Practical Swarm Intelligence in Python

Using Swarm Intelligence and Evolutionary Algorithms
to Solve Problems in Engineering and Science

Image credit David Castor

Ronald T. Kneusel, Ph.D.

Practical Swarm Intelligence in Python

Using Swarm Intelligence and Evolutionary Algorithms
to Solve Problems in Engineering and Science

Ronald T. Kneusel, Ph.D.

Correspondence: swarmoptimizationbook@gmail.com

Contents

I Algorithms **1**

1 Swarm Algorithms **2**
1.1 What is Swarm Optimization? . . . 3
1.2 A High-Level Taxonomy . . . 7
1.3 A Brief Visit To The Zoo . . . 10

2 Setting The Stage **13**
2.1 Our General Approach . . . 13
2.2 Objectives . . . 16
2.3 Boundaries . . . 17
2.4 Initializers . . . 18
2.5 Are We Done Yet? . . . 22
2.6 Inertia . . . 23
2.7 Setting Up An Optimization . . . 24

3 Random Optimization **28**
3.1 Good Enough For Now . . . 28
3.2 The RO Class . . . 33
3.2.1 Optimize . . . 34
3.2.2 Initialize . . . 35
3.2.3 Step . . . 35
3.2.4 Done . . . 36
3.2.5 CandidatePositions . . . 37
3.2.6 Evaluate . . . 37
3.2.7 Results . . . 38
3.3 Testing the RO Class . . . 38

4 Particle Swarm Optimization **44**
4.1 Making Sense of the World . . . 44
4.1.1 Canonical PSO . . . 44
4.1.2 Bare Bones PSO . . . 48
4.1.3 Configuring a Particle Swarm . . . 48
4.2 The PSO Class . . . 49
4.2.1 Optimize . . . 51
4.2.2 Initialize . . . 51
4.2.3 Step . . . 52
4.2.4 Done . . . 53

4.2.5 NeighborhoodBest . . . 53
4.2.6 RingNeighborhood . . . 54
4.2.7 BareBonesUpdate . . . 55
4.2.8 Evaluate . . . 55
4.2.9 Results . . . 55
4.3 Testing the PSO Class . . . 56

5 New Kids On The Block 62
5.1 Jaya . . . 62
5.1.1 Description . . . 62
5.1.2 Implementation . . . 63
5.2 The Grey Wolf Optimizer . . . 64
5.2.1 Description . . . 65
5.2.2 Implementation . . . 66
5.3 Testing Jaya and GWO . . . 68
5.3.1 Success or Failure? . . . 69
5.3.2 Dispersion . . . 70
5.3.3 Convergence . . . 71
5.3.4 Precision . . . 71
5.3.5 Runtime . . . 74
5.3.6 Evaluation . . . 75

6 Genetic Algorithm 78
6.1 Making Darwin Proud . . . 78
6.2 The GA Class . . . 82
6.2.1 Step . . . 82
6.2.2 Evolve . . . 83
6.2.3 Mutate . . . 83
6.2.4 Crossover . . . 84
6.3 Testing the GA Class . . . 84
6.3.1 Modifying Population Size and Generations . . . 84
6.3.2 Modifying CR, F, and η . . . 86
6.3.3 Comparison with Other Algorithms . . . 90
6.3.4 Higher-Dimensional Searches . . . 90

7 Differential Evolution 95
7.1 Unnatural Mutation . . . 95
7.1.1 Configuring DE . . . 97
7.2 The DE Class . . . 97
7.2.1 Step . . . 98
7.2.2 CandidatePositions . . . 99
7.2.3 Candidate . . . 99
7.3 Testing the DE Class . . . 100
7.3.1 Experiments with a 2D Gaussian . . . 100
7.3.2 Modifying CR and F . . . 102
7.3.3 Comparing DE to Other Algorithms . . . 105

II Experiments **110**

8 Initial Experiments **111**
8.1 Standard Test Functions . . . 111
8.2 The 0-1 Knapsack . . . 119
8.2.1 The Problem . . . 120
8.2.2 The Setup . . . 120
8.2.3 The Results . . . 124
8.3 Curve Fitting . . . 127
8.3.1 The Problem . . . 128
8.3.2 The Setup . . . 129
8.3.3 The Results . . . 133

9 Training a Neural Network **139**
9.1 The Problem . . . 139
9.2 The Setup . . . 142
9.3 The Results . . . 148

10 Images **153**
10.1 Image Registration . . . 153
10.1.1 The Problem . . . 153
10.1.2 The Setup . . . 154
10.1.3 The Results . . . 158
10.2 Image Segmentation . . . 164
10.2.1 The Problem . . . 164
10.2.2 The Setup . . . 165
10.2.3 The Results . . . 168
10.3 Image Enhancement . . . 171
10.3.1 The Problem . . . 171
10.3.2 The Setup . . . 172
10.3.3 The Results . . . 174

11 Music **178**
11.1 Setting the Stage . . . 178
11.1.1 Tools . . . 179
11.1.2 Building Melodies . . . 179
11.2 Learning and Merging Melodies . . . 180
11.3 Learning Similar Melodies . . . 185
11.4 Learning Melodies from Scratch . . . 187
11.4.1 The Code . . . 188
11.4.2 The Experiments . . . 193

12 Cell Towers and Circles **197**
12.1 Cell Towers . . . 197
12.1.1 The Setup . . . 197
12.1.2 The Code . . . 198
12.1.3 The Experiments . . . 201
12.2 Packing Circles . . . 204

12.2.1 The Code 207
12.2.2 The Experiments 208
12.3 Summary 210

13 Grocery Store Simulation **213**
13.1 The Design 213
13.1.1 Inventory 213
13.1.2 Stores 215
13.1.3 Shoppers 216
13.1.4 Running the Simulation 216
13.2 The Shopper 217
13.3 The Store 219
13.4 The Simulation 221
13.4.1 Testing the Algorithms 222
13.4.2 Working with RO 224
13.4.3 Varying the Number of Shoppers 226

14 Discussion **228**

Introduction

I encountered my first swarm algorithm in 2007 when I stumbled across a description of particle swarm optimization (PSO). At the time, I had a particular optimization problem to solve, one that didn't lend itself to more traditional, gradient-based methods. Grid search was an option, but I was hoping some creative people had found a better way; it turns out they did. Particle swarm optimization solved my problem quickly. Along the way, I realized how powerful and widely applicable a tool it is.

After PSO, I encountered other swarm algorithms beginning with differential evolution (DE). The flood-gates opened when I read about the vast and ever-expanding array of nature-inspired optimization methods. I was intrigued and have consistently used swarm algorithms ever since for tasks ranging from curve fitting with nonlinear functions to generating architectures for convolutional neural networks. This isn't to say I don't have reservations about the utility of a continuously growing swarm of swarm algorithms, but such reservations need to wait until Chapter 1.

This book is the result of my experience with swarm algorithms and continues my fascination with the power of randomness and how it can generate order from disorder – witness biological evolution.

We'll define what we mean by *swarm algorithm* or *swarm optimization* more rigorously in Chapter **??**. But, for now, think of a swarm optimization as a way to find the best set of parameters to solve a problem. Finding the parameters is the optimization part. The swarm is how the optimization happens – think of a flock of birds wandering through a space where each point in the space represents a possible solution to the problem.

This book is meant as a *practical* introduction to using swarms to solve optimization problems. I added emphasis on the word practical because the goal is to introduce you to the thought process behind using swarms. At the same time, we'll present a representative set of swarm algorithms, explain how they work, and provide implementations in Python along with all the other parts necessary to quickly build solutions to problems.

This book is not, however, a scholarly treatment of the subject. We won't be proving theorems or diving into deep mathematical formulations. Instead, we'll learn by doing, by experimentation, and by empirical observation of how our algorithms perform.

It's my sincere hope that by the time you put this book down, you've learned how to apply swarm optimization algorithms to your own problems using the Python framework as a guide.

If you are already familiar with swarm techniques, you might have raised an eyebrow at the above mention of differential evolution as DE is an evolutionary algorithm, something distinct from the concept of a swarm algorithm.

We won't be pedantic or overly academic. If an algorithm fits our framework and involves multiple agents searching a space according to some method of updating that search, we'll consider it a "swarm algorithm" and fair game for inclusion and discussion. My defense

for this affront is the book's practical emphasis. I want you to come away from the book prepared to apply swarm algorithms to your problems, regardless of whether the method is appropriately labeled "swarm intelligence" or "evolutionary algorithm."

Who Is This Book For?

This book is for people who need to solve problems expressable as locating the best set of parameters. These are often problems in engineering, like where to place a set of cell towers with specific characteristics. Or, they are problems in commerce: how should I arrange the products in my grocery store to maximize my profit? As we'll see in Part II, we can use swarms for creative efforts, like generating melodies from scratch. I have even heard of brewers using optimization techniques to improve the quality of their beer; something of which I wholeheartedly approve.

Swarm approaches are that generic; seemingly, the only limiting factor is creativity in mapping the problem to searching a space of possible solutions. For example, a bit of imagination was necessary to map a multidimensional vector to a convolutional neural network architecture. However, once the mapping was defined, the swarms performed brilliantly.

Therefore, this book is for engineers and scientists. It's also for students of these disciplines. Additionally, it's for economists and mathematicians, as well as composers and brewers of fine beer. If you can cast the problem in the form of locating the best position in a multidimensional space, then it's likely swarm algorithms apply, even if your measure of the quality of a solution is complex and not a simple, closed-form mathematical function.

What Can You Expect To Learn?

The emphasis here is on building intuition through experimentation. This is not a theory book. We don't ask *why* any particular algorithm works, at least in detail, but instead, we use them and learn via experience how well they work and in what situations one might work better than another. A key concept we'll learn is there's no free lunch; there is no best swarm algorithm, you need experience with many to understand, even intuitively, when to try one or another.

To be explicit, you can expect to learn the following,

- What swarm algorithms entail and how to cast a problem in a form where it applies.
- The algorithm and implementation of several widely-used swarm optimizers, including classics like particle swarm optimization and differential evolution, and newer algorithms like the Grey Wolf Optimizer and Jaya.
- How to develop intuition about when to try which algorithm and what sort of performance to expect.
- How to construct a basic code framework in Python suitable for our experiments and useful as a model for more serious implementations tailored to your own problems.

What Do I Expect You To Know Already?

The requirements for this book are minimal. I expect you to know mathematics through high school algebra. Familiarity with the concept of a vector will be helpful, but there is

no need to run out and take a crash course in linear algebra. I try to keep math notation to a minimum. Should the notation be unfamiliar, a glance at the source code will clarify.

Throughout the book, we develop what I call a "framework" – a set of Python 3.X classes implementing the swarm algorithms and supporting utilities. Therefore, it will help matters if you are familiar with Python. If you are a programmer but not familiar with Python, I suspect you'll pick up on it rather quickly. The web abounds in quality Python tutorials.

The framework does make heavy use of Python's NumPy library. NumPy adds array processing abilities to Python. Frankly, without NumPy, Python would not be a good choice for most of the scientific applications to which it is currently applied. For example, Python is the primary language used by the deep learning community because of NumPy and the power and performance it provides an otherwise elegant but slow language. Therefore, familiarity with NumPy will help in understanding the code. Again, the web abounds in tutorials. If you are familiar with existing array-oriented scientific languages like Matlab or IDL, or their open-source versions, Octave and GDL, then you'll have little difficulty parsing NumPy code.

The Code

All code and data files referenced in the text are on the book's Github site:

`https://github.com/rkneusel9/SwarmOptimization`

The algorithm and framework code uses NumPy (`https://numpy.org/`). Code producing plots depends on Matplotlib (`https://matplotlib.org/`). These libraries are standard fare for Python, and both are easily installed on Linux, macOS, and Windows. The music experiments of Chapter 11 have additional requirements, which are detailed there. At all times, my assumed environment is Linux, specifically Ubuntu 18.04 or later. While it is quite possible to work in Windows, I think it will be easier to work instead in a virtual machine running Ubuntu, especially for the music experiments.

About This Book

The book is divided into two parts. The first part introduces swarm optimization, the framework, and the selected algorithms. The second part puts everything to work running a set of experiments intended to show swarm optimization's diverse applicability and give you enough background to apply swarm algorithms in your work.

Part I

1. **Swarm Algorithms** We begin with an introduction to swarm algorithms. This chapter presents a high-level description of what these terms mean, followed by a high-level taxonomy of optimization algorithms and some comments on the current state of affairs.

2. **Setting The Stage** In this chapter, we develop our Python framework. The framework is the environment in which we conduct our experiments. This chapter defines

concepts and classes for handling boundaries, initialization, and the all-important objective function, the thing we seek to minimize or maximize. The chapter concludes with a worked example to demonstrate how the components function together.

3. **Random Optimization (RO)** The simplest of all swarm algorithms is random optimization where a set of independent particles wanders through the search space, each particle looking to find someplace better than its current position. Random optimization is so simple that many don't consider it a swarm algorithm at all. RO is described and then implemented in code. We then introduce a simple example to test the implementation. The example carries forward through the remainder of Part I.

4. **Particle Swarm Optimization (PSO)** In a sense, PSO is the grandfather of swarm algorithms. It was developed in the mid-1990s and has strongly influenced many other algorithms. As with Random Optimization, PSO is described and then implemented and finally tested on our running example, including comparison with RO. There are many versions of PSO. Here we introduce only two: canonical and bare bones.

5. **New Kids On The Block** Two algorithms are presented in this chapter as representatives of the many new algorithms presently in the literature. Specifically, we introduce the Grey Wolf Optimizer (GWO) and Jaya. As with RO and PSO, we describe the algorithms, develop Python classes, and test them against each other and PSO and RO.

6. **Genetic Algorithm (GA)** Here, we introduce our version of the genetic algorithm. The implementation and approach are slightly non-standard but fit nicely with our framework. Unlike the swarm algorithms of previous chapters that use a particular algorithm to update the position of the swarm particles, the genetic algorithm seeks to evolve a solution from a population of individuals via breeding and random mutation. The correspondence between "population" and "swarm", along with a specific approach to breeding (crossover) and mutation, lets us cast GA in a form amenable to our framework. After implementing the algorithm, we test it against all the previous algorithms.

7. **Differential Evolution (DE)** We save DE for last because it is in the same class as the genetic algorithm – it is evolutionary instead of algorithmic. However, it could be argued to fall somewhere between the two classes of algorithms. As with GA, DE fits our framework and the concept of a swarm, so we include it. Additionally, DE is too powerful and useful to ignore. We implement DE in Python, including both the "rand" and "best" versions and a new hybrid, "toggle". We also tweak the crossover rule to allow traditional or GA-style. Don't be concerned if the terms are meaningless now; we'll describe them thoroughly when the time comes. As before, DE is compared to all the other algorithms on our running test problem.

Part II

8. **Initial Experiments** Part I of the book set up our working environment and introduced us to the swarm algorithms we want to explore. This chapter starts our explorations with some elementary experiments, including evaluating a series of standard test functions. Virtually all new swarm algorithms present themselves with

metrics based on how well they handle the test functions. Additionally, we try the algorithms on the 0-1 knapsack, a classic computer science problem, and nonlinear curve fitting.

9. **Training A Neural Network** Modern deep neural networks are generally not trained via swarm algorithms. The experiments in this chapter are not intended to bridge that gap. Rather, we'll instead use swarms to train modest, traditional, fully-connected neural networks. All experiments used search spaces of perhaps a couple dozen or fewer dimensions to this point in the book. In this chapter, we increase the dimensionality about two orders of magnitude using swarms to train neural networks with nearly 2000 learnable parameters.

10. **Images** This chapter conducts experiments in image registration, segmentation, and enhancement. We'll see clearly (pun intended) that swarms can help in this area. Registration means aligning images so they match as best as possible. Segmentation means changing the image into regions that have some reason for being grouped. For us, segmentation means best representing a grayscale image histogram by the sum of a specified number of Gaussian functions. Finally, image enhancement means making images look nicer. We'll experiment with a widely-used enhancement function and learn the best parameters to use for a set of standard test images.

11. **Music** Can swarms generate melodies? That's the question we answer in this chapter. Spoiler, the answer is "yes." First, we learn to copy a single melody, then to merge two melodies. Next, we ask if a swarm can learn a melody that sounds like a set of related melodies. Here the two sets are Irish skip jigs and Bach chorales. Finally, we test whether swarms can learn melodies in a given musical mode. The point of this chapter is to explore how a swarm might assist in what is, fundamentally, a subjective, human endeavor.

