Spring 5-25-2021

Can Parallel Gravitational Search Algorithm Effectively Choose Parameters for Photovoltaic Cell Current Voltage Characteristics?

Alan Kirkpatrick
alanmk@bgsu.edu

Can Parallel Gravitational Search Algorithm Effectively Choose Parameters for Photovoltaic Cell Current Voltage Characteristics?

Alan Kirkpatrick

Introduction

Sustainability and renewable energy are becoming more and more prevalent in today's world. This will only become truer as electric cars replace gasoline, the population increases, and climate change increases the demand for heating and cooling capabilities. One of the alternatives to fossil fuels is solar power. Photovoltaic (PV) cells take the photons emitted from the sun and convert them into electricity.

PV cells have several characteristics that depend on the parameters of the cell. These parameters are used to determine the Current Voltage (IV) curve. This curve relates the current and voltage at various real-world conditions such as temperature and light levels. IV curves provide the essential performance metrics of PV modules, including power conversion efficiency, maximum power output, open circuit voltage, short circuit current density, and fill factor.

The curve lets consumers apply real world conditions to solar cells to get an idea of the true performance for the given application. This performance can then be used to determine the best location, angle, and possibly active motion of the panel if used in the setup. All of these give greater efficiency to the panel and allow for more electricity to be generated for a given area.

This study asks the question "Can parallel Gravitational Search Algorithm (GSA) effectively choose parameters for photovoltaic cell current voltage characteristics?" These parameters will be plugged into the Single Diode Model to create the IV curve. It will also investigate Particle Swarm Optimization (PSO) and a population based random search (PBRS) to see if GSA performs the search better and or more quickly than alternative algorithms.

Previous research into this topic has used several algorithms to solve the problem, such as GSA and PSO. These papers have shown the algorithms are effective in finding an optimal solution. Fitness scores are determined using Root Mean Squared Error (RMSE).

Photovoltaic Effect

The primary way solar cells generate electricity is though the Photovoltaic effect. When a photon strikes the cell, an electron absorbs the photon. This photon is then excited to a higher energy level, leaving a positive "hole" behind [1]. Normally, these would recombine fairly quickly, resulting in no charge gained. In order to create current, there needs to be a separation of the opposite charges. This separation is known as the potential barrier.

The potential barrier is a division of charges. This barrier separates the negative electrons from the positive holes. To create this barrier, pure silicon materials must be doped with impurity atoms with one more electrons. This is known as a n-type silicon with the majority charge carriers being electrons. If doped with impurity atoms that have one less electron than Si atoms, then p-type silicon is formed, and the majority charge carriers are holes. At the junction between these silicon's, the electrons can freely jump from the n-type to the p-type [1]. Eventually enough charges will have moved to where the potentials will be equal, and no more charges will flow. This creates the potential barrier.

When light strikes the solar cell, if the wavelength is correct, an electron will absorb the energy and move to a higher energy level, creating an electron-hole pair. Depending on which side of the potential barrier the pair is created, the electron and hole will separate, with one of them moving across the potential barrier and the other being repelled. Now there is a pair that are

unable to recombine, until an external load is placed, creating an electric current, completing the PV cell's generation of electricity [1].

In the single diode PV model, as shown in Fig. 1, there are 5 different parameters that determine the relationship between voltage and current [2]. There are two currents, photocurrent (I_{PH}) and reverse saturation current (I_O). There are also two resistors added to the model, resistive losses due to metals natural resistance (R_S) and shunt resistive losses (R_{SH}). Note that this model is an idealized one, and these resistors only sum up the resistance of other components, they are not themselves parts of the cell. Finally, we have the diode ideality factor (n) which is the relationship between the diode and an ideal diode. It is these 5 parameters that will be estimated.

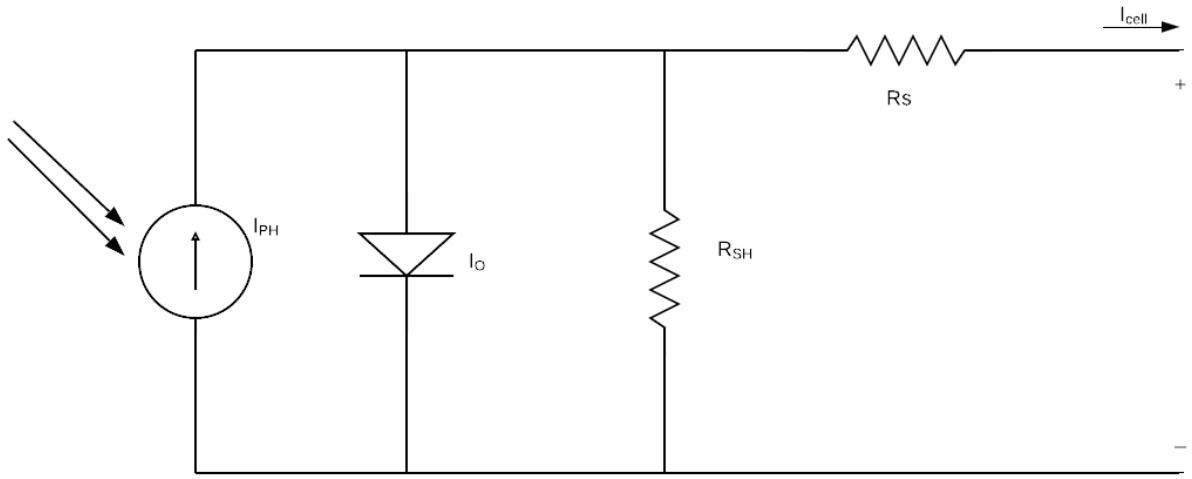

Fig. 1. Model of an idealized photovoltaic cell using the single diode model.

