hence every worker was responsible for moving around 7 particles. Moving each particle in a low number of dimensions is relatively cheap, and so moving 50 particles could be done quickly by our sequential code. However, when running in parallel, between each iteration, workers were required to send seven potentially large numbers back to the master which is expensive. In addition, in order to run the parallelised algorithm, we had to reformat the input and results every time which would add to the computation time. These increases in time would be compounded with every iteration. It is possible that if we optimised a function in an extremely high number of dimensions which needing more calculations, and which would require a higher number of particles, then the overhead communication costs and time-cost of reformatting the data would no longer be as significant in comparison to the time it takes to update the particles. In this case, the parallelised code might actually help decrease the run-time compared to the non-parallelised versions.

About half of the parallelised versions of the algorithms took a higher number of iterations compared to their non-parallelised versions, and the other half took a fewer number of iterations. The lack of difference in the number of iterations between the parallelised and non-parallelised versions implies that the decreased frequency of communication between the particles does not appear to greatly affect the algorithm.

6.3.3 Simple functions

The algorithm ranking described in 6.3.1 and 6.3.2 held true for our test function. Although both PSO and QPSO found the minimum, the QPSO algorithm was around 20% faster. Consistent with 6.3.2, both algorithms took twice as long to run when parallelised. Interestingly, both PSO and QPSO required more iterations to find the minimum of the sphere function than to find the minimum of some of the more complicated low dimensional functions. For example, QPSO required half as many iterations to converge to the minimum of Schaffer N.2 as it did to converge to the minimum of the sphere function. Schaffer N.2 is supposedly difficult to minimise as the global minimum is located very close to the local minima, see Jamil & Yang (2013). This suggests that the dimensionality of the function may pose a greater constraint than the number of local minima for PSO-type algorithms.

6.3.4 High dimensional functions

Other than the sphere function, we tested our code on three high dimensional function: the Rastrigin ($f : \mathbb{R}^8 \to \mathbb{R}$), Rosenbrock ($f : \mathbb{R}^4 \to \mathbb{R}$), and Styblinski-Tang ($f : \mathbb{R}^6 \to \mathbb{R}$) functions. These functions were the slowest to run and often did not converge. To make convergence results for these functions robust to outliers we introduced the median absolute error (MedianAE) to the table in appendix C. The MedianAE shows that the performance of PSO and QPSO can be erratic for high dimensional functions. For example, with the Styblinski-Tang function, the PSO algorithm had an MAE of $1.19\text{e}+00$ and a MedianAE of $6.71\text{e}-10$, indicating that the algorithm occasionally converged to the wrong value but in general performed well.

The poor performance of PSO and QPSO in high dimensions is in part to be expected as the number of 'wrong' directions in which a particle can move increase exponentially with the dimensionality of the problem. For example, when minimising a function $f : \mathbb{R} \to \mathbb{R}$ particles choose between moving (left) and (right). When minimising a function $f : \mathbb{R}^2 \to \mathbb{R}$ particle choose between (right, up), (right, down), (left, up), and (left, down). In general, for a function $f : \mathbb{R}^D \to \mathbb{R}$ particles make 2^D such choices.

Comparisons of PSO and QPSO may be contaminated by the fact that our stopping criteria performed poorly with high dimensional functions. For example, in appendix A we see that QPSO was able to find the minimum of the Rastrigin function, but needed over 100 iterations to do so. From appendix C we can see that on average the algorithm was stopped around the 50th iteration, and as a result, these simulations never converged. Although PSO does not find the minimum of the Rastrigin function in appendix A, the plots were generated over a maximum of 250 iterations so it is not clear if given more time the algorithm would converge.

Finally, we note that the difference in run time between parallel and non-parallel implementations for high dimensional functions was smaller than for low dimensional functions. This is likely because the factors discussed in section 6.3.2 were outweighed by the computational costs of evaluating a high dimensional function.

6.3.5 Functions with many local minima

To examine the differences between functions with many local minima and those without, we will focus on functions in 2 dimensions. Test functions with many local minima are more difficult to achieve convergence because particles get stuck in local minima. For example, the Eggholder and Bukin N.6 functions both had a much MAE and MedianAE.