12. **Cell Towers and Circles** Many tasks involve allocation of resources according to constraints. This chapter addresses one such problem and then another related to it. The first problem is to learn where to place a set of cell towers to maximize coverage while simultaneously avoiding places where a tower cannot be put, like a building or a road. Yes, towers can be placed on buildings, but not in our world. The related problem is that of packing a square with a given number of identically sized circles.

13. **Grocery Store Simulation** Grocery stores tend to put frequently purchased items, like milk, towards the back of the store. In this chapter, we simulate a one-dimensional grocery store and learn that a swarm comes to the same conclusion to maximize daily revenue. This experiment demonstrates using a simulation as the objective function.

14. **Discussion** We've reached the end of the book. We spend a bit of time summarizing what we've learned about each algorithm and its suitability for different tasks based on the results of our experiments.

Contact

This is an active book, meaning you try things, get results, and increase your understanding. Included in that is the ability to communicate with me should you desire to, especially if you – gasp! – find a bug in the code. Reach me here:

swarmoptimizationbook@gmail.com

with comments and questions, etc.

Part I

Algorithms

Chapter 1

Swarm Algorithms

Why?

Most books do not start with a question, but I think one should. The purpose of this book is to introduce you to practical swarm algorithms so you can add them to your algorithmic toolbox. You already have many other tools in that box, and, presumably, when you pull one out, you do so because you know it to be suitable for the task. The same should be true for swarm algorithms.

So, again, why? Why learn about and use these techniques?

We know there is no such thing as "one algorithm to rule them all," so any claim swarm algorithms are always the best option is misinformed. However, swarm algorithms are several other things that might entice you to learn about them and apply them to your projects.

Firstly, and to me, most importantly, swarm algorithms are fascinating and fun. I've always been interested in the intersection between randomness and utility.[1] Swarm algorithms use randomness to arrive at something useful, a solution to a problem.

Secondly, swarm algorithms apply to hard problems, those that are not easily solved by other techniques. The approach is so general that almost any creative mapping between what a particle in the solution space represents and the actual solution you want stands a chance of succeeding (caveat emptor!)

Thirdly, as we'll do in this book, it is possible to set up a code framework for many swarm algorithms and options, a framework you can quickly use to find a solution to a problem as it arises. Here I'm thinking of the day-to-day work of an engineer who suddenly needs to find the proper shape for an antenna, and she doesn't have time to go and learn about the latest and greatest approaches; she needs a workable solution *now*.

Fourth, you might use swarm algorithms because you don't have access to expert mathematicians or operations research people who can help you implement a state-of-the-art solution. Of course, when your swarm solutions fail or are not precise enough, then you should seek out such people. If expert approaches are like a Corvette, then swarm algorithms are like an old Ford pickup that just keeps running and eventually gets you where you want to go.

Hopefully, these are sufficient reasons for you to keep reading. As I said, the first is adequate for me; the others are merely icing on the cake.

I once read that engineering is the art of making what you want from what you can get. Swarm algorithms are well-suited to this mode of thinking – they are easy to try, and even

[1] For example, see my book "Random Numbers and Computers" (Springer 2018).

though they are stochastic and cannot give guarantees; they often get close enough; they are often the "what you can get" part.

In many cases, this is all you need. CPU cycles are cheap and plentiful. It might seem like overkill to run a million iterations of a swarm of particles to find the parameters to fit your one dataset best, but if in clock time it takes ten seconds per run, that's just fine – run a bunch of searches to see what consensus you get, then use the one-off answer and get on with your project.

This chapter introduces us to swarm algorithms (Section 1.1). Unlike the hopes of many physicists in the 1980s searching for a grand unified theory of nature, the swarm algorithm does fit nicely on a t-shirt.

We follow in Section 1.2 with a brief taxonomy of swarm techniques. The goal is to give you some insight into the characteristics of different algorithms.

Recent decades have witnessed an explosion of new swarm algorithms, the majority of which claim to be "nature-inspired." I'm tempted to make a comparison with the Cambrian explosion, the sudden diversification of life 540 million years ago. At that time, most of the major groups of animals appear for the first time in the fossil record. Of course, while descendants of these animals still exist, many have gone extinct, i.e., trilobites (sadly). I fully expect most nature-inspired algorithms to follow the trilobites and relatively quickly. In Section 1.3, we visit the swarm algorithm zoo with the sort of critical eye we should regularly employ in science.

A note on terminology before we proceed. I'm using the phrase "swarm algorithm" as a catch-all for algorithms that manipulate a swarm or population of agents in some way to search space. This includes both swarm intelligence and evolutionary algorithms. At times, I'll refer to these algorithms as "swarm optimization." I do this to emphasize the goal of the algorithm. I'm still using the word "swarm" in the expanded sense. In practice, I see little utility in distinguishing between swarm intelligence and evolutionary algorithms, while, as part of a research program, the distinction is crucial and necessary.

1.1 What is Swarm Optimization?

We have a problem. To solve the problem, we need to find the best set of something. The something might be the parameters of a function best fitting a set of data, an arrangement of cell towers, products on store shelves, circles in a square, the weights and biases of a small neural network, or even the notes and durations of a melody. All of these are examples found in this book. In general, we have a problem where we need to find the "best" set of something from a broader set of possible somethings. How should we proceed?

If we can make the following two statements true:

- We have a problem where the set of possible solutions can be mapped in some way to a position in a multidimensional space.
- We have a function defined everywhere in that multidimensional space such that the value of the function is a proxy for the quality of the solution represented by that position.

then we might hope to solve our problem by somehow searching through this space of solutions seeking the best position as that will, we hope, map to the best solution to our actual problem. The experiments in this book demonstrate methods for developing this mapping. For now, let's assume we have it.

This is good; we have now reduced our problem to moving through a multidimensional space looking for a particular position, the position with the best, usually minimum, function value. How should we search this space?

One fairly obvious way to search the space is to pick points at equal intervals along each axis and evaluate our function at those positions. The position leading to the best function value will be our solution. We'll note here that we can always define our function so smaller is better; we'll seek the global minimum of the function. We lose nothing by insisting on this, so, henceforth, please assume all functions are to be minimized.

Searching by picking points at equal intervals is known as a *grid search*. It's quite systematic, and if the space to search is small in terms of the number of dimensions and size in each dimension, then grid search might be a sensible thing to do. For example, grid search is often used to optimize the C and γ parameters of a support vector machine classifier.

However, with a bit of thought, we see a problem with this approach. As the dimensionality of the search space increases, the number of points to consider to claim a reasonable search was performed increases far faster. This effect is quite similar to the curse of dimensionality that so plagued early machine learning models.[2]

Can we do better? Here's a thought: let's pick a point in the space at random and evaluate the function there. Then, using some rule, choose a nearby point, and evaluate the function at that point. If the nearby point has a smaller function value, we move to that point. If not, pick another nearby point and try again.

If we do this repeatedly, we might hope to move, eventually, to a good position in the space and thereby find a suitable solution to our problem. Indeed, if the space is convex with only one minimum position, think of a bowl, then this process will succeed as we only ever move closer and closer to the global minimum. Of course, it might be highly inefficient compared to more sensible approaches, but it will work.

Excellent! We have an approach to solving our problem. Unfortunately, we don't. Our approach could fail because our space might have multiple minima. If it does, and we fall into one, we'll never move back up to get out and find a still better minimum.

Therefore, instead of picking just one point at a time, let's scatter many points throughout the space and apply the same rule to each of them to search for the best position. Scattering points throughout the search space and moving them according to some rule to find the best position in the space is the essence of *swarm optimization*.

Swarm optimization falls under the larger heading of *metaheuristics*:

> **A metaheuristic is a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms. The term is also used to refer to a problem-specific implementation of a heuristic optimization algorithm according to the guidelines expressed in such a framework. ([1])**

From this definition, it is clear we are using metaheuristics in the second sense, as a particular implementation of an algorithm to solve a specific problem. To be even more precise, our goal is *global optimization*, we seek a single, best solution for a single function.

Let's be still more precise about our terminology. When we use the word "swarm", we mean the following:

[2]Modern deep learning has mostly overcome this curse, though mainly by accident and not by design.

> **A *swarm* is a collection of *agents*, usually identical, moving through a *space* guided by a *set of rules* governing their overall motion.**

Here the set of rules means any set, from the empty set (no rules, pure randomness) to highly ordered motion (like a grid search, no randomness).

This definition tells us what a swarm is, but not precisely what we mean by agents, space, and rules. Nor does it tell us anything more about the mysterious function referred to above. We need additional definitions, so let's start with *agent*:

> **An *agent* is a conceptual entity representing a position in a multi-dimensional space. The agent may also possess other characteristics and information, such as knowledge of its previous motion through the space.**

In practice, our agents are vectors of continuous numbers representing a position in the space.

Let's continue defining terms:

> ***Space* is multidimensional and (frequently) continuous. It is also often bounded. The positions in space are representations of solutions to a problem.**

This definition assumes the mapping from a position in space to a problem solution exists. Sometimes this mapping is obvious. If we want to find the minimum of a function over some range, the space is the argument to the function itself. For example, if we have $z = f(x, y)$ and we want the minimum value, $z_{\min}$, our space has two dimensions, x and y. However, in most interesting cases, the number of dimensions is higher and the mapping less obvious. Chapter 9 has us using a swarm to train a traditional neural network. In that case, the space encompasses all the weights and biases of the network, so each position in the space represents, quite literally, the actual weight and bias values of the network. The space here is large, with nearly 2000 dimensions. Most of our experiments are less ambitious. For example, our melody experiments use a space of 40 dimensions to represent a melody of 20 notes where dimensions are note and duration pairs.

Let's recap: we've taken our original problem, cast it as a space in which agents can move, knowing that each position represents a potential solution to our problem. We know now what we mean by a swarm, an agent, and a space. We've also hinted that we'll use a swarm of agents to search this space for an acceptable solution. This leaves two terms to define: the set of rules governing the motion of the swarm of agents and the function defined at every point in the space telling us how good a solution that point represents.

First, the set of rules:

> **A *set of rules* governing the motion of a swarm of agents *is* a swarm optimization algorithm. The algorithm uses the positions of the swarm agents and other related information to decide how to move each agent to a new position.**

We developed a simple swarm optimization algorithm above: scatter a set of agents throughout the space and move them to new positions whenever an agent finds someplace better than the place it currently is. This simple approach has a name, random optimization,

Algorithm 1 A swarm optimization search.

Input: An objective function, bounds, and initialization type
Output: The best position found by the swarm
  Initialize the swarm
  while not done **do**
    Update the particle positions
    Evaluate the new positions
    if new global best position found **then**
      Store the new global best
    end if
    Increment the iteration counter
  end while

and we'll develop it in detail in Chapter 3. Naturally, there are many other ways to move the swarm through the search space; otherwise, there would be no point in this book. We'll get to these other ways for a select set of algorithms throughout all of Part I.

We have one definition remaining, the mysterious function defined everywhere in space that tells us how good a solution a position represents:

> **The *objective function* is defined everywhere in space. The value returned by the objective function, almost always a scalar, represents the quality of the solution that position represents.**

The word "optimization" implies something is being optimized, being refined, or made better. The objective function, the formal name for the function we've been referring to so far, is the thing we intend to optimize by finding its minimum value. Since we define the mapping between the points of the multidimensional space and possible solutions to the problem, we likewise need to define a suitable objective function so that the minimum of the objective function truly represents the best solution to the problem. The objective function needs to capture the problem's essence, so the values returned correspond to better or worse solutions.

If we seek the minimum of a mathematical function over some range, then the function's value is the objective function. However, objective functions in swarm optimization can be more complicated. The objective function for the experiment of Chapter 9 where we are training a neural network is the network's performance on the held-out test dataset. In Chapter 11, our objective function is a set of measurements we intend to represent how "nice" a melody sounds in terms of fidelity to the desired musical mode and the types of intervals and note durations it contains. The objective function is the key to success in a swarm search, so we need to develop or chose it carefully. There are many cases, however, where the choice of the objective function is rather apparent. For example, we'll use the mean squared error between two values or sets of values more than once.

Algorithm 1 presents, in generic terms, the method employed by virtually all swarm optimization algorithms. It will fit on a t-shirt. What distinguishes one algorithm from another is implementing the phrase *Update the particle positions*. Note, Algorithm 1 refers to the agents as "particles." This is common, and we'll continue to use the word particle throughout the book, even for algorithms more traditionally associated with a genetic metaphor, i.e., genetic algorithms. This handy abuse of terminology simplifies the presentation.

If you peruse the literature on swarm optimization, you'll run into two terms repeatedly: exploration and exploitation. The former refers to the swarm's wandering through the space of possible solutions to the problem. The latter refers to the swarm's deciding a particular location in the solution space is worth a closer look. Different swarm algorithms seem to favor one over the other or switch during the search from one mode to another, usually from exploration to exploitation.

When the search starts, the swarm as a whole knows nothing about the search space. As the swarm evolves from iteration to iteration, one pass through the `while` loop of Algorithm 1, it discovers more about the search space by its explorations. When a particularly promising location is found, many algorithms switch, usually implicitly, to a mode of exploitation – the swarm focuses on that region of the search space to narrow down the search and locate the best solution. For many algorithms, it is not unusual for the swarm to collapse on itself in the vicinity of the best solution found. At that point, barring some additional feature of the algorithm allowing for renewed exploration, the swarm is done; it won't find any place in the search space that might be better. We'll see examples of this collapse several times during our experiments.

As with most things in life, balance matters. One, often serious, issue with swarm optimization is switching to exploitation mode too quickly – the swarm's collapse. If not identical, this is akin to becoming trapped in a local minimum of the objective function. Many algorithms have some ability to avoid this effect. Perhaps part of the swarm focuses on the promising region, but other parts of the swarm continue to explore if the grass is greener somewhere else.

For example, we'll work extensively with particle swarm optimization (PSO) and differential evolution (DE). PSO addresses the exploration versus exploitation balance by adjusting parameters: c_1, c_2, and ω. Typically, c_1 and c_2 are fixed for the problem, and ω is decreased as the search proceeds under the (hopeful) assumption that later iterations of the swarm will have found a good place in the search space, and it makes sense to begin exploitation. Still, parts of the swarm will continue to explore, and there are variations of PSO that explicitly add a "repulsive" term to prevent premature exploitation.

By contrast, though useful and powerful, differential evolution is known for giving up on exploration too quickly. If the objective function has a strong minimum that is easily found or lacks many local minima, this tendency to converge quickly is a benefit – fewer iterations are needed to find a good solution. However, balance again shows itself the better approach, so there are variants of DE that balance exploration and exploitation by using random components of the swarm regardless of the quality of their current location in the search space.

The previous two paragraphs are for background, do not be concerned if they don't make much sense now. Chapter 4 and Chapter 7 present PSO and DE respectively and in-depth.

1.2 A High-Level Taxonomy

A taxonomy is an organization into meaningful groups, a classification system. The best-known taxonomy is from biology and groups all living things into a hierarchy: kingdom, phylum, class, order, family, genus, and species. Thankfully, swarm techniques are not so diverse that a detailed and hierarchical taxonomy is needed.

Taxonomies are imposed from without based on characteristics; they are not unique. The

high-level taxonomy we'll present here uses terms from [2] along with a simpler grouping between nature-inspired swarm algorithms and those that are not.

In [2], global optimization algorithms are grouped into five categories where the category description is my summary:

- **"Mountaineer"** The mountaineer's goal is to climb a mountain, to reach the highest peak. Algorithms in this class only move towards the goal; they only move up, never down – exploitation over exploration.
- **"Sightseer"** The sightseers traverse the search space looking for interesting places to exploit by maintaining knowledge gained by the history of objective function evaluations.
- **"Team"** The team is a population algorithm, a group of individuals (agents) work together to explore and exploit the search space.
- **"Surveyor"** The surveyor builds an approximate map of the search space, then uses the map to select new regions to explore and map in finer detail.
- **"Chimera"** The chimera is a hybrid algorithm built from components of existing algorithms.