The IV curve is the relationship between the voltage and current of a cell. This relationship is complex and depends on many factors, such as those in the idealized model above. The standard equation is a transcendental, meaning it cannot be solved in a closed form. This equation is given as

$$I_{cell} = I_{ph} - I_O * [e^{\frac{q(V_{cell}+I_{cell}*R_S)}{n*k*T_C}} - 1] - \frac{V_{cell}+I_{cell}*R_S}{R_p} \quad (1)$$

where k is the number of cells and T_c is the temperature in Celsius. This equation is known as the modified single-diode model and is what will be used to solve for the current.

Metaheuristics

The chosen algorithm for this problem will be GSA [3]. GSA is inspired by Newton's laws and the interaction of mass. Each particle has a set mass, with a higher mass equaling a more optimum solution. A single particle is one estimation of parameters, which each dimension being one variable that is being optimized. As opposed to traditional 3-D space, there is d dimensions, which is dependent on the problem. Each mass moves throughout the search space looking for the optimum solution to the objective function. So, in the case of single cell PV estimation, there will be five dimensions to be estimated.

At each iteration in GSA a set of steps are taken. After the initial random distribution of particles, the masses of each are calculated. Then the gravitational constant G(t) is updated. Next the forces must be calculated between each mass, with a certain number of bodies (k) being allowed to create force between the others. Then the acceleration and velocity are calculated for each mass. Finally, the positions are updated according to the velocity. This process then repeats until a mass reaches the desired solution [4].

In addition to GSA, there are numerous other metaheuristic algorithms that could be used to estimate the PV cell parameters. Some have separated these algorithms into four distinct groups, Evolutionary, Physics based, Swarm based, and Human based. I will be discussing the algorithms used in Olivia's article, "A review on meta-heuristics methods for estimating parameters of solar cells [5]." Since the optimization problem they solve is the same as this study,

this will serve as an excellent comparison to test both the accuracy and reliability of my algorithm compared to their results.

The first category of algorithms is Evolutionary algorithms. These follow the process of evolution in which the population of samples gradually works their way towards the optimum solution. Examples of evolutionary algorithms include Genetic algorithm (GA), Differential evolution (DE), and Shuffled complex evolution (SCE). GA is modeled after evolution in nature, where the fittest individuals have a better chance to survive and reproduce. Some also undergo random mutations to prevent the population from being stuck in a local minim. DE also uses crossover and mutation. However, it can also create additional trial vector comparing two individuals [5]. This lets DE self-organize which GA cannot do. Finally, SCE uses a controlled random search combined with shuffling. They also improved upon the traditional SCE by added a few specifics to the solar cell problem, such as reflection absorbing bound handling.

Physics based algorithms are generally inspired by physical properties observed in the world. These include Wind driven optimization (WDO), Flower pollination algorithm (FPA), and Gravitational search algorithm (GSA). WDO simulates newtons 2^{nd} law in air particles. It also includes gravity, friction, and the Coriolis force all working together to find the optimum solution [5]. FPA follows the pollination of flowers using both biotic and abiotic transfer methods of pollen. This functions by allowing the fittest plants to survive, i.e. those that are able to spread their pollen. GSA was previously discussed above.

Then, we have Swarm based algorithms. These involves many particles all affecting each other following a set algorithm. Artificial bee colony (ABC), Particle swarm optimization (PSO), Cat swarm optimization (CSO), and Whale optimization algorithm (WOA) are all examples of a swarm algorithm. ABC simulates the process by which honeybees collect food. This depends on

the location of the food, the amount of nectar, and the classes of bee. The bees search and move towards the highest amount of nectar for a given food location. PSO involves a random displacement of particles with position and velocity. The particles influence each other based on their fitness, which the swarm gradually coalescing at the optimum solution [5, p. 6]. CSO involves a random displacement of cats which move in 2 modes, tracking and seeking. Seeking the cats move slowly where tracking they move fast towards a new location. These combine to locate the optimum solution. WOA simulates whale hunting and circling their prey. A ring of bubbles is creating which the other whales move to as it slowly contracts. These two phases are the exploration and exploitation phases.

Finally, we have one Human based algorithm, Harmony search (HS). HS is based on musicians improvising music that is pleasing to hear. In place of improvisation, the variables are modified by the algorithm to minimize the objective function. There are two different types, grouping-based global harmony search (GGHS) and innovative global harmony search (IGHS). Both are designed to improve performance and to avoid local minima.

Parallelization

Parallelization involves taking code, which typically executes in a single thread, procedurally and allows it to take advantage of multiple cores of a CPU or even multiple CPUs. There are two main types. The first is shared memory. This is where a program is split up in a way to take utilize more than 1 core of a single CPU that shares memory. One of the most common methods in C++ is using a library called OpenMP [6]. Conversely, distributed memory can use more than one CPU and memory stores that are physically distinct. This is done with a

library like MPI [7]. It is possible to combine OpenMP and MPI to enable parallel execution on both multiple cores and multiple CPUs.

In both cases the library handles the low-level commands required, but it is up to the developer to split the workload in a way that avoids the most common problems. One of these problems is a race condition, where two threads or processes attempt to access the same location at the same time for writing. This problem can be eliminated by setting locks on data that must be accessed by the same threads, or by splitting the data in such a way that each thread can access its own copy of the data, which can then be recombined when the program finishes its parallel processing.

The better the code is split up the faster the execution time. A perfectly parallel algorithm can be split among n threads, and the execution time will be 1/n. However, most code will have a less than perfect split, either due to the data size or due to the calculations that must be done. This leads to the speedup and efficiency metrics. Speedup (`S = single-thread time / multi-thread time`) is how much faster the parallel version is compared to the single version. Efficiency is speedup over the number of threads (E = `S/n`), showing how well the code is parallelized. The higher efficiency the better scaling can be obtained with more cores.