One thing to note is that, when we were doing 50 simulations of the Eggholder function, only one simulation converged to within 0.1 of the true global minimum for each PSO, Parallel PSO, and Parallel QPSO each, and no simulations converged to within 0.1 of the true global minimum for QPSO. As figure 4 is shown, there are many local minima in Eggholder function, and many particles get stuck around those positions.

Meanwhile, we found good convergence properties and low run times for functions with few local minima but additional features which supposedly made the functions difficult to optimise. For example, the Goldstein–Price function is supposedly difficult to optimise as scaling problems result from differences in orders of magnitude between the domain and the function hyper-surface. Nonetheless, both PSO and QPSO found the function's minimum in around 100 iterations.

7 Conclusion

Overall, we found that QPSO consistently outperformed PSO, and parallel implementations of both algorithm ran slower than non-parallel implementations. In section 6.3.2, we propose that this may be a result of high communication overheads, and speculate that benefits from parallelisation will be seen only when the dimensions of the function being minimised are large and when we use a large number of particles. Indeed, in Kaur et al. (2016) PSO implemented in parallel is only compared with PSO implemented sequentially for 32 and 64-dimensional functions. Additionally, since the communication between particles within an iteration did not appear to affect the convergence, it might be interesting to try the asynchronous method of parallelisation which would have even less communication and should run faster than the synchronous version. Venter & Sobieszczanski-Sobieski (2006), for example, found their asynchronous implementation of PSO to be significantly faster and did not experience a significant decrease in numerical accuracy.

As mentioned in section 5, we set parameters in PSO and QPSO as specified in Jiang et al. (2007) and Fang et al. (2010) respectively, and did not change the parameters to match the properties of the function being optimised. The better performance of QPSO suggests that the algorithm is better for 'black box' optimisation where the user is not required to set parameter values. This, however, does not rule out the possibility that PSO may outperform QPSO when c_1, c_2, and ω are chosen more carefully. It would be interesting to look at other variants of the PSO, and compare all of them against each other; in particular, the performance of QPSO respective other simplifications of the original PSO algorithm, such as bare bones PSO.

References

Born, M. (1927), 'Physical aspects of quantum mechanics'.

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.

Eberhart, R. & Kennedy, J. (1995), A new optimizer using particle swarm theory, *in* 'MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science', Ieee, pp. 39–43.

Fang, W., Sun, J., Ding, Y., Wu, X. & Xu, W. (2010), 'A review of quantum-behaved particle swarm optimization', *IETE Technical Review* **27**(4), 336–348.

Heisenberg, W. (1985), Über den anschaulichen inhalt der quantentheoretischen kinematik und mechanik, *in* 'Original Scientific Papers Wissenschaftliche Originalarbeiten', Springer, pp. 478–504.

Jamil, M. & Yang, X.-S. (2013), 'A literature survey of benchmark functions for global optimization problems', *arXiv preprint arXiv:1308.4008* .

Jiang, M., Luo, Y. P. & Yang, S. Y. (2007), Particle swarm optimization-stochastic trajectory analysis and parameter selection, *in* 'Swarm intelligence, Focus on ant and particle swarm optimization', InTech.

Kaur, J., Singh, S. & Singh, S. (2016), Parallel implementation of pso algorithm using gpgpu, *in* '2016 Second International Conference on Computational Intelligence & Communication Technology (CICT)', IEEE, pp. 155–159.

Kennedy, J. (2003), Bare bones particle swarms, *in* 'Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No. 03EX706)', IEEE, pp. 80–87.

Mandal, P., Zareipour, H. & Rosehart, W. D. (2014), 'Forecasting aggregated wind power production of multiple wind farms using hybrid wavelet-pso-nns', *International journal of energy research* **38**(13), 1654–1666.

Mirjalili, S. & Lewis, A. (2016), 'The whale optimization algorithm', *Advances in engineering software* **95**, 51–67.

Reynolds, C. W. (1987), Flocks, herds and schools: A distributed behavioral model, *in* 'ACM SIGGRAPH computer graphics', Vol. 21, ACM, pp. 25–34.