The six algorithms we'll work with in this book are random optimization (RO), particle swarm optimization (PSO), Jaya, Grey Wolf Optimizer (GWO), genetic algorithm (GA), and differential evolution (DE). The details of each algorithm are given in their respective chapters, but we can group them according to the taxonomy above as follows:

Algorithm	*Category*
Random Optimization	"Mountaineer"
Particle Swarm Optimization	"Team"
Jaya	"Team"
Grey Wolf Optimizer	"Team"
Genetic Algorithm	"Team"
Differential Evolution	"Team"

I suspect you've noticed a theme in the assignments. Except for random optimization, all of our algorithms fall into the "Team" category – a population of agents working together in some manner to locate the best position in the search space.

Our implementation of random optimization uses a population of agents as well, but, unlike all the other algorithms, the agents are blissfully unaware of each other. Individually, they are all mountaineers, each moving only when a better position is located relative to their current position.[3] A supreme overlord watches the population to pick who is doing the best. Still, that knowledge is cached; it is not used to influence the motion of the individuals.

The taxonomy of [2] is useful, but less so to us, because it's likely almost all swarm-based algorithms fall into the "Team" category – that's what makes them swarm-based.

Within swarm optimization, we can create a trivial categorization, not worthy of the name "taxonomy", by separating algorithms into two groups: those inspired by something in nature and those that aren't. Many such partitions use an animal or plant-based criterion, as

Artificial physics algorithm	Particle collision algorithm
Big bang-big crunch	Rain water algorithm
Black hole	Rain-fall optimization algorithm
Central force optimization	Ray optimization
Charged system search	River formation dynamics
Chemotherapy Science algorithm	Self-driven particles
Colliding bodies optimization	Simulated annealing
Electro-magnetism optimization	Simulated raindrop algorithm
Fractal-based algorithm	Sine cosine algorithm
Galaxy-based search algorithm	Sonar inspired optimization
Gravitational search	Space gravitational algorithm
Harmony search	Spiral optimization
Hydrological cycle algorithm	Stochastic difusion search
Intelligent water drop	Thermal exchange optimization
Ions motion algorithm	Vision correction algorithm
Integrated radiation algorithm	Vortex search algorithm
Light ray optimization	Water cycle algorithm
Mass and energy balances algorithm	Water wave optimization
Optics inspired optimization	Weighted attraction method

Table 1.1: Selected physics-inspired optimization algorithms.

opposed to something based on physics. We'll save the animal and plant-inspired algorithms for Section 1.3.

Table 1.1, based on [3] with additions from [4], lists optimization algorithms involving some process in physics. Building an optimization algorithm by simulating some aspect of the physical world is a reasonable thing to do. After all, physics *is* the foundational science, and physicists have spent centuries working to understand the basic principles involved. Table 1.1 is for your reference, should you care to explore these algorithms. For this book, we'll declare these algorithms *not* to be nature-inspired.

Since we are restricting the term "nature-inspired" to algorithms based on the behavior of animals and plants, let's see what label to use for our algorithms:

Algorithm	*Nature-inspired?*
Random Optimization	No
Particle Swarm Optimization	Yes
Jaya	No
Grey Wolf Optimizer	Yes
Genetic Algorithm	Yes
Differential Evolution	Yes

Something called Grey Wolf Optimizer is likely to be nature-inspired, and it is. Particle swarm optimization is often said to be inspired by the flocking of birds. Both the genetic algorithm and differential evolution are, not surprisingly, based on evolution. As stated above, random optimization is a hill-climbing algorithm, not at all nature-inspired. Jaya,

[3] As we've declared we will always minimize our objective functions, perhaps we should use a term like "submariner" in place of "mountaineer?"

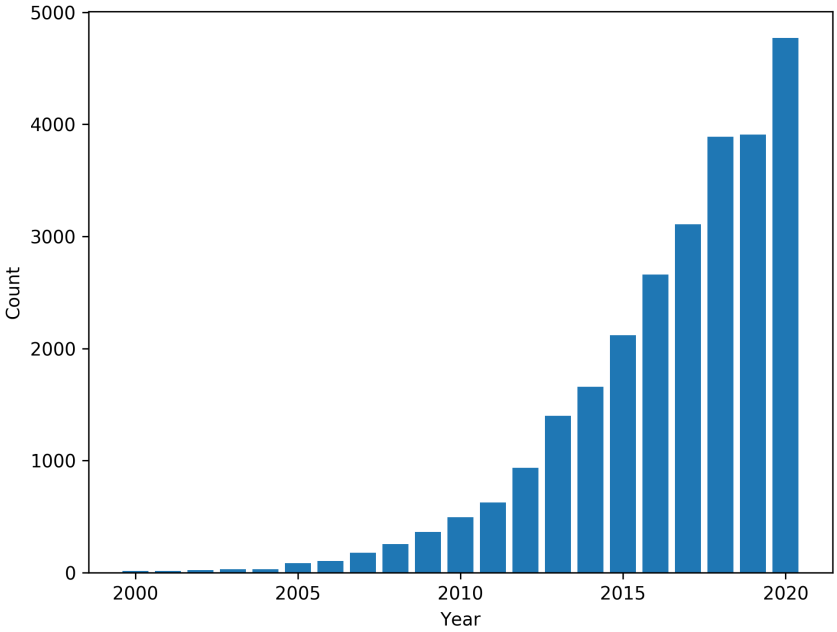


Figure 1.1: Nature-inspired results in Google Scholar by year.

likewise, does not claim inspiration from nature. Therefore, we have a healthy mix of algorithms for our experiments.

1.3 A Brief Visit To The Zoo

Before we wander too far along the path, I feel a need to pause and make a few observations on the plethora of new "nature-inspired" algorithms now on the market. To be clear, if an algorithm offers something substantive, something that previous algorithms lack in some way, then I'm all for it. However, that does not seem to be the case with what we've decided to label as "nature-inspired" algorithms, meaning those distinct from algorithms seeking to emulate some physical process (e.g., simulated annealing).

Wait, you say, Section 1.2 makes it clear that you, dear author, are willing to use nature-inspired algorithms. True, this book discusses several, starting with one of the first, particle swarm optimization. However, except for the Grey Wolf Optimizer and Jaya, the algorithms selected for inclusion are well-proven and battle-hardened. Also, Jaya makes no claims to be inspired by nature. In that sense, the only new, nature-inspired algorithm we'll explore is the Grey Wolf Optimizer.

The following is my take on the current state of affairs; your mileage will, of course, vary. However, I'm not alone. I point you to [5]. The introduction frames the situation precisely and plainly. I recommend you spend a little time with that paper.

Figure 1.1 shows the number of Google Scholar results by year to the search: nature-inspired optimization metaheuristic. These results include five books published since 2016: [6], [7], [8], [9], and [10]. Of these five books, three were published as recently as 2020. Interest in nature-inspired algorithms is indeed high and growing. This interest is a good thing, but that doesn't mean the ever-growing list of nature-inspired algorithms necessarily is.

Table 1.2 lists many nature-inspired algorithms, and the list is by no means exhaustive

– see [3]. It is difficult to believe all of these algorithms are unique or sufficiently distinct from existing algorithms that their publication is warranted.

That the algorithms of Table 1.2 "work" I have no doubt. The usual formula for a new swarm algorithm paper includes testing the algorithm against standard test functions (Section 8.1) and a select handful of existing algorithms, usually including at least canonical PSO. The typical result is something along the lines of "our new <insert-nature-inspired-name-here> algorithm is <better|competitive|comprable> in these limited test cases to existing algorithms." While no doubt true, I find such a conclusion unsatisfying and, to be frank, unhelpful in the long run. Are we really to accept that there is something about the Salp Swarm algorithm that is fundamentally better than PSO or one of its variants? Given that no algorithm will always be best at highly diverse tasks, why even report a competitive result? I have nothing against the salp swarm algorithm; it was selected randomly as the first name I noticed when I glanced at the list in Table 1.2. But, even if the nature-inspiration is valid, why would we expect salp motion to be better at searching a space than the flocking of birds when the same rules of evolution produced both?

Thoughtful comments from [11] are in order here:

> Are recent nature-inspired algorithms novel? Yes and no. On the one hand, most (but certainly not all) of the algorithms reviewed in this paper ([11]) are distinct from existing optimisation algorithms, and given a particular search space, they would likely follow different trajectories to existing algorithms. On the other hand, many of these algorithms use variants of well-established metaheuristic concepts that are also found in existing metaheuristic frameworks such as PSO, EAs and local search. Furthermore, the analysis of PSO-style algorithms shows that many of their underlying ideas have also been explored by the more mainstream PSO community.

Should this trend of inventing and publishing nature-inspired algorithm after nature-inspired algorithm continue, we might run out of obvious names. I recommend researchers wishing to continue the trend go prehistoric. Indeed, there's a certain pleasant ring to "Trilobite Swarm Optimization," a name I offer to the community as a candidate. Indeed, given their long successful run and vast numbers of fossils, swarms of trilobites effectively plied the Cambrian and Ordovician seas, locating food sources while avoiding hostile predators like Anomalocaris. Might an algorithm combining the near-mindless wanderings of a trilobite with primitive evasion strategies to avoid fast swimming Anomalocaris' overhead be a useful analogy for searching a space of solutions? I leave the implementation to interested readers, but please, don't publish your results.

Tongue-in-cheek names and mild ranting aside, I believe there is yet much to discover in this field, perhaps less in seeking inspiration from nature, at least at the level of animal behavior, and perhaps more from enhancing the intelligence of the agents combined with existing search strategies. Hybrid algorithms, which we do not discuss in this book because of space constraints, also seem promising, at least from my testing, to say nothing of the mountain of literature references. A hybrid algorithm combines aspects of one or more swarm algorithms and is categorized as a chimera according to [2].

Let's get started with the rest of the book. You'll learn key swarm optimization algorithms, how they work, and how to create reference implementations you can use or modify for your own tasks. With the intuition gained through the experiments of Part II, I believe you'll be in a good position to effectively evaluate and assess the utility of existing algorithms and the many new ones that will inevitably appear in the years to come.

African buffalo optimization	Grasshopper optmisation algorithm
Alienated ant algorithm	Great salmon run
Ant colony optimizer	Grey wolf optimizer
Ant lion optimizer	Harris hawks optimization
Artificial bee colony	Invasive weed optimization
Artificial root foraging algorithm	Jaguar algorithm
Bacterial foraging	Japanese tree frogs calling
Bacterial-GA foraging	Keshtel algorithm
Bat algorithm	Killer whale optimization
Bee colony optimization	Krill herd
Bee hive	Lion optimization algorithm
Bees algorithm	Locust swarm algorithm
Bees swarm optimization	Marriage in honey bees
Bee system	Monkey search
Bottlenose dolphin optimization	Moth-Flame optimization algorithm
Bumblebees	Opt bees
Cat swarm	Owl search algorithm
Chicken swarm optimization	Paddy field algorithm
Coral reefs optimization algorithm	Queen-bee evolution
Coyote optimization algorithm	Red deer algorithm
Cricket algorithm	Roach infestation algorithm
Crow search algorithm	Salp swarm algorithm
Cuckoo search	Shark smell optimization
Cuttlefish algorithm	Sheep shepherding algorithm
Dolphin echolocation	Shuffled frog leaping algorithm
Dragonfly algorithm	Sperm whale algorithm
Dynamic virtual bats algorithm	Spotted hyena optimizer
Eagle strategy	Squirrel search algorithm
Egyptian vulture algorithm	Swine flow optimization algorithm
Elephant search algorithm	Termite colony optimization
Emperor penguins colony	Tree growth algorithm
Fast bacterial swarming algorithm	Virtual ant algorithm
Firefly algorithm	Virtual bees
Fish swarm school	Virus colony search
Flower pollination algorithm	Weightless swarm algorithm
Fruit fly optimization	Whale optimization algorithm
Glowworm swarm optimization	Wolf search

Table 1.2: Selected nature-inspired optimization algorithms based on animals or plants.

Chapter 2

Setting The Stage

Now that we have a general idea of what our topic is, let's set the stage for the algorithms and experiments that follow. In this chapter, we'll build the Python framework used throughout the remainder of the book. In doing so, we'll see how to structure swarm optimization problems and understand the parts the algorithms have in common. Our framework expresses these parts as a set of classes.

In Section 2.1, we lay out our general approach, the structure of our framework. Section 2.2 presents an `Objective` class to wrap the objective function. We know the objective function is the most task-specific portion of the framework and that it will vary during our experiments. Next, in Section 2.3, we discuss boundaries, limitations on the allowed values used during the search. These fit nicely in a `Bounds` class that we'll subclass as needed.

Swarm algorithms need to be initialized. This initialization process is essential and often crucial to the success of the search. We'll define a set of initialization classes covering common cases in Section 2.4. From this set, you'll see how to define your initializations.

Knowing when to stop is a good thing. We'll run the algorithm until we've exhausted the number of iterations we set at the beginning, or until our objective function has reached a set threshold. However, we might, at times, wish to use more custom stopping criteria. Therefore, in Section 2.5, we discuss what it means to be done with a search.

One of our algorithms, particle swarm optimization (PSO), uses a concept referred to as inertia to control the evolution of the swarm. In Section 2.6, we define basic inertia classes to schedule how PSO uses inertia during a search. Again, these will guide the way should you wish to experiment with different approaches.

Finally, in Section 2.7, we put all the pieces together in code to illustrate how to set up a swarm optimization run and interpret its results.

2.1 Our General Approach

A swarm optimization search follows a set algorithm as we saw in Chapter 1. In that chapter, we introduced our general algorithm, here reproduced as Algorithm 2 for easy reference.

The inputs to the search are,

- The objective function, the thing that lets us know how well we are doing.
- The bounds for the search, including the number of dimensions our objective function is expecting along with the ranges allowed for each of those dimensions.

Algorithm 2 A swarm optimization search.

Input: An objective function, bounds, and initialization type
Output: The best position found by the swarm

```
Initialize the swarm
while not done do
    Update the particle positions
    Evaluate the new positions
    if new global best position found then
        Store the new global best
    end if
    Increment the iteration counter
end while
```

- How we'll initialize our swarm, which includes the type of initialization and the number of particles,
- And how we'll know when we are done searching.

Let's pick an example to make things less abstract. We have a function of two variables, $z = f(x, y)$, and we're looking for the minimum value of this function in the range $[0.01, 1]$ for both x and y. In other words, we seek two numbers, x_m and y_m, such that $z_m = f(x_m, y_m)$ is as small as possible for $0.01 \leq x_m, y_m \leq 1$.

We already know two things: we have a two-dimensional search space, and we know the bounds on the dimensions, $[0.01, 1]$. What we have left to select, aside from the type of search algorithm – the *Update the particle positions* part of Algorithm 2 – is the objective function, the number of particles in the swarm, the type of swarm initialization, and what it means for the search to be done.

The objective function measures the quality of each particle, i.e., how good of a solution it represents. For our example, the value of $f(x, y)$ for a given x and y *is* the objective function since we seek the minimum value of this function. Notice that we are making no assumptions as to what $f(x, y)$ is. It could be a simple algebraic function, or it could be a complex multistep algorithm in its own right. All we know or care about at this point is that $f(x, y)$ accepts two numeric inputs and returns a numeric output. This is already a step better than traditional optimization as we don't need to know of nor require the existence of any derivatives of $f(x, y)$.

The larger the swarm, the more computation needs to be done, but we might also expect to find the solution more quickly. However, this is not always the case; sometimes, it is better to use a smaller swarm and search more by using more iterations. In general, a modest-sized swarm is a good place to start, say in the range of 10 to 100 particles. The computational overhead of the objective function plays into selecting the swarm size. If the objective function can be evaluated quickly, we might choose a slightly larger swarm or more iterations of the swarm. However, if the objective function is computationally expensive, we might go with a smaller swarm and hope for quicker convergence to the solution or use tighter bounds to reduce the size of the search space.

We'll use N particles for the swarm. This means that our swarm consists of N two-dimensional vectors represented in Python as a Nx2 NumPy matrix of positions where each row of the matrix is a particle, and each column of the matrix is the position of the particle along that dimension. As we've stated before, the goal of the search is to find the best

position within this space where best means we can use the particle position to construct a solution that best solves our problem.

For our running example, the swarm represents candidate positions, candidate values of x and y. The search moves the swarm through the two-dimensional search space evaluating candidate positions as it goes to try and find the best x and y possible to minimize $f(x, y)$ subject to the boundary condition of $0.01 < x, y < 1$. The difference between types of swarm algorithms is how they move the particles through the search space.