When creating parallel code, there will be a slight overhead, as the threads must be created, private variables must be copied, and work must be distributed. This is more noticeable when threads are repeatedly created and destroyed. This is why the code only creates threads once and reuses them for the duration of the program. Another side effect of the overhead is the single thread time will be slightly slower than if the code was not parallel, as the omp statements must still execute and a single thread is still created.

Methodology

In order to solve for the optimal parameters of the PV cell, it is necessary to develop an algorithm capable of determining them in an efficient and accurate manner. For this reason, this study uses the GSA, PSO, and a random search. The first two are optimization algorithms designed to solve problems like this, while the random is the generation of random solutions. If the algorithms do not perform as well or better than the random then there is no need for an advanced algorithm to determine the parameters.

The program will consist of several parts. The first part will consist of the implementation of the three algorithms. Both GSA and PSO will be standard implementations with no modifications made. The random algorithm will also be standard as it is just a repeated random selection of values.

The only change made to all three is that they will be parallelized with OpenMP. This will allow some sections of the code to run in parallel, increasing the performance and decreasing the time it takes to get a result. All tests will be run with up to 16 cores on a Ryzen 5950X overclocked to 4.575 Ghz. The specific parallel improvements will be discussed further below.

The second part will be the timing code, which will record the execution time of both the whole program, and individual pieces to get an accurate idea of how well the speedup and efficiency of the parallelization. This is critical as obtaining a high efficiency is just as important as the accuracy of the estimated parameters. If two algorithms have a similar fitness score but one runs much faster, than the faster algorithm is the better choice to use.

The third and final portion of the code will be the fitness function. While much smaller than the other parts this is the most important. The function is what determines how close the parameters are to the accepted values, or the fitness of the data. This is split into two functions. The first will create the IV curve based on the known voltages and estimated parameters. The second will use root mean squared error (RMSE) to find the difference between the estimated and experimental curves.

The program (source code is publicly available here: https://github.com/Darth-Falcon/GSA-solar) is operated with command line arguments. These control which algorithm will be used, the number of particles, time steps, as well as the bounds for all 5 parameters. There is also a parameter to control the number of dimensions, so a fitness function to be created for use in double diode cells. However, there is no way to select the fitness function without modifying the source code. More explanation of the arguments and the programs use will be detailed in the README in the GitHub repository.

The results of this project will be an algorithm capable of estimating the parameters of a solar cell. These parameters will be plugged into the Single Diode Model to create the IV curve. Finally, this curve will be used to estimate the efficiency of solar cells in real world conditions.

Previous research into this have used several algorithms to solve the problem, such as GSA and PSO. These papers have shown the algorithms are effective in finding an optimal solution. This paper will verify those results and also compare them against a random selection, to see if the algorithm is choosing the best values both in terms of computation time and final fitness score.

Discussion of Code

The code was more difficult to create than I had initially anticipated. This is due to two reasons. The first is the fitness function is a transcendental equation, which means the solution is on both sides of the equation. This means that you cannot plug in the numbers to solve like a traditional expression. After trying several functions to solve this including lambert and newtons methods, I settled on a simpler approach. I took the experimental values for current and plugged those into the right-hand side of the equation. Then the left side gives me the estimated current. This meant it could be solved like a normal equation. In testing, it was found that this method combined with the experimental parameters taken from [2] gives a fitness score of 0.028, which is similar to the fitness score they got.

The second issue was with GSA. None of the papers that worked with GSA provided their source code, meaning the algorithm had to be created from scratch, with only their notes on the rough operation to guide me. Despite my efforts at combining the paper where GSA was originally proposed and the papers which used it to solve the solar cell optimization problem, I was not able to create an algorithm which would produce the results as accurate as those papers. This is excellent proof that source code should be made public with the publication of a research paper. No matter what language it was done it, this would allow other developers and researchers to see how it was solved to not only verify the original papers findings, but to improve upon to either achieve the same result in less time, or a more accurate result.

Because of this, I used my PSO algorithm from my parallel programming class two years prior as a second algorithm to try and get better values. I also created a random function, since there is no point in running a complex algorithm if picking random values will get a better fitness score. Each of the 3 algorithms were parallelized with OpenMP. This allows the program to take

advantage of multiple threads on a single CPU. Each algorithm was parallelized slightly differently depending on what could be accessed and modified at the same time. Any data that depends on data from a previous run can't be done in parallel.

The random algorithm was parallelized in two places. The first was the loop which created the random values and the second is the loop which creates the fitness scores. These two for loops are the only parts that can be made to run in parallel. The global best values can only be done by a single thread. PSO was parallelized in a similar manner. The loops which calculate the position, velocity, and fitness were made to run in parallel. Only the calculation of the global and local best wasn't parallelized. Any timing code was also not parallelized as only one thread needs to start or stop the timing code. GSA was parallelized just like PSO, with everything except the global bests being parallel. Each of the parallel statements were accomplished with a `#pragma omp for`.

Any code that was to be executed by only a single thread was given either `#pragma omp single` or `#pragma omp master`. The entire function was created inside a parallel section to prevent the unneeded creation and deletion of threads.

Analysis of Code

Due to time constraints on completing the project, and the sizes of particles/time steps chosen, not all GSA runs were completed. However, there is still enough data to make a conclusion. Five runs were completed for each set of parameters and their averages taken to minimize error. There were 1-16 threads used with 1,000, 5,000, and 10,000 particles, all with 50,000 timesteps. The exception to this was GSA, which was run with only 1,000 particles and 10,000 timesteps, as the only run done with 5,000 particles took 2 hours alone, and I did not have

the time for all 240 total runs at 5,000 particles. Below are the average speedup, efficiency, and karp-flatt metric, as well as the percent difference between the best fitness score for that algorithm.