Saraswathi, S., Sundaram, S., Sundararajan, N., Zimmermann, M. & Nilsen-Hamilton, M. (2011), 'Icga-pso-elm approach for accurate multiclass cancer classification resulting in reduced gene sets in which genes encoding secreted proteins are highly represented', *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)* **8**(2), 452–463.

Sun, J., Feng, B. & Xu, W. (2004), Particle swarm optimization with particles having quantum behavior, *in* 'Proceedings of the 2004 congress on evolutionary computation (IEEE Cat. No. 04TH8753)', Vol. 1, IEEE, pp. 325–331.

Sun, J., Xu, W. & Feng, B. (2004), A global search strategy of quantum-behaved particle swarm optimization, *in* 'IEEE Conference on Cybernetics and Intelligent Systems, 2004.', Vol. 1, IEEE, pp. 111–116.

Venter, G. & Sobieszczanski-Sobieski, J. (2006), 'Parallel particle swarm optimization algorithm accelerated by asynchronous evaluations', *Journal of Aerospace Computing, Information, and Communication* **3**(3), 123–137.

Waldrop, M. M. (2016), 'The chips are down for moore's law', *Nature News* **530**(7589), 144.

Wikipedia (2004), 'Test functions for optimization'.
URL: *https://en.wikipedia.org/wiki/Test_functions_for_optimization*

Yang, C., Tu, X. & Chen, J. (2007), Algorithm of marriage in honey bees optimization based on the wolf pack search, *in* 'The 2007 International Conference on Intelligent Pervasive Computing (IPC 2007)', IEEE, pp. 462–467.

Zhang, Y., Wang, S. & Ji, G. (2015), 'A comprehensive survey on particle swarm optimization algorithm and its applications', *Mathematical Problems in Engineering* **2015**.

Zielinski, K. & Laur, R. (2007), 'Stopping criteria for a constrained single-objective particle swarm optimization algorithm', *Informatica* **31**(1).