We have the objective function, the swarm size and dimensionality, and the search bounds. The next step is to initialize the swarm. Think of this as scattering the particles throughout the search space in some manner. The most obvious, and often wholly satisfactory, approach, is to scatter the particles randomly within the bounds of the search space. So, for our example here, our initialization step is to assign each particle to randomly selected values in the range $0.01 < x_i, y_i < 1$ where i refers to the i-th particle in the swarm. We'll encounter two other initialization approaches later in the chapter.

The final choice we have to make is when to stop searching. Typically, we use two criteria. The first is to set an upper limit on the number of iterations. Here an iteration is a pass through the `while` loop of Algorithm 2. The second is to test the best position the swarm currently knows of, and if it is within some tolerance, we call it a day and stop searching. For our minimization of $f(x, y)$, we set an iteration maximum of M and a tolerance of θ. Therefore, after each swarm position update and evaluation of the objective function for each new particle position, we check to see if we've performed the maximum number of steps or if our $f(x_{\text{best}}, y_{\text{best}})$ is below our threshold, θ. Recall, the framework always minimizes the objective function. This is not a difficulty in cases where we desire to maximize. In those cases, we negate the objective function value so maximization is now minimization.

Algorithm 2 has the step: *Update the particle positions.* This is the part where the particular swarm optimization algorithm comes into play. Each does this in its own way, as we will see in later chapters where we develop the optimization classes. However, the general form of a search does not change: we still initialize a swarm and step through updating and evaluating it until we're done returning the best position found by the swarm as our solution.

The framework is a collection of classes we'll pass to the individual swarm algorithms. The classes encapsulate concepts like initialization, boundaries, and objective functions. We'll also define ancillary classes used by particular swarm optimizations, like the inertia classes used by the particle swarm optimization (PSO) class of Chapter 4.

To set up a swarm optimization problem, then, we'll use the framework classes to create objects passed to an instance of the desired swarm algorithm. We run the search by calling the `Optimize` method of the swarm object.

Briefly, here are the framework components (classes) developed in this chapter,

- Objective - the objective function used by the swarm. We'll typically implement this class from scratch.
- Bounds - the boundary conditions used by the swarm and by the Initializer.
- Initializer - the particular initialization method used by the swarm. We use this functionality to explore the effect of different initialization approaches.
- Done - the concept of "done". If not used, each swarm algorithm counts iterations and, optionally, looks for a tolerance to be met.

```
class Objective:
    def __init__(self):
        pass
    def Evaluate(self, pos):
        pass
```

Figure 2.1: The most basic Objective class.

- Inertia - the inertial update method used by PSO. This is particular to particle swarm optimization. It can be used to explore the effect of different inertia schedules.

Let's develop the components of the framework beginning with the objective function.

2.2 Objectives

The objective function is key to successfully applying swarm optimization. It is also the thing most tailored to the particular task at hand. We saw above how the objective function might be as simple as evaluating an algebraic function. However, we'll see complex objective functions later in the book that bear no resemblance to an algebraic function.

The goal of the objective function is to measure the quality of each particle position, each possible solution to the problem. The objective function is assumed to return a floating-point value where smaller is better.

In code, the objective function class is quite simple. A skeleton of one is presented in Figure 2.1. There are only two methods we need to fill in. The constructor (`__init__`) can be used to pass any extra information to the objective function when initialized. For example, in Chapter 8, we'll experiment with nonlinear curve fitting. In that case, we'll pass the sampled data points we want to fit to the objective function via the constructor.

The most important method is `Evaluate`. This method is called for each particle on each pass through Algorithm 2. The single argument to `Evaluate` is a particle position vector. In the example of the previous section, where we sought to minimize $f(x, y)$, `pos` would be a two-element NumPy vector where `pos[0]` is x and `pos[1]` is y. The `Evaluate` method's job is to return a single number indicating how good of a solution `pos` represents. Let's define $f(x, y) = xy$ going forward. The `Objective` class becomes,

```
class Objective:
    def Evaluate(self, pos):
        return pos[0] * pos[1]
```

where we need not define the constructor explicitly as we are passing no information used by `Evaluate`.

To use the objective function with a swarm algorithm, we create an instance of the objective function class, passing any information it needs via the constructor, and then pass that instance to the swarm algorithm.

In Section 2.7, we'll see how to put all the framework pieces together to set up an optimization problem. For now, let's move on to defining boundaries.

```
    class Bounds:
        def __init__(self, lower, upper, enforce="clip"):
            self.lower = np.array(lower)
            self.upper = np.array(upper)
            self.enforce = enforce.lower()
        def Upper(self):
            return self.upper
        def Lower(self):
            return self.lower
        def Limits(self, pos):
1:          npart, ndim = pos.shape
2:          for i in range(npart):
                if (self.enforce == "resample"):
3:                  for j in range(ndim):
4:                      if (pos[i,j] <= self.lower[j]) or
                           (pos[i,j] >= self.upper[j]):
                            pos[i,j] = self.lower[j] +
5:                             (self.upper[j]-self.lower[j])*
                                 np.random.random()
                else:
6:                  for j in range(ndim):
                        if (pos[i,j] <= self.lower[j]):
                            pos[i,j] = self.lower[j]
                        if (pos[i,j] >= self.upper[j]):
                            pos[i,j] = self.upper[j]
                pos[i] = self.Validate(pos[i])
            return pos
        def Validate(self, pos):
            return pos
```

Figure 2.2: The Bounds class.

2.3 Boundaries

We said above we'll reimplement the `Objective` class to tailor it to each problem. The `Bounds` class will most often be used as-is or subclassed. The purpose of the `Bounds` class is to ensure swarm positions are kept within a set range.

The `Bounds` class sets the upper and lower values for each dimension of the problem. It also handles checking of these limits and allows, in subclasses, for an extra step we'll use to ensure that not only are values for a particular dimension within the set bounds, they are also valid in terms of what we expect that dimension to represent. For example, we may want to enforce integer values for particular dimensions.

The `Bounds` class is shown in Figure 2.2. The constructor expects two arguments, `lower` and `upper`. These are lists or NumPy vectors with the lower and upper limits for each dimension, respectively. The optional third argument decides what to do if a particular particle's position along a dimension is out of bounds. The default is to clip, to set the dimension to the lower or upper limit. If, however, it is set to "resample", a randomly selected value within the lower and upper limit replaces the offending value. All of this checking is done by the `Limits` method. Note, these methods are called by the swarm algorithm classes; we need not call them ourselves. The `Upper` and `Lower` methods are used to return the upper and lower limits passed in via the constructor. The `Validate` method by default does nothing beyond returning its argument. This is the method we'll

sometimes override in a subclass to enforce special requirements on particle values.

Let's look more closely at the `Limits` method in Figure 2.2. Its argument, `pos`, is a matrix representing the current positions for each particle in the swarm. First, we return the number of particles (rows of `pos`) and the number of dimensions for each particle (columns of `pos`) `(1)`. We then loop over the particles to process them individually `(2)`. How each particle is processed depends on whether we're clipping or resampling. In either case, we loop over the dimensions of the particle `(3)` `(6)`. If clipping, we ask if the current particle's current dimension (`pos[i,j]`) is less than or greater than the limits for that dimension. If so, we set the value to the respective limit. If we're resampling, and we find a particle dimension that exceeds its bounds `(4)`, we replace that value by a randomly selected one that is within limits `(5)`. Regardless of whether we're resampling or clipping, after processing the limits on the current particle's position, we call `Validate`, which, by default, does nothing.

Returning to our running example for this chapter, if we want to minimize $f(x, y)$ over x and y in the range $[0.01, 1]$, we create a `Bounds` object like so,

```
bounds = Bounds([0.01,0.01],[1,1])
```

where we set the lower and upper limits on x and y to be 0.01 and 1.0, respectively. Note, we are implicitly taking the default action to clip particle positions to the lower or upper limit if they go out of bounds.

The `Bounds` class is used by the swarm algorithms and by the initializers, to which we now turn.

2.4 Initializers

The first step in Algorithm 2 says *Initialize the swarm.* This is the part where we set up our initial particle positions within the search space. We already alluded above to one possible way to do this: by randomly scattering the particles within the bounds of the search space. As far as our framework is concerned, whatever object we pass to the swarm algorithms as an initializer needs to support a constructor that accepts the number of particles in the swarm, the dimensionality of the particles, and an optional `Bounds` object to set the limits on the initial positions. It also needs to have an `InitializeSwarm` method that takes no arguments, but returns a NumPy matrix representing initial particle positions. This matrix has as many rows as there are particles in the swarm and as many columns as there are dimensions in the search space. This is the matrix that the algorithms will evolve to search for the best solution to our problem.

We'll define three initializers here and illustrate how they work graphically for a two-dimensional search space. The three are `RandomInitializer`, `QuasirandomInitializer`, and `SphereInitializer`. We'll experiment with these in later chapters. The first does what you might expect; it scatters the swarm particles randomly within the bounds of the search space. The second is similar, but instead of a pseudorandom generator, it uses a quasirandom generator. Quasirandom generators are space-filling, meaning they distribute the particles more uniformly throughout the search space. The `SphereInitializer` places the particles on the edge of a hypersphere bounded by the search space. The idea here is to put the particles near the edges so that the likely solution is within the hypersphere.

```
class RandomInitializer:
    def __init__(self, npart=10, ndim=3, bounds=None):
        self.npart = npart
        self.ndim = ndim
        self.bounds = bounds
    def InitializeSwarm(self):
        if (self.bounds == None):
            self.swarm = np.random.random((self.npart, self.ndim))
        else:
            self.swarm = np.zeros((self.npart, self.ndim))
            lo = self.bounds.Lower()
            hi = self.bounds.Upper()
            for i in range(self.npart):
                for j in range(self.ndim):
                    self.swarm[i,j] = lo[j] +
                      (hi[j]-lo[j])*np.random.random()
            self.swarm = self.bounds.Limits(self.swarm)
        return self.swarm
```

Figure 2.3: The RandomInitializer class.

The code for `RandomInitializer` is in Figure 2.3. The constructor (`__init__`) accepts the number of particles in the swarm (`npart`), the dimensionality of the search space (`ndim`), and, optionally, a `Bounds` object to define the limits of the search space. These values are stored in the instance for use by `InitializeSwarm`.

Initialization happens when `InitializeSwarm` is called. If no `Bounds` object was given, initialization is particularly simple; we just return a NumPy matrix of random values in the range $[0, 1)$. This matrix has `npart` rows and `ndim` columns.

If a `Bounds` object was given, we first define the matrix representing the swarm, extract the per dimension lower (`lo`) and upper (`hi`) bounds from the `Bounds` object and then loop over particles and the dimensions of the particles selecting random values in the bounds for each.

The call to `Limits` seems superfluous at first since we just finished selecting values for the swarm particles that we know are within the limits given by the `Bounds` object. However, this is not the whole story. If we look back at Figure 2.2, we see that the `Limits` method, after checking that the entire swarm is in bounds, calls `Validate`. If the object passed to `RandomInitializer` is a subclass of `Bounds`, it is possible that the subclass implements this method. Hence, the call to `Limits` ensures that `Validate` will be called on the newly initialized swarm.

The two other initializer classes, `QuasirandomInitializer` and `SphereInitializer`, implement the same methods as `RandomInitializer` but tailor the assignment of values. Let's start with `QuasirandomInitializer` as shown in Figure 2.4.

A Halton process can generate a quasirandom sequence. This process is implemented in Figure 2.4 in the `Halton` method. This method returns the i-th Halton number for the given base, b.[1] Halton sequences work best when the bases are primes, so in `__init__` we set up a table to have a different prime for each dimension (`self.primes`). The constructor accepts the number of particles (`npart`), dimensionality (`ndim`), and a `Bounds` object as before. It also optionally accepts an initial index into the Halton sequence (`k`) and a jitter

[1] See p. 75, Section 2.9, *Random Numbers and Computers*, Kneusel, Springer 2018.

```
    class QuasirandomInitializer:
        def Halton(self, i,b):
            f = 1.0
            r = 0
            while (i > 0):
                f = f/b
                r = r + f*(i % b)
                i = floor(i/float(b))
            return r
        def __init__(self, npart=10,ndim=3,bounds=None,k=1,jitter=0.0):
            self.npart = npart
            self.ndim = ndim
            self.bounds = bounds
            self.k = k
            self.jitter = jitter
            self.primes = [
                 2,  3,  5,  7, 11, 13, 17, 19, 23, 29,
                31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
                73, 79, 83, 89, 97,101,103,107,109,113,
               127,131,137,139,149,151,157,163,167,173,
               179,181,191,193,197,199,211,223,227,229,
               233,239,241,251,257,263,269,271,277,281,
               283,293,307,311,313,317,331,337,347,349,
               353,359,367,373,379,383,389,397,401,409,
               419,421,431,433,439,443,449,457,461,463,
               467,479,487,491,499,503,509,521,523,541,
               547,557,563,569,571,577,587,593,599,601,
               607,613,617,619,631,641,643,647,653,659]
        def InitializeSwarm(self):
            self.swarm = np.zeros((self.npart, self.ndim))
            if (self.bounds == None):
                lo = np.zeros(self.ndim)
                hi = np.ones(self.ndim)
            else:
                lo = self.bounds.Lower()
                hi = self.bounds.Upper()
1:          for i in range(self.npart):
                for j in range(self.ndim):
                    h = self.Halton(i+self.k,self.primes[j %
                                    len(self.primes)])
                    q = self.jitter*(np.random.random()-0.5)
2:                  self.swarm[i,j] = lo[j] + (hi[j]-lo[j]) * h + q
            if (self.bounds != None):
                self.swarm = self.bounds.Limits(self.swarm)
            return self.swarm
```

Figure 2.4: The QuasirandomInitializer class.

```
class SphereInitializer:
    def __init__(self, npart=10, ndim=3, bounds=None):
        self.npart = npart
        self.ndim = ndim
        self.bounds = bounds
    def InitializeSwarm(self):
        self.swarm = np.zeros((self.npart, self.ndim))
        if (self.bounds == None):
            lo = np.zeros(self.ndim)
            hi = np.ones(self.ndim)
        else:
            lo = self.bounds.Lower()
            hi = self.bounds.Upper()
        radius = 0.5
        for i in range(self.npart):
            p = np.random.normal(size=self.ndim)
            self.swarm[i] = radius + radius* p / np.sqrt(np.dot(p,p))
        self.swarm = np.abs(hi-lo)*self.swarm + lo
        if (self.bounds != None):
            self.swarm = self.bounds.Limits(self.swarm)
        return self.swarm
```

Figure 2.5: The SphereInitializer class.

value (`jitter`). The jitter value is used to slightly adjust the actual value returned by the Halton sequence to prevent each initialization from being the same. By default, we don't use jitter and start the Halton sequence from one.

The `InitializeSwarm` method creates the swarm matrix and gets the bounds from the `Bounds` object or uses $[0, 1)$ if no `Bounds` object is given. All the action happens in the dual loops `(1)`. We loop over the particles (`i`) and the dimensions (`j`) where for each particle and dimension, we get the next value of the Halton sequence and any jitter (`q`). The actual swarm position is set as a random value in the range for that dimension `(2)`. Note, like a pseudorandom generator, the Halton sequence is bounded to $[0, 1)$, so we can use it directly as we did for the random case (see Figure 2.3). Finally, if we have a `Bounds` object, we call `Limits` to let the `Validate` method run.

The `SphereInitializer` selects random points on an `ndim`-dimensional hypersphere. To select a p-dimensional vector on a hypersphere, we need to select p values from a Gaussian distribution and scale them by the norm of the resulting vector.[2] The code for the `SphereInitializer` class is in Figure 2.5.

The constructor is identical to that of the `RandomInitializer`. All the fun happens in the `InitializeSwarm` method. After selecting the bounds, we set `radius = 0.5` to construct the set of points in $[0, 1)$. After all swarm points have been sampled, we'll scale them from $[0, 1)$ to the bounds limits in `lo` and `hi`. This sets the size of the hypersphere so that it will fit within the bounds of all dimensions. We build the swarm, one particle position vector at a time using the loop over `npart`. First, we select the vector, `p`, of `ndim` dimensions, from a Gaussian with zero mean and standard deviation of one. We then apply the hypersphere transformation scaling by the radius. Note, we add in the middle point, 0.5, to shift the hypersphere to center on 0.5. Finally, we get the actual particle positions

[2] See Muller, M. E. "A Note on a Method for Generating Points Uniformly on N-Dimensional Spheres." Comm. Assoc. Comput. Mach. 2, 19-20, Apr. 1959.