TABLE I. SPEEDUP OF POPULATION BASED RANDOM SEARCH ALGORITHM

Number of Threads	**Number of Particles**		
	1000	***5000***	***10000***
1	1	1	1
2	1.817303145	1.779622	1.730642
3	2.438534287	2.320151	2.247501
4	2.970637193	2.891261	2.745701
5	3.506201619	3.289285	3.069835
6	3.889482702	3.625732	3.336928
7	4.21192357	3.865711	3.609828
8	4.556579936	4.112855	3.844505
9	4.768173158	4.304044	4.106554
10	5.011439962	4.58945	4.453124
11	5.190987883	4.691483	4.356624
12	5.377941579	4.923639	4.784494
13	5.577509594	5.057029	4.908425
14	5.756648799	5.210942	5.049197
15	5.862041276	5.354303	5.170072
16	6.083612319	5.500277	5.256688

TABLE II. EFFICIENCY OF POPULATION BASED RANDOM SEARCH ALGORITHM

Number of Threads	**Number of Particles**		
	1000	***5000***	***10000***
1	1	1	1
2	0.908652	0.889811	0.865321
3	0.812845	0.773384	0.749167
4	0.742659	0.722815	0.686425
5	0.70124	0.657857	0.613967
6	0.648247	0.604289	0.556155
7	0.601703	0.552244	0.51569
8	0.569572	0.514107	0.480563
9	0.529797	0.478227	0.456284
10	0.501144	0.458945	0.445312
11	0.471908	0.426498	0.396057
12	0.448162	0.410303	0.398708
13	0.429039	0.389002	0.377571
14	0.411189	0.37221	0.360657
15	0.390803	0.356954	0.344671
16	0.380226	0.343767	0.328543

TABLE III. PERCENT DIFFERENCE OF POPULATION BASED RANDOM SEARCH ALGORITHM

Number of Threads	**Number of Particles**		
	1000	***5000***	***10000***
1	27.68226497	12.19841	10.60797
2	26.5275876	12.15053	13.8724
3	28.32353435	10.63226	6.216174
4	28.02310509	22.3116	7.376512
5	27.60274416	12.68505	0
6	28.90496218	11.54703	3.162926
7	28.40686313	13.13106	8.544504
8	24.87118514	19.49323	6.142861
9	22.16427817	17.68206	13.53046
10	25.85680777	10.43002	7.310364
11	18.86207621	18.19691	5.492049
12	23.88818443	12.70775	11.92566
13	26.90566138	18.84172	9.036868
14	25.09795395	16.5662	15.90845
15	20.91298416	16.4841	10.52319
16	24.39565618	8.191426	4.238023

TABLE IV. KARP-FLATT FOR POPULATION BASED RANDOM SEARCH ALGORITHM

Number of Threads	Number of Particles		
	1000	***5000***	***10000***
1	#DIV/0!	#DIV/0!	#DIV/0!
2	0.100532	0.123834	0.155641
3	0.115124	0.14651	0.167408
4	0.115504	0.127826	0.152274
5	0.106511	0.130022	0.157188
6	0.108524	0.130968	0.159612
7	0.110325	0.135132	0.156525
8	0.107958	0.135017	0.154413
9	0.110939	0.136382	0.148952
10	0.110604	0.13099	0.138402
11	0.111906	0.134467	0.152489
12	0.11194	0.130657	0.1371
13	0.110899	0.13089	0.137376
14	0.110152	0.129743	0.136363
15	0.111345	0.128677	0.135808
16	0.108668	0.127263	0.136249

TABLES I-IV show the random result suffers poorly from efficiency. Larger particle sizes reduce the efficiency rather than increase it. This is most likely due to the sorting to find the best fitness taking longer with more particles, and this being a non-parallel section of code. The sort_indexes() function first creates a vector the same size as the number of particles, then sorts that vector based on the fitness vector. Since increasing the number of threads does not decrease the execution time of the function, the efficiency drops off. Same with more particles, it takes longer to create and sort, further decreasing the efficiency. The fitness scores, however, are much better with a larger particle count, with most of the best results being on the 10k particle test. Since in random, the particles are only randomized, having more particles would give a higher probability of having a favorable selection of parameters. It also shows that thread count does not

have an impact on the fitness score, meaning the parallelization did not affect the accuracy of the results.

TABLE V. SPEEDUP OF PARTICLE SWARM OPTIMIZATION

Number of Threads	**Number of Particles**		
	1000	***5000***	***10000***
1	1	1	1
2	1.984906	1.984067	1.973952
3	2.908934	2.909571	2.7787
4	3.791924	3.792419	3.605523
5	4.588503	4.631802	4.333673
6	5.361996	5.471373	5.161054
7	6.038425	6.224631	5.797607
8	6.902542	7.020892	6.598619
9	7.726953	7.690315	7.809394
10	8.280902	8.492211	8.57297
11	9.076593	9.21226	9.420746
12	9.691964	10.04861	10.24597
13	10.35788	10.86712	10.96195
14	11.04018	11.57202	11.53917
15	11.67175	12.31159	12.61004
16	12.34589	13.04898	13.37828

TABLE VI. EFFICIENCY OF PARTICLE SWARM OPTIMIZATION

Number of Threads	**Number of Particles**		
	1000	***5000***	***10000***
1	1	1	1
2	0.992453	0.992034	0.986976
3	0.969645	0.969857	0.926233
4	0.947981	0.948105	0.901381
5	0.917701	0.92636	0.866735
6	0.893666	0.911896	0.860176
7	0.862632	0.889233	0.82823
8	0.862818	0.877612	0.824827
9	0.85855	0.854479	0.86771
10	0.82809	0.849221	0.857297
11	0.825145	0.837478	0.856431
12	0.807664	0.837384	0.853831
13	0.79676	0.835932	0.843227
14	0.788584	0.826573	0.824227
15	0.778116	0.820772	0.84067
16	0.771618	0.815562	0.836143