A Convergence Plots

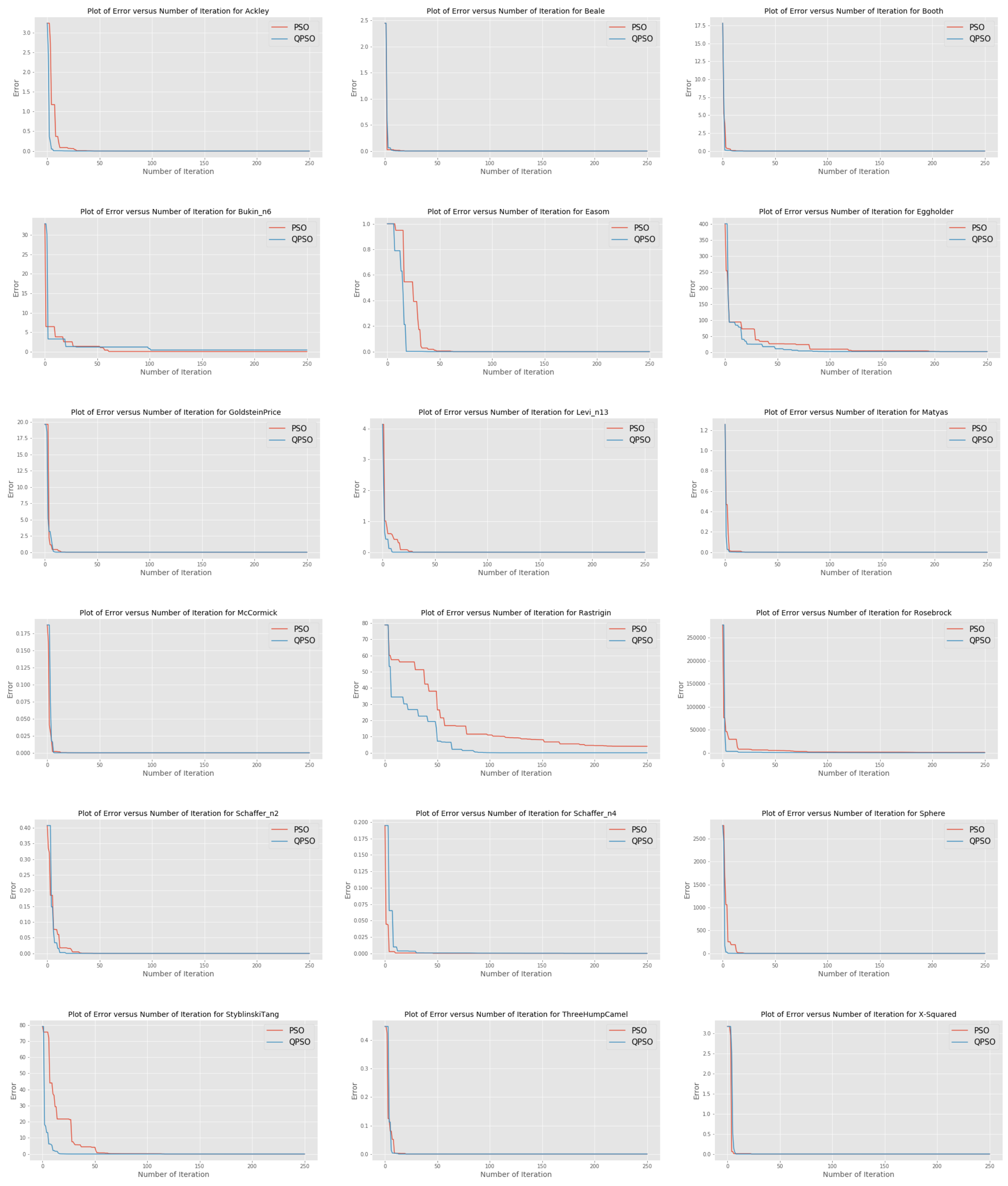

Figure 3: Plot absolute errors vs. number of iteration for all 18 different test functions