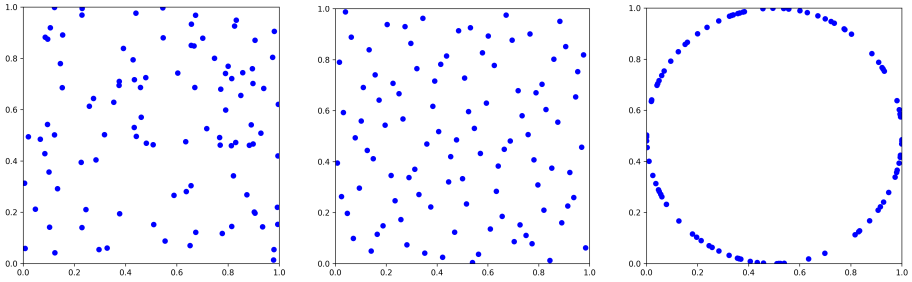


Figure 2.6: Initialized swarms using random (left), quasirandom (middle), and hypersphere initialization (right). Each swarm contains 100 particles.

by scaling the $[0, 1)$ positions by the per dimension range (`np.abs(hi-lo)`) and add in the lower bounds. As before, if we have a `Bounds` object, we call `Limits` before returning the newly initialized swarm matrix.

Let's trade code for plots to see what these initializers are doing. We'll initialize swarms of 100 particles in two dimensions so we can plot them. The way to get the initialized swarms is straightforward,

```
>>> import numpy as np
>>> from RandomInitializer import *
>>> from QuasirandomInitializer import *
>>> from SphereInitializer import *
>>> r = RandomInitializer(npart=100, ndim=2).InitializeSwarm()
>>> q = QuasirandomInitializer(npart=100,ndim=2).InitializeSwarm()
>>> s = SphereInitializer(npart=100, ndim=2).InitializeSwarm()
```

Each of the swarms (`r`, `q`, and `s`) is a 100x2 matrix. We plot the particle positions using the first column as the x-coordinate and the second as the y-coordinate to show us how the initial particles are scattered throughout the two-dimensional search space. This leads to Figure 2.6.

On the left, we see the randomly initialized swarm. It looks pretty random, as it should. We also see that it is not uniform over the two-dimensional search space. There are areas of concentrated particles and voids. In the middle of Figure 2.6, we see the quasirandomly initialized swarm. Here the distribution of particles is much more uniform. Finally, on the right, we see the spherically initialized swarm. All of the particles are on a 2D sphere (circle) within the default $[0, 1)$ bounds.

2.5 Are We Done Yet?

Each swarm algorithm knows how to stop the search when the maximum number of iterations through Algorithm 2 has been reached or when the objective function value for the best particle position known by the swarm is below a given threshold. However, if the swarm algorithm was passed an instance of a class supporting a `Done` method, it calls the method passing in information about the state of the search. The `Done` method returns a boolean value. If it returns `True`, the search is completed. If it returns `False`, the search continues. The information passed to the `Done` call includes the list of current swarm best positions found and their associated objective function values. It also includes the current position of each swarm particle, the maximum number of iterations set, and the current

iteration value.

For the majority of our experiments, we'll simply use the intrinsic checks inside the swarm algorithms to decide if the search is done or not. An exception is when we explicitly iterate through the swarm algorithms by calling the `Step` method. In those cases, we'll call the `Done` method directly. We'll see exactly what that means when we discuss the individual swarm algorithms in later chapters.

We'll end this chapter with an example showing how to use the framework to set up and execute an optimization search, but, before we do that, let's take a quick look at the last part of the framework, the inertia class used by PSO.

2.6 Inertia

While we do not yet know the details of the PSO class, we know it includes a parameter, the inertia, which is generally changed during the search. Historically, this value is changed linearly with increasing iteration number so it gets smaller the longer the search has been run. We'll define two inertia classes for PSO, though it is straightforward after seeing how they are implemented to design your own. These are the `LinearInertia` and `RandomInertia` classes.

The purpose of the inertia objects is to return a single floating-point number in the range $[0, 1]$, which is typically labeled `w` or ω in the PSO literature. We'll see in Chapter 4 precisely what this ω represents, but for now, we'll content ourselves with the interface the PSO class expects from any inertia object passed to it.

The `LinearInertia` class is,

```
class LinearInertia:
    def __init__(self, hi=0.9, lo=0.6):
        if (hi > lo):
            self.hi = hi
            self.lo = lo
        else:
            self.hi = lo
            self.lo = hi
    def CalculateW(self, w0, iterations, max_iter):
        return self.hi - (iterations/max_iter)*(self.hi-self.lo)
```

The constructor accepts upper (`hi`) and lower (`lo`) limits for ω, both scalar values. A bit of code ensures that `hi` $\geq$ `lo`. The main method of the class is `CalculateW`, which the `PSO` class calls. The arguments are `w0`, an initial ω value defined when the `PSO` object is created, the current number of swarm iterations completed (`iterations`), and the maximum number of iterations (`max_iter`). For `LinearInertia`, we want ω to decrease linearly from `hi` to `lo` over `max_iter` iterations. The code in `CalculateW` does this by subtracting an ever increasing fraction of the difference between `hi` and `lo` from `hi` so when the search starts, ω is `hi` and when the search ends, ω is `lo`. Typical literature values for `hi` and `lo` are 0.9 and 0.5, respectively. If you are familiar with neural networks, ω acts in much the same way as momentum during gradient descent.

If, instead of a constantly decreasing ω, we want a random value, we can use the `RandomInertia` class. Structurally, it is identical to `LinearInertia` except for the actual equation used to return ω in `CalculateW`. In this case, the return value is,

```
return 0.5 + random.random()/2.0
```

which returns a random value in the range $[0.5, 1.0)$.

2.7 Setting Up An Optimization

Throughout this chapter we've made use of a running example, that of finding the minimum of $f(x, y) = xy$ for x and y in $[0.01, 1]$. Now that we have a developed framework to support the swarm optimization algorithms, let's set up and run a search using the yet-to-be-defined `PSO` class. Of course, given our bounds on x and y, we already know the answer we should get: $x = 0.01$ and $y = 0.01$. Let's see if the PSO algorithm agrees.

To set up the problem, we need an objective function class, an instance of the `Bounds` class, an initializer (we'll use `RandomInitializer`), and, since we're using PSO, an inertia object. We'll use the `LinearInertia` class. We'll list the code in pieces. The actual code is in the file `fxy.py` in the source code available with this book.

First, we import NumPy and the framework components,

```
import numpy as np
from PSO import *
from LinearInertia import *
from Bounds import *
from RandomInitializer import *
```

Next, we define our objective function class and make an instance of it,

```
class Objective:
    def Evaluate(self, pos):
        return pos[0]*pos[1]
obj = Objective()
```

where we see how simple an objective function can be.

We now define the parameters of the search, including the bounds, initializer, and inertia,

```
npart = 10
ndim = 2
m = 100
tol = 1e-4
b = Bounds([0.01,0.01], [1,1])
i = RandomInitializer(npart, ndim, bounds=b)
t = LinearInertia()
```

where `npart` is the number of particles in the swarm, `ndim` is the number of dimensions for each particle, here two because we are looking for x and y to minimize $f(x, y)$. We set the maximum number of iterations to 100 (`m`) and the tolerance to 0.0001 (`tol`). Recall, this means that the search will run for at most 100 swarm updates or stop if the best position found by the swarm returns an objective function value less than 0.0001.

We then define the bounds for the problem, `b`, to keep both x and y inside $[0.01, 1]$. The first argument to `Bounds` is a list of the lower limit for each dimension, the second is for the upper limit. Next, we set up the random initializer (`i`). We tell it how many particles (`npart`), how many dimensions for each (`ndim`), and pass in the bounds instance so it knows the range of values it can use. Finally, `t` defines the default linear inertia

object which will start the PSO ω parameter at 0.9 and decrease it linearly to 0.5 over the iterations of the search.

We are now ready to define the actual PSO swarm and run the optimization. The code required is particularly straightforward given the framework,

```
swarm = PSO(obj=obj, npart=npart, ndim=ndim, init=i, tol=tol,
          max_iter=m, bounds=b, inertia=t)
swarm.Optimize()
```

The first line constructs the `swarm` by passing in the objective function (`obj`), number of particles (`npart`), their dimension (`ndim`), the initializer (`i`), tolerance (`tol`), maximum number of iterations (`m`), the bounds (`b`), and the inertia (`t`). The actual search becomes a single call to the `Optimize` method. This method, defined virtually identically for each of the swarm algorithms we'll discuss in later chapters is,

```
def Optimize(self):
    self.Initialize()
    while (not self.Done()):
        self.Step()
    return self.gbest[-1], self.gpos[-1]
```

which follows Algorithm 2 quite closely. The swarm is initialized using the initializer object passed to it (`Initialize`), then a loop searches until `Done` returns `True`. The `Step` method performs the required swarm position updates and evaluations of the objective function. When the search is over, the best objective function value (`gbest[-1]`) and associated position in the search space (`gpos[-1]`) are returned. These are lists tracking the evolution of the search so the last element of the list is the best position found. We'll use these lists later to show how the search evolved. In our example call to `Optimize` above, we ignored these return values. That's because we can get more information by calling the `Results` method,

```
res = swarm.Results()
x,y = res["gpos"][-1]
g = res["gbest"][-1]
print("f(%0.8f, %0.8f) = %0.8f" % (x,y,g))
print("(%d swarm best updates, %d iterations)" %
    (len(res["gbest"]), res["iterations"]))
```

The `Results` method returns a Python dictionary containing the `gbest` and `gpos` values along with other information like the number of iterations performed. The number of elements in the `gbest` list is the number of times the swarm improved its best solution during the search.

If we run this code, we'll get slightly different answers each time because of the stochastic nature of the initialization process, but, because our search space is straightforward, we'll always find the true minimum position of (0.01, 0.01), though the number of swarm improvements needed will vary. For example, a single run produced this output,

```
f(0.01000000, 0.01000000) = 0.00010000
(8 swarm best updates, 100 iterations)
```

showing that the minimum position was found, that all 100 iterations were used, and there were eight times that the swarm improved its initial best position. By displaying the

positions (`res["gpos"]`) and objective function values (`res["gbest"]`), we can see how the swarm moved to the best solution,

```
f(0.96153452, 0.01689004) = 0.01624035
f(1.00000000, 0.01000000) = 0.01000000
f(0.72279699, 0.01000000) = 0.00722797
f(0.66630552, 0.01000000) = 0.00666306
f(0.39462535, 0.01000000) = 0.00394625
f(0.37345245, 0.01000000) = 0.00373452
f(0.15094400, 0.01000000) = 0.00150944
f(0.01000000, 0.01000000) = 0.00010000
```

where the first line is the best position found by the initial swarm. The PSO algorithm then improved the initial swarm to progressively find better and better locations until the true minimum was found. Notice that in this particular run, y converged to is limit very quickly.

You may have a bit of an uneasy feeling about this example. Yes, it was contrived, but especially so because we told the `Bounds` object to use clipping when a particle dimension went out of bounds, and it so happens that the proper value for our minimum is on the boundary in this case. This is why the y value so quickly found the edge. What if we change the boundary conditions to use resampling instead? The change to the code is,

```
b = Bounds([0.01,0.01], [1,1], enforce="resample")
```

with everything else the same. If we now run the search, we see a different result,

```
f(0.07176122, 0.01060304) = 0.00076089
(4 swarm best updates, 100 iterations)
```

where the quick convergence we had before has disappeared. If we increase the number of swarm iterations from $m = 100$ to $m = 1000$, we get closer,

```
f(0.01084622, 0.02919965) = 0.00031671
(8 swarm best updates, 1000 iterations)
```

Here, we've told the algorithm to use a small swarm size (10 particles) and search for a long time (1,000 iterations). As we work with the algorithms, we'll build intuition and see that 10 particles are probably too few, so let's increase the number of particles to 100 (`npart=100`) while still using resampling on the `Bounds` object. We'll leave the number of iterations at $m = 100$, as before. A search with this larger swarm gives us,

```
f(0.02181587, 0.01145074) = 0.00024981
(3 swarm best updates, 100 iterations)
```

which is a better result. Notice, the smaller swarm for more iterations and the larger swarm for fewer iterations both performed the same amount of work: 10,000 objective function evaluations after initialization since $10 \times 1000 = 100 \times 100 = 10,000$. The stochastic nature of the random initializer means that multiple runs of each of these swarms will oscillate between which one returns a better result.

In the following chapters, we'll define multiple swarm search algorithms. Some, like PSO, are swarm intelligence algorithms, while others, like differential evolution (DE), are really evolutionary algorithms in disguise. The point of the framework introduced in this chapter is to make switching and experimenting with these algorithms straightforward.

Let's replace the `PSO` object with `DE` for the exercise above where we are using 10 particles for 100 iterations. After importing the DE module, we need only replace the call to `PSO` with,

```
swarm = DE(obj=Objective(), npart=npart, ndim=ndim, init=i,
           tol=tol, max_iter=m, bounds=b)
```

and run the code again. Notice that the `DE` class does not use an inertia object. This run produces,

```
f(0.01000140, 0.01000028) = 0.00010002
(31 swarm best updates, 100 iterations)
```

which is a still better result, about as good as we could hope to get. This illustrates a general observation about differential evolution: it seems particularly good at rapidly converging when the search space is relatively simple. However, as we'll see later in the book, more sophisticated search spaces are sometimes best searched with an algorithm like PSO, where an algorithm like DE is likely to get stuck in a poor position in the search space. We need to be aware of multiple algorithms and their relative strengths and weaknesses. As you work through the book, you'll develop the intuition you need to know when to try which algorithm, though experimentation is vital. Always indulge your "what if?" questions.

In this chapter, we defined our framework and we demonstrated its use on a simple example. Let's move ahead to investigate our first swarm algorithm, one so simplest it isn't really a swarm intelligence algorithm at all: random optimization.

Chapter 3

Random Optimization

We begin our investigation of swarm algorithms with the simplest of all, *random optimization* (RO). Random Optimization is so simple that while it is, technically, a swarm algorithm, it is not a swarm intelligence algorithm because the swarm exhibits no intelligence, it's purely random.

This chapter introduces the concepts behind the algorithm (Section 3.1). Then, we build the Python class using the framework pieces of Chapter 2 to implement the algorithm (Section 3.2) and do a few basic tests to demonstrate how it is working (Section 3.3).

3.1 Good Enough For Now

Recalling our mental picture of optimization as a search by the swarm through a space of possible solutions, random optimization is akin to a set of prospectors, each searching for gold by doing his or her thing in separate parts of the search space. In this sense, random optimization is a purely local search approach. The individual particles search their immediate region, but they never communicate with each other to see if there is a better place to search somewhere else in the search space, hence, there is no global intelligence to the algorithm.

Algorithm 4 lays out a random search. Naturally, it closely resembles what we saw in Algorithm 2 of Chapter 2, but we've filled in some of the details a bit to tailor the algorithm to RO.

As before, we need framework parts for the objective function, the bounds, and the type of initialization we want to use. Then, the `for` loop initializes the swarm by scattering the particles throughout the search space according to the initializer we decided to use.

The swarm searches individually by letting each particle perform an independent search while checking after each particle move whether or not the new position is the best the entire swarm as a whole has seen. The critical point in this algorithm that separates it from all the others we'll investigate in later chapters is the particles *do not* communicate with each other. There is no sharing of information that might cause one particle to move to a different region of the search space based on what another particle has learned. Regardless, with a proper swarm size and initialization, this approach can be surprisingly effective.

The concept behind RO is straightforward enough, but there is that one line in Algorithm 4,

Select a new position some random distance away from the current position

Algorithm 4 The random optimization algorithm.

Input: An objective function, bounds, and initialization type
Output: The best position found by the swarm
  for each particle **do**
    Select an initial position within the bounds of the search space
    Evaluate the objective function at this position
    Mark this position as the best found by the particle so far
  end for
  Store the best initial particle position as the swarm global best position
  while not done **do**
    for each particle **do**
      Select a new position some random distance away from the current position
      Evaluate the new position
      if fitness of new position < fitness of current position **then**
        Move to the new position
        if new position fitness < swarm best position fitness **then**
          Store the new global best
        end if
      end if
    end for
    Increment the iteration counter
  end while

What should we make of this? What does "at random" and "some distance away" actually mean?