TABLE VII. PERCENT DIFFERENCE OF PARTICLE SWARM OPTIMIZATION

Number of Threads	Number of Particles		
	1000	*5000*	*10000*
1	27.68226497	12.19841	10.60797
2	26.5275876	12.15053	13.8724
3	28.32353435	10.63226	6.216174
4	27.28132	12.84748	10.98947
5	23.13056	10.84179	1.702854
6	20.65271	12.20088	5.378083
7	19.8418	16.51051	5.537406
8	27.887	12.82123	8.835138
9	22.4317	17.01068	11.56999
10	25.26885	13.72036	0
11	24.7233	11.15957	6.430317
12	22.28931	10.33769	7.79417
13	20.92834	12.24763	9.896238
14	23.51627	6.88614	8.127673
15	20.41172	6.090079	6.294
16	28.67067	15.18492	7.205673

TABLE VIII. KARP-FLATT FOR PARTICLE SWARM OPTIMIZATION

Number of Threads	Number of Particles		
	1000	*5000*	*10000*
1	#DIV/0!	#DIV/0!	#DIV/0!
2	0.007605	0.00803	0.013196
3	0.015653	0.01554	0.039821
4	0.018291	0.018245	0.03647
5	0.02242	0.019873	0.038439
6	0.023797	0.019323	0.032511
7	0.02654	0.020761	0.034566
8	0.022713	0.019922	0.030339
9	0.020594	0.021288	0.019057
10	0.023066	0.019728	0.018495
11	0.021191	0.019406	0.016764
12	0.021649	0.017654	0.015563

Number of Threads	Number of Particles		
	1000	***5000***	***10000***
13	0.021257	0.016356	0.015493
14	0.020623	0.01614	0.016405
15	0.020368	0.015597	0.013538
16	0.019732	0.015077	0.013065

TABLES V – VIII suggest that... with PSO, the algorithm scales much better with the number of threads. It also becomes more efficient with the growing particle count. This is to be expected as most of the code is parallelized and was optimized to be as efficient with multiple threads as possible. The only parts which are not parallelized are quick in terms of execution time compared to the longer parts, meaning overall the efficiency stays quite high, with even 16 threads staying about 75%. Similarly to the random test, more particles yield a better result, as this gives the algorithm more data to work with when locating the optimum solution. The worst solutions were with the smallest number of particles, and the best were with the highest, even with the large number of time steps.

TABLE IX. SPEEDUP OF GRAVITATIONAL SEARCH ALGORITHM

Number of Threads	Number of Particles		
	1000	*5000*	*10000*
1	1	1	
2	2.195564		
3	3.30818		
4	4.402756		
5	5.563386		
6	6.473704		
7	7.448444		
8	8.190317		
9	9.104378		
10	9.889294		
11	10.98151		
12	11.39476		
13	12.24176		
14	13.24038		
15	13.78928		
16	15.89791	9.463927	

TABLE X. EFFICIENCY OF GRAVITATIONAL SEARCH ALGORITHM

TABLE XI.

Number of Threads	**Number of Particles**		
	1000	***5000***	***10000***
1	1	1	
2	1.097782		
3	1.102727		
4	1.100689		
5	1.112677		
6	1.078951		
7	1.064063		
8	1.02379		
9	1.011598		
10	0.988929		
11	0.998319		
12	0.949563		
13	0.941674		
14	0.945741		
15	0.919285		
16	0.993619	0.591495	

TABLE XII. PERCENT DIFFERENCE OF GRAVITATIONAL SEARCH ALGORITHM

Number of Threads	Number of Particles		
	1000	***5000***	***10000***
1	1.913136	6.603032	
2	5.474883		
3	7.69871		
4	3.442456		
5	3.835169		
6	2.678255		
7	3.251674		
8	3.149148		
9	0.944752		
10	4.500195		
11	4.825286		
12	2.763859		
13	2.957494		
14	3.724862		
15	6.070027		
16	15.89975	0	

TABLE XIII. KARP-FLATT FOR GRAVITATIONAL SEARCH ALGORITHM

Number of Threads	Number of Particles		
	1000	*5000*	*10000*
1	#DIV/0!	#DIV/0!	
2	-0.08907		
3	-0.04658		
4	-0.03049		
5	-0.02532		
6	-0.01463		
7	-0.01003		
8	-0.00332		
9	-0.00143		
10	0.001244		
11	0.000168		
12	0.004829		
13	0.005162		
14	0.004413		
15	0.006272		
16	0.000428	0.046042	

TABLES IX – XIII show GSA scales super linearly, meaning it gains more than 100% speedup for each additional thread. This means that less work was done with multiple threads, indicating an error with the parallelization. With the limited testing at 5000 particles, this was no longer true. This anomaly will be discussed more later. Looking at the percent differences we can see that the test with 5000 particles was the best, which lines up with the results of the other two algorithms. Interestingly, the 1000 particle 16 thread gave the worst fitness score by far, which could be due to the error in calculation. There were, however, scores within 1% of the best when using multiple threads.

In order to fully understand the results, we must look at the fitness scores and execution times of all 3 algorithms together. This will show which algorithm can find the best parameters,

and which can run the fastest. Below are the average times and fitness of each run for all 3 algorithms.