B Contour Plots

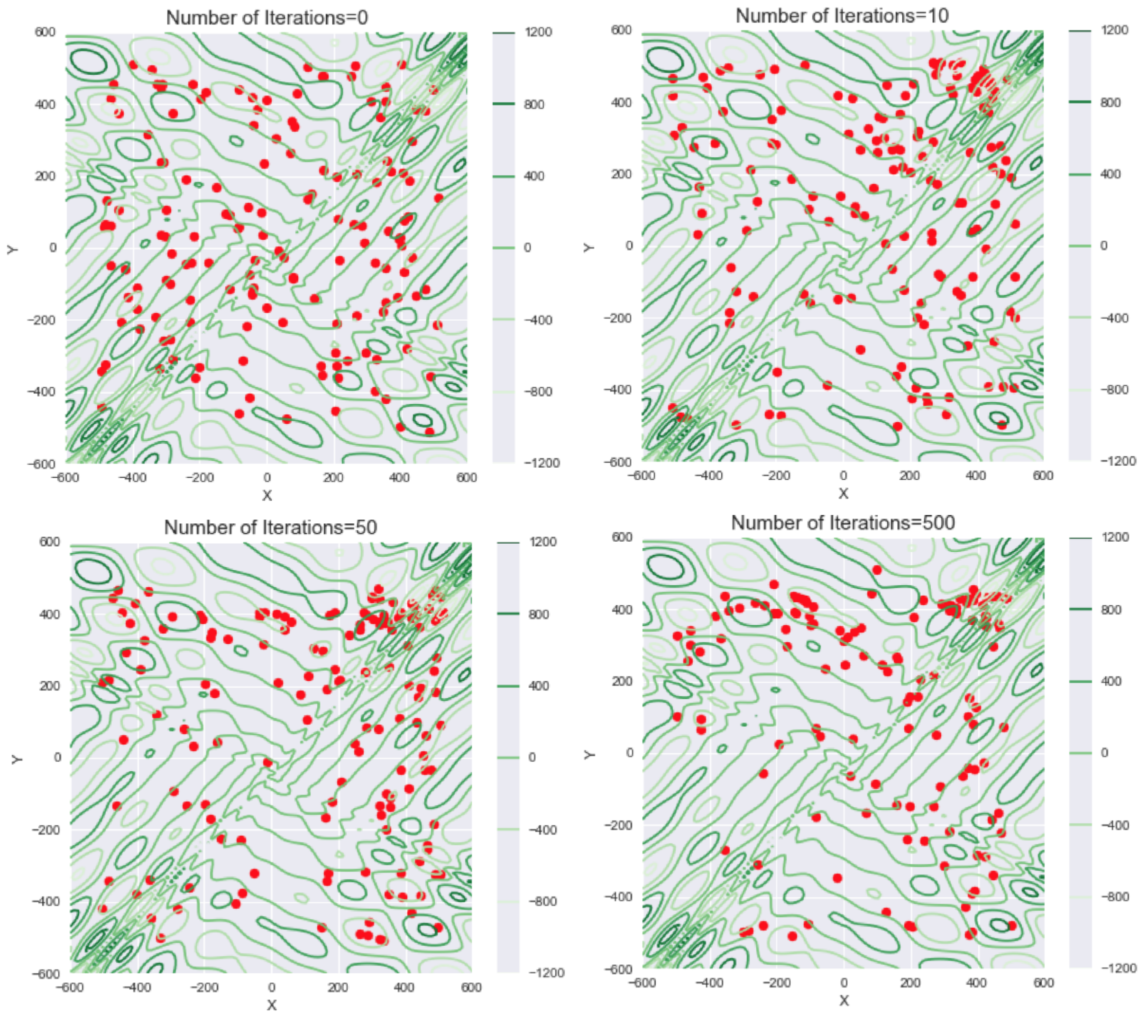

Figure 4: How PSO particles move for the Eggholder function. Most end in global minimum (512, 404.2319), though some get stuck in local minima.

C All Results for Test Functions

Function	Search Area	Method	Time (sec)	Iterations	MAE	MedianAE
Ackley	$[-5,5]^2$	PSO	0.25817	155.52	1.19e-03	2.62e-09
		PSO_Par	0.66714	184.46	3.46e-06	7.03e-10
		QPSO	0.20031	123.20	7.46e-05	1.88e-10
		QPSO_Par	0.47454	124.60	2.62e-05	3.61e-10
Beale	$[-4.5,4.5]^2$	PSO	0.11982	108.68	3.61e-05	2.56e-10
		PSO_Par	0.37632	106.38	2.04e-04	1.31e-10
		QPSO	0.08495	74.06	6.99e-06	5.20e-08
		QPSO_Par	0.28642	76.54	1.24e-05	2.99e-08
Booth	$[-10,10]^2$	PSO	0.11161	102.94	7.47e-05	6.35e-11
		PSO_Par	0.34759	97.86	7.25e-05	1.32e-10
		QPSO	0.08983	82.18	2.52e-07	4.76e-11
		QPSO_Par	0.28652	76.86	8.99e-07	7.42e-11
Bukin N.6	$[-15,-5]\times[-3,3]$	PSO	0.05480	49.00	1.61e+00	1.27e+00
		PSO_Par	0.19416	53.64	1.32e+00	9.74e-01
		QPSO	0.05617	48.74	8.09e-01	6.71e-01
		QPSO_Par	0.17140	44.94	1.09e+00	1.01e+00
Easom	$[-100,100]^2$	PSO	0.15224	116.86	2.81e-03	1.43e-10
		PSO_Par	0.39783	111.50	1.46e-02	1.04e-10
		QPSO	0.08536	64.38	1.32e-02	1.62e-03
		QPSO_Par	0.24504	65.32	1.06e-02	1.42e-03
Eggholder	$[-512,512]^2$	PSO	0.11872	83.76	3.98e+01	2.43e+01
		PSO_Par	0.38124	106.68	3.45e+01	2.29e+01
		QPSO	0.07155	50.30	3.85e+01	2.06e+01
		QPSO_Par	0.19464	51.58	3.90e+01	1.31e+01
Goldstein-Price	$[-2,2]^2$	PSO	0.16096	107.12	2.09e-05	5.56e-11
		PSO_Par	0.37584	104.48	2.00e-04	6.07e-11
		QPSO	0.12858	83.82	9.73e-05	1.38e-11