By "at random" we mean just that, we select a new position relative to the current position by selecting a random offset vector to add to the current position vector. There are multiple ways to do this, but ultimately, we need a random offset vector to add to the current particle's position to move to a new position in the search space. As we lack any outside information to tell us that moving from the current particle position to a new position in a particular direction is any better than moving in any other direction, we can pick the offset vector freely. At the same time, we lack any information to tell us how far we should move in these steps, so we might as well make most steps relatively small, but allow for the possibility of selecting a large step from time to time.

Selecting a set of random values that are usually small but sometimes larger speaks to a Gaussian or normal distribution. For example, Figure 3.1 shows a histogram of samples from a standard normal distribution, one with a mean of zero and a standard deviation of one. The data for the figure is easy to generate in Python,

```
import numpy as np
import matplotlib.pylab as plt
d = np.random.normal(size=(200000,))
h,x = np.histogram(d, bins=100)
h = h / h.sum()
plt.plot(x[:-1],h,color='b')
plt.show()
```

Here, we select 200,000 random values from $N(0,1)$ (`d`) and then plot a histogram of those values. Each time the code is run, we'll get a slightly different plot, but it will always have

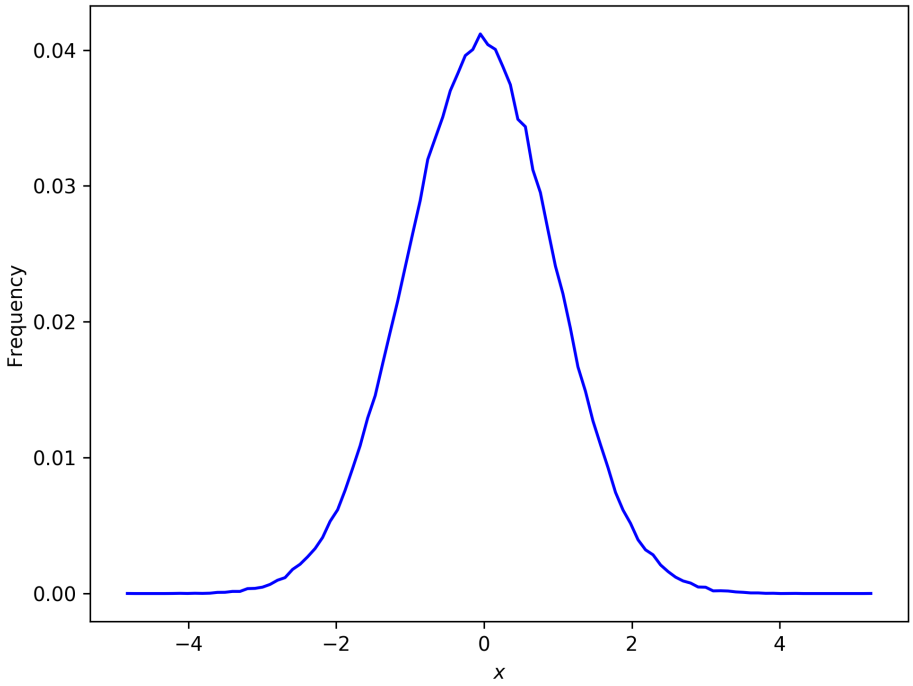


Figure 3.1: A histogram of samples from a standard normal curve.

much the same appearance and be like the plot shown in Figure 3.1.

In Figure 3.1, we see that the vast majority of samples are very close to zero. This satisfies the desire to select new candidate positions for our swarm that aren't too far from where the particle currently is. However, with decreasing likelihood, we'll sometimes get output that is much further from zero. This satisfies the desire to occasionally make a larger jump to explore elsewhere.

We see that the output is, to a high degree, contained within the range $[-5, 5]$. So, if we divide our selected samples by five, we'll be close to selecting in the range $[-1, 1]$. We want both positive and negative values to make the offset vector point in all directions.

We still haven't addressed what we mean by "some distance away". We're close, though. We know we want an offset vector with specific characteristics, and we see how to get it, but we'd like to control the relative distance or size of the vectors, too. To do that, we introduce a parameter, η, which we'll use as a scale factor. We pass this scale factor in at swarm creation to adjust for a particular problem, but we'll see that often we don't need to adjust η at all.

Let's get specific. If the current particle's position is p, where we know that p is an n-dimensional vector (here n is `ndim` from Chapter 2), we can select a new candidate position like so,

$$p' = p + \eta p q / 5$$

where q is the offset vector made up of random draws from $N(0, 1)$ for each of the n dimensions in p. The size of the jump is controlled by η as a fraction of the current position of the particle, p. Of course, we must ensure that the new candidate position, p', is within the bounds of the search space. We'll see how in Section 3.2 when we develop the code for the RO class.

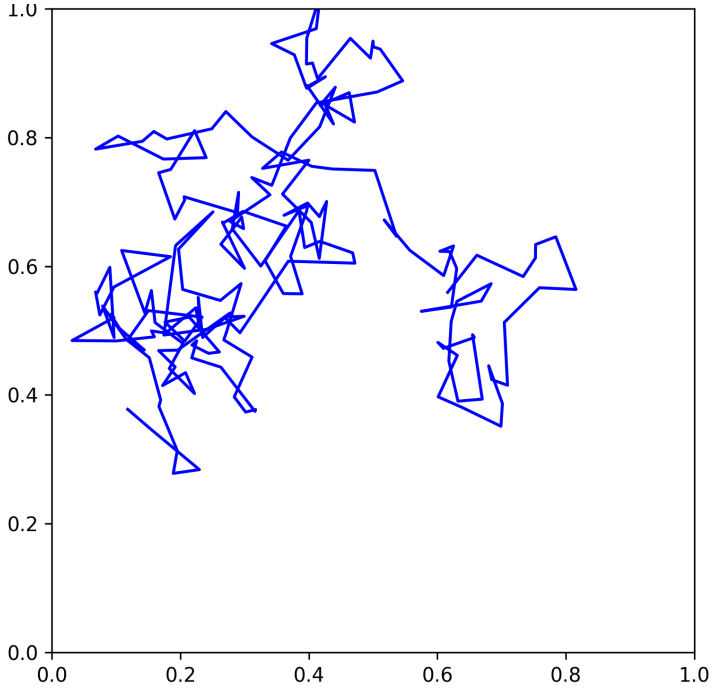


Figure 3.2: A 2D random walk of 300 steps using the RO candidate update equation.

Next, we evaluate the objective function at the new candidate position, p'. If the objective function value at the candidate position is less than the objective function value of the particle's current position, we found a better place to be, so we update the particle's position, $p \leftarrow p'$ and store the associated objective function value. However, if the candidate position is *not* better, we stay where we are until we find someplace better to go.

In Algorithm 4, we evaluate candidate positions for each of the particles to make up one iteration of the swarm. Whenever we find a new position, we also check to see if this position is better than the global best position known by the swarm. If so, we update that position as well. It is the global best position which decides whether the tolerance has been met and is returned when the search concludes.

The elegance of RO is its simplicity and the implicit invitation to expand the algorithm to make it more sophisticated. With RO as a base, it is no wonder so many researchers have developed their different approaches, either to RO updates or to entirely new algorithms where the swarm *does* pass information among itself to converge more rapidly or reliably. For example, we're using a normal curve to select new candidate locations. It isn't too hard to imagine replacing the normal curve with something else, something with parameters that could be adjusted for the task at hand. A beta distribution, perhaps?

Let's generate a two-dimensional walk for a particle using our update equation to see how it might explore the search space. We'll limit the space to $[0, 1]$ and set $\eta = 0.2$. The result is Figure 3.2, where the path shows how the particle moved through space. In terms of the RO algorithm, this path shows cases where the particle found a new, better position and moved to it. Most jumps are small, corresponding to the small offset vector we'd expect from the distribution of Figure 3.1. However, some are large, so the particle does have the opportunity to move through the search space to explore new regions while still spending time looking closely around its current position. The path starts near the center of the

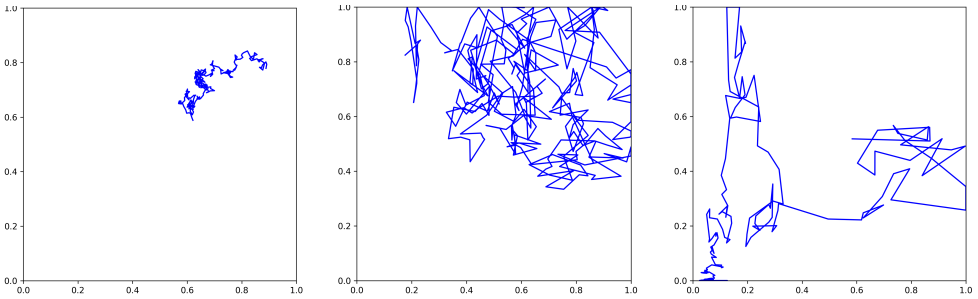


Figure 3.3: Random walks of 300 steps for $\eta = 0.05$ (left), $\eta = 0.5$ (middle), and $\eta = 0.8$ (right).

search space so we can track the motion more easily. The code to generate the random walk is,

```
def candidate(p, eta):
    return p + eta*p*np.random.normal(size=2)/5.0

eta = 0.2
d = []
p = [0.5,0.5] + np.random.random(2)/8
d.append(p)
for i in range(300):
    p = candidate(p, eta)
    p = np.clip(p,0,1)
    d.append(p)
d = np.array(d)
```

where `candidate` returns a new "candidate" position according to our update equation. For the walk, we always move to the candidate position. The current particle position is in `p`, and the path is tracked in `d`, which is turned into a 2D NumPy array for plotting in the final line. The `np.clip` function acts like a call to the `Limits` method of a `Bounds` object.

The RO algorithm has only one adjustable parameter, η. Figure 3.3 shows the effect of different values of η where on the left $\eta = 0.05$, in the middle $\eta = 0.5$, and on the right $\eta = 0.8$. When η is large, the particle jumps around quickly but does not closely inspect its local region before moving on. When η is small, the particle explores the local region without jumping to more distant regions of the search space. As we'll see in Section 3.2, the RO class uses a default value of $\eta = 0.1$ as a compromise between local and regional searching.

Look again at Figure 3.3 and picture the overlapping paths from a swarm of particles, each one taking a similar random walk. This is what the RO algorithm will do, so we have some reason to hope that this approach, as unguided as it is, isn't entirely doomed to failure. The overlapping paths will explore the space, eventually. With the right compromise between local exploitation and regional exploration, we might expect the swarm, or at least parts of it, to collapse into the best region of the search space. This implies a possible strategy is to set η to some value like 0.1 or 0.2 initially, and as the swarm evolves to make η smaller to force more local searching to fine-tune the optimization. In effect, this is what the ω parameter of PSO, which we saw in Chapter 2, does. We'll see explicitly how in Chapter 4.

```
class RO:
    def __init__(self, obj,
                 npart=10,
                 ndim=3,
                 max_iter=200,
                 eta=0.1,
                 tol=None,
                 init=None,
                 done=None,
                 bounds=None):
    def Results(self):
    def Initialize(self):
    def Done(self):
    def Evaluate(self, pos):
    def CandidatePositions(self):
    def Step(self):
    def Optimize(self):
```

Figure 3.4: Skeleton of the RO class.

Parameter	*Description*
`obj`	Objective function object
`npart`	Number of particles in the swarm
`ndim`	Number of dimensions in the search space
`max_iter`	Maximum number of swarm iterations
`eta`	Step size parameter
`tol`	Tolerance value
`init`	Initializer object
`done`	Done object
`bounds`	Bounds object

Table 3.1: The arguments to the RO class constructor.

Let's move on from algorithm description to implementation and build the RO class.

3.2 The RO Class

The `RO` class is an archetype for the swarm algorithms in following chapters. In each case, we'll present the skeleton of the class, the methods, and then fill in the source code for the methods with description. Most of the algorithms we implement in code have at least the methods we see in the `RO` class, though how the methods work varies somewhat from algorithm to algorithm. All, however, use the framework objects we defined in Chapter 2. Note, the code listings have been made more compact for presentation purposes. The actual source code file, `RO.py`, contains appropriate comments and spacing.

Figure 3.4 shows the skeleton of the `RO` class. There are only a handful of methods. One point of this book is to help you see what these approaches have in common and how many are just variations on a theme.

The constructor (`__init__`) accepts nine possible arguments. The constructor is the interface to the framework parts. The arguments are as shown in Table 3.1.

Except for η, all of these parameters are present in the constructor argument lists for each of the swarm algorithms. We already know what these parameters do. If the default value is `None`, the `RO` class supplies default functionality. For example, the `done` argument will typically be `None` to use the simple case of counting swarm iterations or meeting the tolerance value. If no tolerance value is given, the search stops after completing all iterations. The code for the constructor is,

```
def __init__(self, ...):
    self.obj = obj
    self.npart = npart
    self.ndim = ndim
    self.max_iter = max_iter
    self.init = init
    self.done = done
    self.bounds = bounds
    self.tol = tol
    self.eta = eta
    self.initialized = False
```

where the arguments to the constructor, or their default values, are stored as member variables. The final line sets `initialized` to `False`. This is used to prevent calling methods before actually initializing the swarm.

Let's now implement each of the remaining methods of the `RO` class. We'll do so in a top-down fashion beginning with `Optimize`.

3.2.1 Optimize

The `Optimize` method is typically how we'll use the `RO` class. It is a literal implementation of Algorithm 2,

```
def Optimize(self):
    self.Initialize()
    while (not self.Done()):
        self.Step()
    return self.gbest[-1], self.gpos[-1]
```

Before we can use the swarm, we need to set up the initial conditions, so, `Optimize` calls `Initialize` (see Section 3.2.2). With the swarm initialized, we start iterating where each iteration step selects possible candidate positions, moves to them if they are better than the current positions, and updates the global best found by the swarm. All of this is inside the `Step` method. The `while` loop runs until `Done` returns `True`.

When the search ends, the list representing the sequence of global best objective function values is in `gbest`, and the corresponding particle positions in `gpos`. Therefore, `Optimize` returns these values to the caller, though, as we saw in Chapter 2, we can get this information and more by calling the `Results` method.

Note, the methods of the class are public, meaning we can call any of them from external code. While using `Optimize` is easy to do, there will be times when we call `Initialize` and loop calling `Step` so we can interrogate the swarm as it evolves.

Let's now look at how the swarm is initialized.

3.2.2 Initialize

The RO constructor was passed an initializer object to set up the initial particle positions (see Section 2.4). The Initialize method uses this object and sets up additional housekeeping to track the evolution of the swarm. In code,

```
def Initialize(self):
    self.initialized = True
    self.iterations = 0
    self.pos = self.init.InitializeSwarm()
    self.vpos= self.Evaluate(self.pos)
    self.gidx = []
    self.gbest = []
    self.gpos = []
    self.giter = []
    self.gidx.append(np.argmin(self.vpos))
    self.gbest.append(self.vpos[self.gidx[-1]])
    self.gpos.append(self.pos[self.gidx[-1]])
    self.giter.append(0)
```

where we set initialized to True. We also set the iteration counter (iterations) to zero.

The particle swarm is stored in pos, a NumPy matrix with npart rows and ndim columns. Therefore, each row of pos represents a single swarm particle's current position in the ndim-dimensional search space. We get the initial positions by calling the InitializeSwarm method of the initializer passed to the RO constructor.

Each particle is located somewhere in the search space. The position in the search space translates into a particular value of the objective function. So, we need to know not only the current position of the particle, but also the current value of the objective function for that position. We'll store the objective function values in vpos. To initialize the search, we need to evaluate the initial positions and keep their objective function values. This is accomplished by the call to Evaluate, which we'll define below in Section 3.2.6. As the objective function values are scalars, vpos is a NumPy vector of npart elements. Therefore, if we want to know the status of particle i, we get its position by asking for pos[i] and the objective function value at that position by asking for vpos[i].

The next four lines set up lists to track the evolution of the swarm search. These values are updated together each time a new global best is located. We track the particle number of the new best (gidx), the objective function value (gbest), the position (gpos), and the iteration number when it was found (giter). With this approach, we get the current best objective function value of the swarm by asking for gbest[-1] and the position of the best with gpos[-1]. Or, if we'd rather track the evolution, we can walk through the list and see how the swarm moved to its final best position.