Random

TABLE XIV. AVERAGE TIME OF POPULATION BASED RANDOM SEARCH ALGORITHM

Number of Threads	Number of Particles		
	1000	*5000*	*10000*
1	75.77638	389.875	790.6932
2	41.69716	219.0774	456.8786
3	31.07456	168.0386	351.81
4	25.50846	134.846	287.975
5	21.6121	118.5288	257.5686
6	19.48238	107.53	236.9524
7	17.99092	100.8546	219.039
8	16.6301	94.79424	205.6684
9	15.89212	90.58342	192.5442
10	15.12068	84.95026	177.5592
11	14.59768	83.10272	181.4922
12	14.09022	79.18432	165.2616
13	13.58606	77.09566	161.089
14	13.16328	74.81852	156.5978
15	12.92662	72.81526	152.9366
16	12.45582	70.8828	150.4166

TABLE XV. AVERAGE FITNESS OF POPULATION BASED RANDOM SEARCH ALGORITHM

Number of Threads	**Number of Particles**		
	1000	***5000***	***10000***
1	0.089382	0.04225194	0.03950926
2	0.083722	0.042166	0.04539956
3	0.092786	0.03954948	0.03297046
4	0.091166	0.06706372	0.034567072
5	0.088973	0.04313786	0.02567896
6	0.096051	0.04110122	0.02914718
7	0.093243	0.04397034	0.03626448
8	0.07651	0.05849566	0.03287242
9	0.066573	0.05377834	0.04473312
10	0.080682	0.0392161	0.03447372
11	0.056789	0.05506464	0.03201626
12	0.072663	0.04317974	0.0417653
13	0.085513	0.0567356	0.03700902
14	0.077441	0.05112642	0.04964458
15	0.062604	0.05093828	0.03936926
16	0.074612	0.03574136	0.030435224

TABLE XVI. AVERAGE TIME OF PARTICLE SWARM OPTIMIZATION

Number of Threads	Number of Particles		
	1000	***5000***	***10000***
1	72.89526	364.0886	729.6172
2	36.7248	183.5062	369.6226
3	25.0591	125.1348	262.575
4	19.22382	96.00432	202.361
5	15.8865	78.60626	168.36
6	13.5948	66.54428	141.3698
7	12.0719	58.4916	125.848
8	10.56064	51.85788	110.5712
9	9.433894	47.34378	93.42814
10	8.802816	42.87324	85.1067
11	8.031126	39.52218	77.44792
12	7.521206	36.23274	71.21018
13	7.037664	33.5037	66.55906
14	6.602724	31.46284	63.2296
15	6.245446	29.57284	57.86
16	5.904416	27.90168	54.53744

TABLE XVII. AVERAGE FITNESS OF PARTICLE SWARM OPTIMIZATION

Number of Threads	**Number of Particles**		
	1000	***5000***	***10000***
1	0.230209	0.11448	0.105804
2	0.184192	0.10515	0.072448
3	0.162927	0.111364	0.083989
4	0.156726	0.134404	0.084532
5	0.238368	0.114352	0.096726
6	0.177807	0.137468	0.108425
7	0.205969	0.118863	0.067675
8	0.200063	0.106564	0.087651
9	0.176546	0.102954	0.092671
10	0.165113	0.111586	0.101075
11	0.18786	0.089294	0.093948
12	0.161048	0.086448	0.087167
13	0.249613	0.12671	0.090466
14	0.17598	0.097657	0.083373
15	0.193309	0.083376	0.106779
16	0.178007	0.09892	0.087226

TABLE XVIII. AVERAGE TIME OF GRAVITATIONAL SEARCH ALGORITHM

Number of Threads	Number of Particles		
	1000	*5000*	*10000*
1	190.1464	6542.81	
2	86.60482		
3	57.47764		
4	43.18804		
5	34.17818		
6	29.37212		
7	25.52834		
8	23.216		
9	20.88516		
10	19.2275		
11	17.31514		
12	16.68718		
13	15.5326		
14	14.3611		
15	13.78944		
16	11.96047	691.342	

TABLE XIX. AVERAGE FITNESS OF GRAVITATIONAL SEARCH ALGORITHM

Number of Threads	Number of Particles		
	1000	***5000***	***10000***
1	0.21351	0.257957	
2	0.24641		
3	0.269761		
4	0.22702		
5	0.230633		
6	0.22016		
7	0.225286		
8	0.22436		
9	0.205391		
10	0.236895		
11	0.240023		
12	0.220917		
13	0.22264		
14	0.229612		
15	0.252428		
16	0.382202	0.197773	

TABLES XIV – XIX show PSO runs in half the time or better at high thread counts compared to random. At low counts, the difference is much smaller, only a few percent. This shows the efficiency of PSO and the benefit of parallelization. GSA also runs substantially faster, but as stated previously, this is likely due to an error where calculations are skipped. At 5000 particles you can see how GSA becomes substantially less efficient, as the effect of n-body calculations is exponential with more particles. This is why the runs were shortened and not all tests completed, due to the exponentially longer time GSA takes vs both PSO and random.

Comparing the algorithms in terms of fitness reveals a different story. Random had substantially better fitness scores, with even the 1000 particle tests having better scores than some of the 10k PSO runs. GSA had the worst scores, with no score dropping below 0.19, more

than double the fitness of the worst random score and worst than all but a few of the worst PSO scores. This shows two things. Firstly, that my implementation of GSA was flawed. Given the impossible speedup, and the terrible fitness, the algorithm is not successfully converging on an optimal solution, but rather getting stuck somewhere. It is difficult to say why GSA is not working. No sources I was able to find for the literature review provide their source code. This meant that I had to create the algorithm going only off their diagrams, equations, and pseudocode. Even if the code has been posted in a language I was unfamiliar with, I would have been able to translate it or use it to see where my algorithm is failing.