		QPSO_Par	0.32609	86.14	1.33e-04	1.37e-11
Lévi N.13	$[-10,10]^2$	PSO	0.16443	107.16	7.93e-04	3.96e-11
		PSO_Par	0.35333	97.46	2.45e-03	5.84e-11
		QPSO	0.12221	82.32	3.45e-07	3.82e-11
		QPSO_Par	0.32296	84.48	6.31e-07	2.08e-11
Matyas	$[-10,10]^2$	PSO	0.09028	93.26	7.15e-06	5.61e-11
		PSO_Par	0.31370	88.54	4.34e-05	1.05e-10
		QPSO	0.07871	77.86	7.50e-08	5.37e-11
		QPSO_Par	0.26145	69.80	2.42e-07	4.12e-10
McCormick	$[-1.5,4]\times[-3,4]$	PSO	0.10368	90.70	8.08e-08	9.15e-11
		PSO_Par	0.32787	91.90	7.32e-08	9.13e-11
		QPSO	0.08205	69.62	1.64e-07	9.26e-11
		QPSO_Par	0.26592	70.78	7.78e-08	9.60e-11
Rastrigin	$[-5.12,5.12]^8$	PSO	0.59357	178.18	2.02e+01	1.86e+01
		PSO_Par	0.86057	226.68	1.92e+01	1.29e+01
		QPSO	0.19879	53.64	2.56e+01	2.64e+01
		QPSO_Par	0.20071	49.62	2.48e+01	2.46e+01
Rosenbrock	$[-100,100]^4$	PSO	1.48689	725.16	9.01e+02	1.84e+00
		PSO_Par	2.63585	725.22	8.43e+02	2.10e+00
		QPSO	0.50226	222.38	2.97e+02	7.15e+01
		QPSO_Par	0.82647	212.02	5.61e+02	3.38e+02
Schaffer N.2	$[-100,100]^2$	PSO	0.11011	87.60	5.99e-04	6.85e-11
		PSO_Par	0.32168	89.68	5.08e-04	8.44e-11
		QPSO	0.09665	78.74	4.75e-04	1.22e-08
		QPSO_Par	0.29042	76.72	8.02e-06	2.36e-08
Schaffer N.4	$[-100,100]^2$	PSO	0.10475	78.22	5.37e-04	7.89e-05
		PSO_Par	0.30108	84.80	4.81e-04	7.03e-05
		QPSO	0.08083	61.64	2.91e-04	6.14e-05
		QPSO_Par	0.23107	61.64	1.92e-04	9.30e-05
Sphere	$[-100,100]^4$	PSO	0.31068	175.84	8.19e-02	1.22e-10

		PSO_Par	0.66233	181.40	7.20e-03	7.03e-11
		QPSO	0.26443	138.42	1.79e-05	7.49e-11
		QPSO_Par	0.53153	137.68	1.05e-04	6.01e-11
Styblinski-Tang	$[-5,5]^6$	PSO	0.81072	292.60	1.19e+00	6.71e-10
		PSO_Par	0.92722	250.26	3.53e+00	1.04e-09
		QPSO	0.64136	204.48	1.30e-06	1.39e-10
		QPSO_Par	0.76121	194.58	7.99e-01	6.59e-11
Three-hump camel	$[-5,5]^2$	PSO	0.09685	86.72	1.01e-04	6.37e-11
		PSO_Par	0.33401	92.86	4.69e-05	3.05e-11
		QPSO	0.07872	67.88	1.82e-07	4.38e-11
		QPSO_Par	0.25024	66.02	1.88e-07	5.87e-11
X-Squared	$[-200,200]^2$	PSO	0.03820	67.26	1.11e-04	9.16e-10
		PSO_Par	0.25341	73.28	1.29e-04	8.98e-11
		QPSO	0.03044	53.80	7.46e-09	7.43e-13
		QPSO_Par	0.19419	53.46	1.23e-09	3.02e-13