3.2.3 Step

A single swarm iteration is captured in the Step method,

```
def Step(self):
    new_pos = self.CandidatePositions()
    p = self.Evaluate(new_pos)
    for i in range(self.npart):
        if (p[i] < self.vpos[i]):
            self.vpos[i] = p[i]
```

```
            self.pos[i] = new_pos[i]
        if (p[i] < self.gbest[-1]):
            self.gbest.append(p[i])
            self.gpos.append(new_pos[i])
            self.gidx.append(i)
            self.giter.append(self.iterations)
    self.iterations += 1
```

First, we get a set of new candidate positions for each particle in the swarm, `CandidatePositions`. This sets `new_pos` to a matrix the same size as `pos`. There is a one-to-one correspondence between `new_pos` and `pos`. This means that the current position of particle 17 is in `pos[17]` while the new candidate position is in `new_pos[17]`. We next evaluate the objective function for each of the new candidate positions and put these values in the vector `p`.

Then, we loop over each particle in the swarm. We first ask whether the candidate objective function value for the current particle, `i`, is less the current objective function value, `vpos[i]`. If it is, we move to the new position by updating both `vpos[i]` and `pos[i]`.

Next, we ask the same question about the current particle's candidate position and the best position the swarm has current knowledge of, `gbest[-1]`. If the new position is better, we *append* the new objective function value to `gbest` and append the position to `gpos`. Similarly, we update `gidx` with the current particle number and add the current iteration number to `giter`. When all particles have been processed, the swarm update step is completed, so we increment `iterations`.

3.2.4 Done

Checking whether or not the search is complete means asking a few questions about what has been handed to the `RO` class when the instance was constructed. Specifically, the code is,

```
def Done(self):
    if (self.done == None):
        if (self.tol == None):
            return (self.iterations == self.max_iter)
        else:
            return (self.gbest[-1] < self.tol) or
                   (self.iterations == self.max_iter)
    else:
        return self.done.Done(self.gbest,
                    gpos=self.gpos,
                    pos=self.pos,
                    max_iter=self.max_iter,
                    iteration=self.iterations)
```

First, we ask whether or not an object supporting a `Done` method was given to `RO`. If so, we drop down and call the `Done` method of that object returning whatever boolean value it returns. If `False`, then the search continues; otherwise, the search terminates. Note, the `Done` method is solely responsible for making this decision. The `RO` class will happily loop forever if `Done` consistently returns `False`. The arguments to `Done` reflect the current state of the search. This is to help the code in `Done` decide whether to continue or not.

If no object was given when the RO instance was constructed, we fall back to checking whether or not a tolerance value was set. If not, we return whether we've run out of iterations. If a tolerance value was set, we check for maximum iterations and whether the current best objective function value, gbest[-1], is less than the tolerance value.

The Initialize, Step, and Done methods implement the calls in Optimize. Now, let's develop the code for selecting and evaluating candidate positions.

3.2.5 CandidatePositions

We worked through the candidate position method we're using in Section 3.1. Let's put that into code here. The implementation works with NumPy arrays, so we can select new candidate positions for the entire swarm with a minimum of code,

```
def CandidatePositions(self):
    n = np.random.normal(size=(self.npart, self.ndim))/5.0
    pos = self.pos + self.eta*self.pos*n
    if (self.bounds != None):
        pos = self.bounds.Limits(pos)
    return pos
```

We first set n to an npart by ndim matrix of random values drawn from a normal distribution with zero mean and a standard deviation of one. We divide these values by five to map virtually all of them to the range $[-1, 1]$. This is the offset vector we'll scale by η and use as a fraction of the current position, both positive and negative, to add as an offset. This is the new candidate position.

If we passed a bounds object to the RO instance, we make sure to call its Limits method to give it a chance to clip or resample out of bounds particle positions and to run any custom Validate method the Bounds object may have. We then return the set of candidate positions so we can evaluate them.

3.2.6 Evaluate

When we create the RO instance, we must pass it an objective function object. This object, as we saw in Chapter 2, is problem-specific and initialized outside of the RO class. At that time, any ancillary information needed to evaluate a single particle position must be passed in via the constructor. Here, the RO class Evaluate method calls the objective function object's Evaluate method for each particle,

```
def Evaluate(self, pos):
    p = np.zeros(self.npart)
    for i in range(self.npart):
        p[i] = self.obj.Evaluate(pos[i])
    return p
```

For the RO class, the argument to Evaluate is a set of candidate positions, one for each particle in the swarm. Therefore, we need an npart sized vector to hold the objective function values (p). Then, we loop over each particle and store the objective function value returned by calling obj.Evaluate.

In this book, we do not concern ourselves with performance considerations. However, since the evaluation of the objective function for a single particle position is independent of all other particles, we could at this point parallelize the algorithm and evaluate multiple

positions at one time. This might increase the performance of the search considerably as calling `obj.Evaluate` will happen thousands and thousands of times for a typical search. For our experiments starting in Chapter 8, we'll instrument the objective function object so it tracks how often it is called, and we'll see just how many evaluations are (sometimes) necessary for the algorithm to converge.

We now have all the methods we need to run a random optimization task. The final method is a convenience one to return information about the swarm and the search.

3.2.7 Results

We get information about completed searches by calling `Results`. The definition is,

```
def Results(self):
    if (not self.initialized):
        return None
    return {
        "npart": self.npart,
        "ndim": self.ndim,
        "max_iter": self.max_iter,
        "iterations": self.iterations,
        "tol": self.tol,
        "eta": self.eta,
        "gbest": self.gbest,
        "giter": self.giter,
        "gpos": self.gpos,
        "gidx": self.gidx,
        "pos": self.pos,
        "vpos": self.vpos,
    }
```

The `Results` method packages swarm results and returns them as a dictionary. We used this method in Chapter 2 to demonstrate how to use the framework objects. We'll use this method going forward. We already know what each of the items returned are, so we won't elaborate on them here. Every swarm optimization algorithm we implement has a `Results` method. What is in common between the methods, like `gbest` and `gpos`, will always be in the dictionary. Additionally, any algorithm-specific parameters will be present as well. Therefore, for the `RO` class, we see that η is present.

Our implementation of the `RO` class is complete. Again, the source code is in the file `RO.py`. Let's take the `RO` class for a test drive.

3.3 Testing the RO Class

The example of Chapter 2 was a simple optimization problem for which we had every reason to expect the algorithm to, in time, find a good solution. To test the performance of the `RO` class, and to gain insight into not only how it operates but how all swarm-based algorithms operate, let's pick an example we'll use going forward in subsequent chapters. This lets us compare algorithm performance while testing.

Our test example will be in the same vein as the example of Chapter 2. We'll find the minimum of a 2D function, but one that isn't as simple as $f(x, y) = xy$. Instead, we'll look for the minimum of,

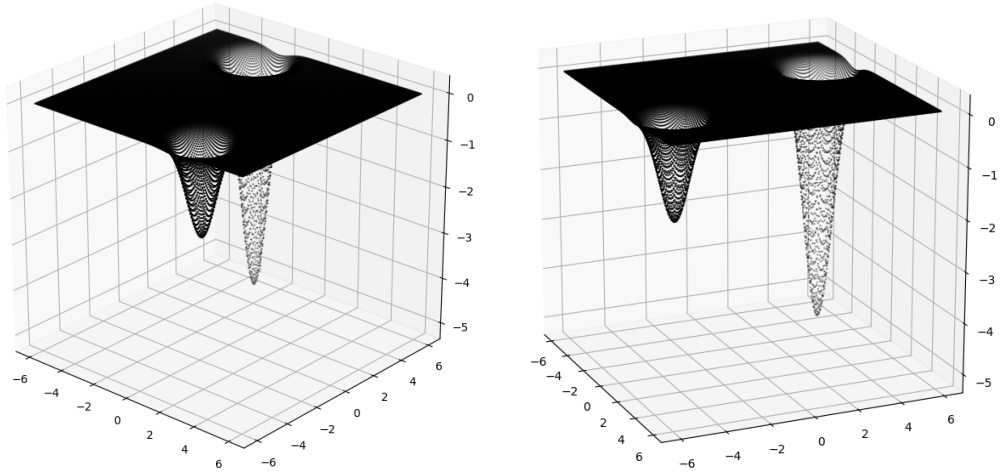


Figure 3.5: Two views of the test function, $f(x, y)$.

$$f(x,y) = -5\exp\left(-\frac{1}{2}\left(\frac{(x+2.2)^2}{0.4} + \frac{(y-4.3)^2}{0.4}\right)\right) + \\ -2\exp\left(-\frac{1}{2}\left(\frac{(x-2.2)^2}{0.4} + \frac{(y+4.3)^2}{0.4}\right)\right) \tag{3.1}$$

which is a pair of two-dimensional Gaussians with minima at $(2.2, -4.3)$ and $(-2.2, 4.3)$, but the minima are not of equal depth so the global minimum is at $(-2.2, 4.3)$. Figure 3.5 shows two views of this function.

The set up for the `RO` class requires initializer and bounds objects, and to define an objective function class. As in Chapter 2, the objective function is simply the value of $f(x, y)$ for a given x, y. We have a two-dimensional problem we'll bound to $-6 \leq x, y \leq 6$.

The code presented below is found in the `fxy_gaussian.py` file of the source code distribution. Note, the `fxy_gaussian.py` file is configured to use all the swarm algorithms we'll develop. The listing in this chapter is specific to the `RO` class only.

First, let's import our modules, define the objective function class, make an instance of it, and the `Bounds` object,

```
from RO import *
from Bounds import *
from RandomInitializer import *
from QuasirandomInitializer import *
from SphereInitializer import *

class Objective:
    def Evaluate(self, p):
        return -5.0*np.exp(-0.5*((p[0]+2.2)**2/0.4+(p[1]-4.3)**2/0.4)) +
               -2.0*np.exp(-0.5*((p[0]-2.2)**2/0.4+(p[1]+4.3)**2/0.4))

obj = Objective()
b = Bounds([-6,-6], [6,6], enforce="resample")
```

Notice, we also import the three initializer classes from Chapter 2. The objective function object is put in `obj` and the class is a direct implementation of the equation for

$f(x, y)$ with `p` standing in for x and y. The `Bounds` object is restricted to $[-6, 6]$ and uses resampling for out of bounds dimensions.

We still need to create an initializer object. We'll show the random case here, but trust you'll experiment with the quasirandom and sphere initializers as well. All we need is a single line,

```
i = RandomInitializer(npart=npart, ndim=2, bounds=b)
```

where we pass in the `Bounds` object and set the number of dimensions to two. We assume `npart` is set to the number of particles we want in the swarm. Below, we'll experiment with the number of particles and the maximum number of iterations of the swarm, which we'll call `miter`.

Let's create our random optimization swarm object and do the search,

```
swarm = RO(obj=obj, npart=npart, ndim=2, max_iter=miter, init=i, bounds=b)
swarm.Optimize()
```

Now, let's get the results and see how we did,

```
res = swarm.Results()
x,y = res["gpos"][-1]
v = res["gbest"][-1]
print("f(%0.8f, %0.8f) = %0.10f" % (x,y,v))
print("(%d swarm best updates)" % (len(res["gbest"]),))
```

Notice, we call `Results` to get the dictionary of results and then extract the best position, `x` and `y`, along with the function value at that position, `v`. The length of `gbest` gives us the number of swarm updates.

Let's run the search using `npart=10` and `miter=100`. Each time we run, we'll get a different output, but one run returned,

```
f(-2.24570451, 3.82387180) = -3.7563742785
(19 swarm best updates)
```

To use the `fxy_gaussian.py` file to run this search, use a command line like,

```
> python3 fxy_gaussian.py 10 100 RO RI
```

We know the global minimum of $f(x, y)$ is at $(-2.2, 4.3)$ and has a value of -5. Our search above was heading in the right direction, but it wasn't able to find a good approximation of the minimum.

If we repeat the search ten times, we'll get some statistics on how well we do on average. Of course, when you run the code ten times, you'll get ten different results. My run showed that for eight of the ten runs, the swarm selected an endpoint near the global minimum of $(-2.2, 4.3)$. However, twice it was moving in the wrong direction, towards $(2.2, -4.3)$. The mean and standard error of the minimum found, and the number of swarm updates needed was,

Minimum found	-3.7851 ± 0.8534
Swarm updates	38.9 ± 6.2

Recall, this is for a swarm of ten particles and 100 iterations.

We have a small swarm, only ten particles. Let's increase the number of iterations by a factor of ten and see if letting the swarm explore more helps. So, change `miter=100` to `miter=1000`. Ten runs gives us a new set of means,

Minimum found	-4.9999777 ± 0.0000098
Swarm updates	49.0 ± 11.8

which is significantly better. For this run, the swarm headed for the proper global minimum in all cases. A small swarm with many iterations was able to do a good job, on average. What if we use a larger swarm but return to using `miter=100`? Let's move from `npart=10` to `npart=100`. In that case, we get,

Minimum found	-4.9997805 ± 0.0001287
Swarm updates	21.4 ± 2.9

where the results are nearly as good as for ten times as many iterations of a swarm with only one-tenth the number of particles. Note, also, that the number of swarm updates is lower, on average, than for the case with only ten particles. This observation makes sense: a smaller swarm is more likely to need to search around to find good places, and that leads to a large number of swarm updates.

All of the above validates our intuition that small swarms that get to explore are good, as are larger swarms with fewer iterations. So, shouldn't we always use large swarms, then? Not really. If the objective function is computationally expensive, and we have good reason to believe our bounds are tight, meaning the solution is likely within them, then we might opt for a smaller swarm to save on doing too many calls to the objective function. If we run the search to `miter` iterations every time, the number of calls to the objective function is `npart`×`miter` plus another `npart` calls to initialize the swarm.

Also, in the example above, the swarm of ten particles run for 1000 iterations, 10,000 objective function calls, was, on average, better at finding the global minimum than the swarm of 100 particles run for 100 iterations, also 10,000 objective function calls. We'll do similar analyses for the other swarm algorithms we develop.

If we don't call `Optimize` of `RO`, but instead manually call `Initialize` followed by `miter` calls to `Step`, we can interrogate the swarm during the search to see where it is in the search space. This opens up the possibility of creating a movie of the search since we are in two dimensions. Naturally, we can't show such a movie in a book, but we can show frames from it. Building the movie and showing specific frames from it gives us Figure 3.6.

Figure 3.6 shows the swarm positions at different points in the search. Moving clockwise from the upper left, we see the initial swarm configuration. The particle positions are circles, the best position of the swarm is a star, and the known global minimum of $f(x, y)$ is shown as a square. Note, for this figure, and similar ones in subsequent chapters, we're fixing the NumPy pseudorandom number seed so each swarm initializes in exactly the same way. We set the NumPy seed by adding,

```
np.random.seed(8675309)
```

before the Bounds object is created.

Fixing the seed means we can compare how the swarms evolve from the same starting point. Notice that the initial swarm best position is actually quite close to the second minima of the function at $(2.2, -4.3)$ yet the swarm evolves to find the true global minima at the end of the search.