The second and more major issue is that the random results were always better than either algorithm. This means that you are better off randomly guessing values vs using an algorithm to traverse possible solutions. The random produced results which were always better than either algorithm by a large margin, especially at lower particle counts. PSO with 10k particles and 16 threads gives a fitness score of 0.087226 found in 63.2296 seconds. Random can produce a fitness score of 0.089382 in 75.77638 seconds with only 1 thread. At 16 threads the fitness was 0.074612 and the time was reduced to just 12.45582 seconds. Given this information, there is no reason to develop and algorithm when randomly picking values produces a better result in less time. The best score made by the random selection was 0.02567896, done in 257.5686 seconds, as it was not a 16-thread run. This is better than the ~0.028 that I based a good run on from [2].

Random being better than GSA was to be expected, given the known limitations of my GSA implementation. However, the PSO algorithm had been tested and verified with rastrigin to find known good values. Even with an algorithm as good as PSO, picking random values still yielded much better results, albeit taking more time to do so.

Looking at other algorithms we can see that their results are substantially better than the ones shown here. Using several different algorithms, they are all able to achieve a fitness score more than 10x better. These fitness scores were ~ 9.84e-04 [8, pp. 6-7]. This shows that while the solutions presented in this paper are effective, there are both better and worse alternatives available.

Application of Results

The application of the results found in this research are interesting. Given the limited scope of the testing, this may not necessarily apply to all algorithms, but in our results using a random search proved more effective at finding a solution than an optimization algorithm. Even an algorithm as efficient and well known as PSO could not produce a result better than random with the same samples, and in fact was much worse.

Given the GSA algorithm has unknown issues, it is not suitable to be used in other problems. However, both the random and PSO could easily be modified to suit a different problem set. A few changes would need to be made to the algorithm, such as allowing to fitness function to be modified, and changing how the bounds are input to the program.

Conclusion

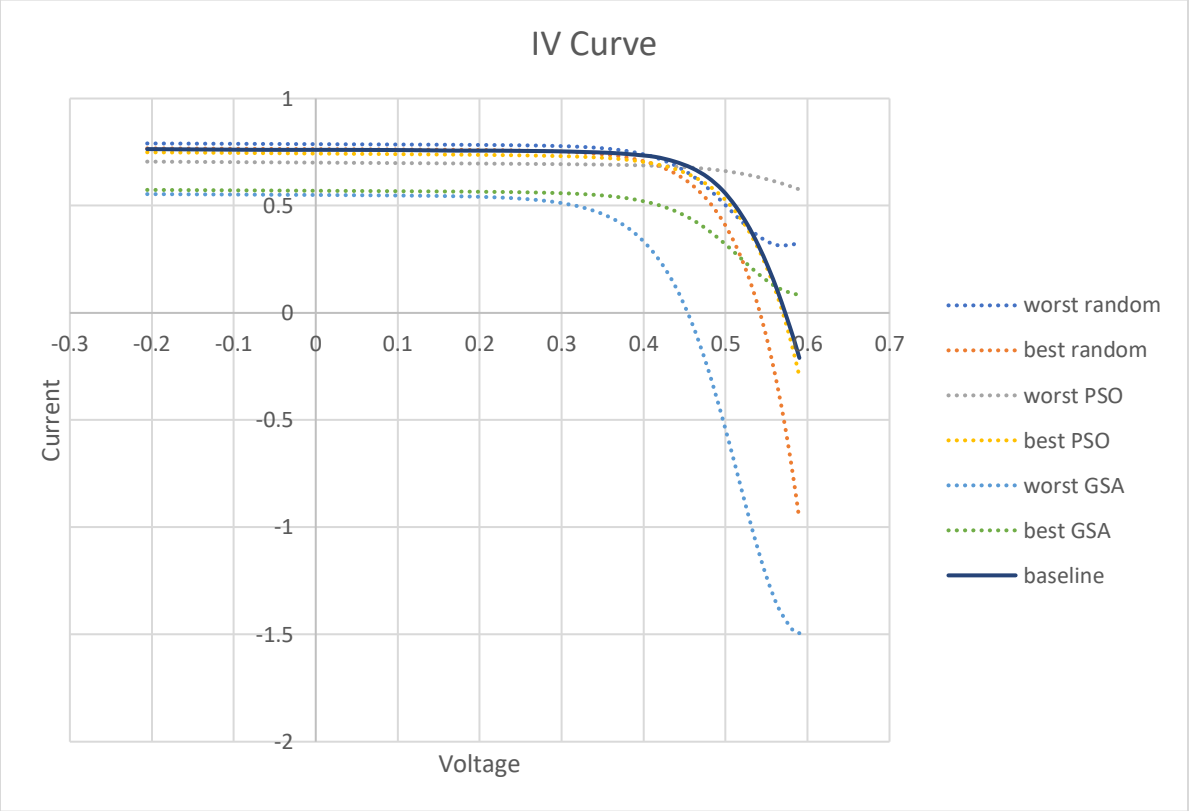

Fig. 2. Best and worst IV curves of various algorithms, along with baseline

TABLE XX. ESITMATED PARAMETERS FOR VARIOUS ALGORITHMS

Algorithm & Run	**Parameter**				
	I_{Ph}	I_O	R_S	R_{sh}	n
Worst Random	0.788184	9.85E-07	0.0900957	61.79	1.66269
Best Random	0.748417	6.02E-06	0.0104458	57.893	1.86124
Worst PSO	0.7015	7.84E-07	0.029655	44.4558	1.86433
Best PSO	0.743657	6.04E-06	0.0146951	34.6244	1.84881
Worst GSA	0.551209	3.73E-05	0.0883014	51.6216	1.99729
Best GSA	0.570781	3.84E-06	0.0742547	50.6512	1.85838

As shown in Fig. 2 and table XX, the results vary quite wildly from best to worst each algorithm. Interestingly, the best random seems to be further away from the baseline vs the best PSO yet has a better fitness score. These plots show not only the flaws with GSA, but also with estimating the parameters. Especially towards the end of the curve at high voltages the curves deviate wildly from the baseline. Until this point, no matter where they start, they remain more or less parallel

until ~0.3v, when they start to deviate. This is further proof that experiments need to be repeatable, and in order for this to be true, source code or other more detailed documentation that would allow others to repeat the experiment is needed.

Several things could be done in the future. One would be the addition of the ability to change the fitness function to make the algorithms more adaptable. There are also a few parameters that were not exposed in the command line for PSO and GSA that could be added to further increase the adaptability of the program. I also did not have time to run all of the GSA, so it is possible, although highly unlikely from previous testing, that the results would look different. Finally, outputting the best timestep would be useful to know if all 10,000 timesteps were really needed, as it is possible that the solution was found much sooner, but no exit condition for a "good enough" solution was created, or was the best timestep recorded in the files, only in the terminal output, which was not saved.

As for the parameter estimation, it is difficult to say why the results I got were much worse than the results of other experiments. I hope this shows the need to release the source code of future research projects. As with any field of science, an experiment should be repeatable. Steps in other fields are detailed enough that the experiment can be repeated. Yet with all of the sources used in the literature review, none of them provided their source code. This means that anyone wishing to recreate and validate the papers findings must recreate the code from scratch, likely running into the same errors that the original researchers did, as well as making assumptions to fill in the missing pieces.

References

[1] P. Hersch and K. Zweibel, Basic Photovoltaic Principles and Methods, Washington, DC: Technical information Office, 1982, p. 71.

[2] A. Valdivia-González, D. Zaldívar, E. Cuevas, M. Pérez-Cisneros, F. Fausto and A. González, "A Chaos-Embedded Gravitational Search Algorithm for the identification of Electrical Parameters of Photovoltaic Cells," *Energies,* 2017.

[3] K. G. Ing, H. Monkhlis, H. A. Illias, M. M. Aman and J. J. Jamian, "Gravitational Search Algorithm and Selection Approach for Optimal Distribution Network Configuration Based on Daily Photovoltaic and Loading Variation," *Nature-Inspired Algorithms for Real-World Optimization Problems,* vol. 2015, 2015.

[4] E. Rashedi, E. Rashedi and H. Nezamabadi-pour, "A comprehensive survey on gravitational search algorithm," *Swarm and Evolutionary Computation,* vol. 41, pp. 141-158, August 2018.

[5] D. Oliva, M. A. Elaziz, A. H. Elisheikh and A. A. Ewwes, "A review on meta-heuristics methods for estimating parameters of solar cells," *Journal of Power Sources,* vol. 435, September 2019.

[6] O. A. R. Board, "OpenMP Specifications," November 2018. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[7] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham and T. S. Woodall, "Open MPI: Goals, Concept, and Design of aNext Generation MPI Implementation," in *11th European PVM/MPI Users' Group Meeting*, 2004, 2004.

[8] D. Yousri, M. A. Elaziz, D. Oliva, L. Abualigah, M. A. Al-qaness and A. A. Ewees, "Reliable applied objective for identifying simple and detailed photovoltaic models using modern metaheuristics: Comparative study," *Energy Conversion and Management,* vol. 223, November 2020.

[9] A. Ramadan, S. Kamel, A. Korashy and J. Yu, "Photovoltaic Cells Parameter Estimation Using an Enhanced Teaching–Learning-Based Optimization Algorithm," *Iranian Journal of Science and Technology, Transactions of Electrical Engineering,* vol. 44, pp. 767-779, August 2019.

[10] A. Al-Shubhi, "Parameters estimation of photovoltaic cells using simple and efficient mathematical models," *Solar Energy,* vol. 209, pp. 245-257, 2020.

[11] K. R. Mahmoud and S. Hamad, "Parallel Implementation of Hybrid GSA-NM Algorithm for AdaptiveBeam-Forming Applications," *Progress In Electromagnetics Research B,* vol. 58, pp. 44-57, 2017.

[12] C. Saravanan and K. Srinivasan, "Optimal Extraction of Photovoltaic Model Parameters Using Gravitational Search Algorithm Approach," *Circuits and Systems,* vol. 7, p. 13, September 2016.

[13] A. Zarrabi, K. Samsudin and E. K. Karuppiah, "Gravitational search algorithm using CUDA: a case study in high-performance metaheuristics," *The Journal of Supercomputing,* vol. 71, pp. 1277-1296, December 2014.

[14] A. Zarrabi, E. . K. Karuppiah, Y. . K. Kok, N. C. Hai and S. See, "Gravitational Search Algorithm using CUDA," in *15th International Conference on Parallel and Distributed Computing, Applications and Technologies*, Hong Kong, 2014.

[15] S. Oliver, 2020. [Online]. Available: https://www.iwomp2020.org/wp-content/uploads/iwomp-2020-P7-task-unbalanced.pdf.

[16] B. Yang, J. Wang, X. Zhang, T. Yu, W. Yao, H. Shu, F. Zeng and L. Sun, "Comprehensive overview of meta-heuristic algorithm applications on PV cell parameter identification," *Energy Conservation and Management,* vol. 208, March 2020.

[17] A. Kumar and D. B. Das, "Comparative Analysis of Metaheuristic Algorithms for the Implementation of Photovoltaic Solar Panel Model in MATLAB/Simulink," *Advances in Systems Engineering,* pp. 369-378, January 2021.

[18] P. Tharawetcharak, T. Karot and C. Pornsing, "An Improved Gravitational Coefficient Functionfor Enhancing Gravitational Search Algorithm's Performance," *International Journal of Machine Learning and Computing,* vol. 9, no. 3, June 2019.