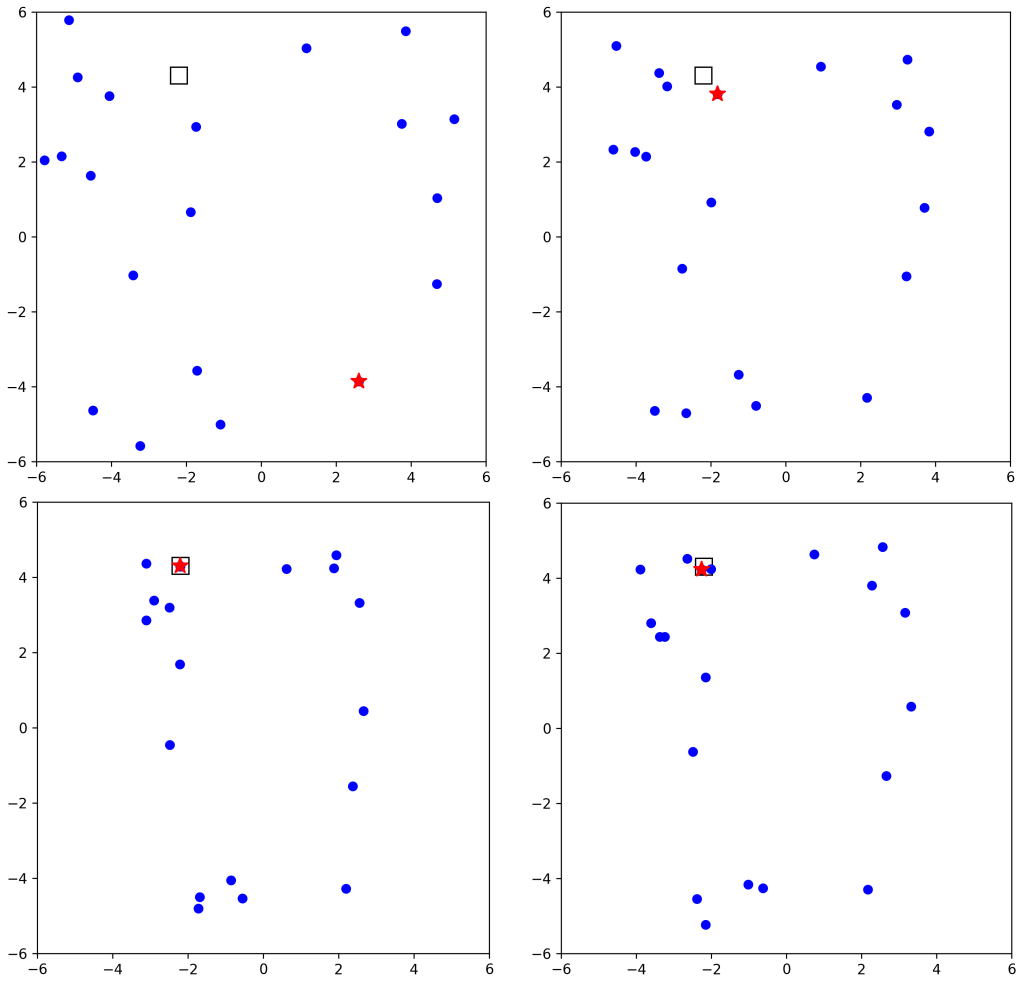


Figure 3.6: Frames from a search with a swarm of twenty particles and 100 iterations. Clockwise from the upper left: initial swarm positions, frame 33, frame 66, and final swarm positions. Swarm particles are circles. The swarm best position is a star and the known global best position is a square.

Next in Figure 3.6, we see frame 33, after 33 iterations of the swarm. The best particle has now moved quite close to the global minimum. The next frame is frame 66. We see that the swarm best is virtually on top of the global minimum. We also see that other particles near the global minimum have moved in that direction. The final frame shows the configuration of the swarm at the end of the search. The global best position has been found with reasonable quality, and we see that other nearby particles are also closing in on the minimum. Given the plot of Figure 3.5, the particles near the minimum are "falling into" the hole.

Random optimization, as mentioned at the beginning of this chapter, is individualistic. There is no communication between members of the swarm. We see this in Figure 3.6. The swarm best particle has moved to the global minimum, but the majority of the swarm is still wandering about in the search space. Those particles are unaware of how well the swarm best particle is doing. We'll repeat this sort of analysis when we test the other swarm algorithms in later chapters. In some of those cases, we'll see a different kind of behavior

from the swarm precisely because the particles share information about their current state.

This concludes our implementation and testing of the `RO` class. Now that we know how a specific swarm algorithm works and interfaces with the framework created in Chapter 2, we can move on to our next swarm intelligence algorithm: particle swarm optimization.

Chapter 4

Particle Swarm Optimization

Chapter 3 started us off with the simplest of swarm optimization algorithms, random optimization. In this chapter, we encounter what is perhaps the most widely used swarm optimization technique, *particle swarm optimization* or PSO.

Section 4.1 describes the PSO algorithm, including variations on the main theme that we'll develop as part of our PSO class. The PSO class itself is defined, method by method, in Section 4.2. We put the PSO class through its paces in Section 4.3 and compare it to random optimization.

4.1 Making Sense of the World

James Kennedy, co-creator of the PSO algorithm, wrote that the main insight leading to PSO was: "People learn to make sense of the world by talking with other people about it." [12].

The random optimization algorithm of Chapter 3 implements a swarm of particles moving independently through the search space. Particle i never shares information with particle j. A supreme overseer watches the independent agents and notes the best solution position any one of them found.

In a particle swarm, the particles *do* share information with each other. They do learn about the world (search space) by talking with other people (particles) about it. We'll outline the canonical algorithm in a bit, but, in essence, in PSO a particle knows two things: the best place it's found in the search space, and the best place the swarm as a whole, or a subset of it (a neighborhood), has found. The first piece of knowledge is *cognitive*, something the particle knows without being told. The second is *social*, something the particle knows because other particles told it. The tension between these two types of knowledge guides particles as they search. The (hopeful) net result is the swarm converging on the best solution to the problem we are trying to solve.

The number of PSO variants developed over the last several decades is legion. We'll content ourselves with only two: canonical PSO [13] and bare bones PSO [14]. Canonical PSO is outlined in Algorithm 6. We'll see below how to modify it for bare bones PSO.

4.1.1 Canonical PSO

Let's contemplate Algorithm 6. The inputs include the now familiar objective function (Section 2.2), a bounds object (Section 2.3), an initialization object (Section 2.4), and an

Algorithm 6 The canonical PSO algorithm.

Input: An objective function, bounds, initialization type, c_1, c_2, ω_0, inertia schedule
Output: The best position found by the swarm

```
for each particle, i do
    Select an initial position within the bounds of the search space, x_i
    Evaluate the objective function at this position
    Mark this position as the best found by the particle so far, x̂_i ← x_i
end for
Store the best initial particle position as the swarm global best position, g
Set the initial particle velocities to zero, v_i ← 0
while not done do
    for each particle, i do
        Set ω for this iteration of the swarm
        Get the neighborhood best position for particle i, ĝ_i
        Update the velocity: v_i ← ωv_i + c_1(x̂_i − x_i) + c_2(ĝ_i − x_i)
        Update the position: x_i ← x_i + v_i
        Evaluate the new position, x_i
        if fitness of new position, x_i < fitness of particle best position, x̂_i then
            Update the particle best position, x̂_i ← x_i
        end if
        if fitness of new position < fitness of swarm best position then
            Store the new global best, g ← x_i
        end if
    end for
    Increment the iteration counter
end while
```

inertia object (Section 2.6). In addition, we need three scalar parameter values: c_1, c_2, and ω_0. If we include an inertia object as input, ω_0 becomes optional.

A read through Algorithm 6 reveals references to several types of vectors. First, we have $\boldsymbol{x}_i$. This is the position of particle i. The position is a d-dimensional vector where d is the dimensionality of the search space. All the vectors in Algorithm 6 are d-dimensional. Associated with each particle are two other vectors. The first is the particle's velocity, $\boldsymbol{v}_i$. The velocity is used to update the particle's position for the next iteration of the swarm. The second vector associated with each particle is $\hat{\boldsymbol{x}}_i$. This vector is the best position in the search space that particle i has found on its own. The $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ vectors are randomly generated on each iteration: $\boldsymbol{c}_1 = c_1\boldsymbol{U}_1$ and $\boldsymbol{c}_2 = c_2\boldsymbol{U}_2$ with $\boldsymbol{U}_1$ and $\boldsymbol{U}_2$ uniform d-dimensional random vectors, $[0, 1)$.

There are two additional vectors called out in Algorithm 6. One we've encountered before, though not as a mathematical symbol. It's $\boldsymbol{g}$. This is the global best position, the value we want to return when the search completes. The concept of a neighborhood of particles, which we'll make more concrete momentarily, gives us $\hat{\boldsymbol{g}}_i$. This vector represents the best position the neighborhood of particle i knows about.

A neighborhood is a set of particles, and each particle belongs to at least one neighborhood. In its simplest form, the entire swarm forms a single neighborhood. In that case, $\hat{\boldsymbol{g}}_i = \boldsymbol{g}$. The idea of neighborhoods is to partition the swarm into sets of particles that can influence each other. The arrangement of particles into neighborhoods is the topology of

the swarm. Researchers have investigated many different types of topologies. We'll implement two in our `PSO` class. The first is global, there is only one neighborhood, and every particle belongs to it. The second is a ring. If the swarm has n particles, we can imagine they are arranged in a ring, so that particle i is linked to particles $i-1$ and $i+1$, with suitable wrapping so that particle $n-1$ is followed by particle zero. With this topology and a given number of neighbors on either side of it, we have a neighborhood for each particle. For example, if working with particle 5 and the number of neighbors on each side of it is two, then particles 3 through 7 form the neighborhood of particle 5 and $\hat{\boldsymbol{g}}_5$ will be the best position known by particles 3 through 7. Notice that particle 5 is part of more than one neighborhood. This is often the case with swarm topologies. Continuing our example, all of the following ring neighborhoods include particle 5:

$$(1,2,3,4,5);(2,3,4,5,6);(3,4,5,6,7);(4,5,6,7,8);(5,6,7,8,9)$$

The people analogy for PSO helps here. As people, we typically belong to more than one group. We have our families, our coworkers, our local community, and many specialized communities related to our beliefs, politics, hobbies, sports, etc. Therefore, when Algorithm 6 refers to a neighborhood, it is referring to an arrangement of the particles, which, for us, means either all particles in one neighborhood or a ring of neighborhoods. We'll see in Section 4.2 exactly how the ring neighborhoods are configured.

The first `for` loop in Algorithm 6 gets the set of initial swarm positions, $\boldsymbol{x}$. It then evaluates the objective function at each of these positions. Since the swarm has only looked at the initial set of positions, we set $\hat{\boldsymbol{x}}_i$ to $\boldsymbol{x}_i$ as that is the best position each particle knows about so far. The best $\hat{\boldsymbol{x}}_i$ position is used to initialize the swarm best position, $\boldsymbol{g}$. While not called out explicitly in Algorithm 6, we also need to store the objective function value at each particle best position and the global best position.

The `while` loop runs the search. The loop continues until a maximum number of swarm updates has been performed, a tolerance on the objective function value has been met, or a `Done` object has returned true (Section 2.5).

The inner `for` loop represents a single swarm update. The first step is to set ω for the current iteration. If we peek ahead a bit, we see that ω is used in the velocity update equation as a coefficient on the current velocity of particle i. Section 2.6 told us that ω is a number in the range $[0,1]$. Typically, it is never below 0.5. We see now why it is called inertia. It reduces the velocity of the particle on the next swarm iteration. As the search progresses, it is beneficial to make ω smaller under the belief that the swarm is closer to the best position in the search space and that we don't want to move too far on the next iteration. We want to slow down, so we keep less of the previous iteration's velocity. This is precisely what the `LinearInertia` class of Section 2.6 does; it reduces ω linearly as the number of swarm iterations increases. Typical values start ω at 0.9 and when the maximum number of iterations is complete ω has decreased to 0.5. After setting ω, we get $\hat{\boldsymbol{g}}_i$ for the neighborhood of particle i.

The next two steps in Algorithm 6 are the core of canonial PSO. The first applies the velocity update equation to decide what the velocity of particle i is for this iteration,

$$\boldsymbol{v}_i \leftarrow \omega\boldsymbol{v}_i + \boldsymbol{c}_1(\hat{\boldsymbol{x}}_i - \boldsymbol{x}_i) + \boldsymbol{c}_2(\hat{\boldsymbol{g}}_i - \boldsymbol{x}_i) \tag{4.1}$$

There are three terms in Equation 4.1. The first we already mentioned, it multiplies the existing velocity vector for particle i by $\omega < 1$ to retain a fraction of the previous velocity.

Intuitively, $\omega \boldsymbol{v}_i$ makes sense. Without it, the velocity might explode in magnitude.

The next two terms in Equation 4.1 are the cognitive and social terms alluded to above. The cognitive component,

$$\boldsymbol{c}_1(\hat{\boldsymbol{x}}_i - \boldsymbol{x}_i)$$

employs the knowledge particle i has gained on its own. The best position particle i has found during the search is in $\hat{\boldsymbol{x}}_i$. The vector difference between this particle best position and the particle's current position is multiplied, component by component, by $\boldsymbol{c}_1$, a random vector with components in the range $[0, c_1)$.

The social component,

$$\boldsymbol{c}_2(\hat{\boldsymbol{g}}_i - \boldsymbol{x}_i)$$

is influenced by the neighborhood best position, $\hat{\boldsymbol{g}}_i$. It is also multiplied by a random vector in the range $[0, c_2)$. The sum of these three components becomes the vector used to update the position of particle i,

$$\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i + \boldsymbol{v}_i \tag{4.2}$$

where, if you are bothered by the straight addition of a velocity and a position, you can imagine the velocity vector to be multiplied by $\Delta t = 1$ as the time step between swarm updates.

The newly updated particle position, $\boldsymbol{x}_i$, is passed to the objective function for evaluation. If the fitness of $\boldsymbol{x}_i$ is less than the fitness of the particle best position, $\hat{\boldsymbol{x}}_i$; the particle best position is updated. If the fitness of $\boldsymbol{x}_i$ is also less than the fitness of $\boldsymbol{g}$, the overall swarm best position, it is also updated.

The swarm update step completes when all particles have been moved to a new position, and those new positions have been evaluated. At this point, the iteration counter is incremented, and the next swarm update loop begins. When done, the position in $\boldsymbol{g}$ is returned as the solution to the search.

Algorithm 6 does not expressly mention two features of many PSO implementations: the bounding of $\boldsymbol{x}_i$ and the bounding of $\boldsymbol{v}_i$. The former establishes the search space. A decision must be made when a particle attempts to move beyond the bounds of the search space. Perhaps the simplest is to take any component of $\boldsymbol{x}_i$ outside the allowed range and clip it to the limit. For example, if $\boldsymbol{x}_{ij}$, the j-th position component of particle i, is updated to 17, and the limit for that component is 15, then change the 17 to a 15 and continue. In our framework, we use `enforce="clip"` when instantiating a `Bounds` object in this case. Another option is to replace the offending component value with one randomly selected from the allowed range. For the framework, `enforce="resample"` causes this to happen.

The velocity may be limited in the same way as the position, component by component. Typically, this is not required because of the first term in Equation 4.1, but if velocities are increasing dramatically, clipping or resampling should be used. The framework uses a `Bounds` object to control the range of allowed velocities just as it does for positions.

Let's take a look at our second PSO variant, bare bones PSO (BBPSO), to see how it differs from the canonical algorithm.

4.1.2 Bare Bones PSO

In 2003, Kennedy outlined a new variant of PSO, which he termed *Bare bones PSO* [14]. Bare bones PSO does away with the concept of velocity, Equation 4.1, and changes the update rule, Equation 4.2, to a stochastic one that randomly updates particle position components with either the corresponding component of the particle best position or a new component value selected from a normal distribution with a mean between the particle best component and the neighborhood best component. [14] refers to the probability threshold deciding which one of these two component updates is used as the *interaction probability*, p_b.

Mathematically, then, $\boldsymbol{x}_{ij}$ is set during the update loop to,

$$\begin{aligned} \bar{x} &= \frac{1}{2}(\hat{\boldsymbol{g}}_{ij} + \hat{\boldsymbol{x}}_{ij}) \\ \sigma &= |\hat{\boldsymbol{g}}_{ij} - \hat{\boldsymbol{x}}_{ij}| \\ \boldsymbol{x}_{ij} &\sim \mathcal{N}(\bar{x}, \sigma) \end{aligned} \tag{4.3}$$

if $p \sim U[0, 1) < p_b$. Otherwise,

$$\boldsymbol{x}_{ij} \leftarrow \hat{\boldsymbol{x}}_{ij} \tag{4.4}$$

copies the corresponding component of the particle best position.

Note that $\bar{x}$ is the mean value between the current component (j) of the neighborhood best position and the particle best position. The variance of the normal distribution is set to the absolute value of the difference in these positions. Doing this sets the position component to some random value near the mean of the best the particle knows and the best the neighborhood knows. As the swarm learns and, as often happens, contracts as particles move towards $\boldsymbol{g}$, the normal distribution becomes tighter and tighter as σ gets smaller and smaller. This is an entirely reasonable thing for the swarm to do as it nears the global best position.

We've described canonical PSO, and bare bones PSO, but we haven't paid much attention to how to configure a PSO swarm for a search. Let's remedy that next.

4.1.3 Configuring a Particle Swarm

To configure any swarm search, we need an objective function, the dimensionality of the search space, bounds, initialization, the number of particles, and what it means to be done (tolerance value or a maximum number of iterations). For a particle swarm, we need to set other values as well, like c_1 and c_2. If we're using a constant ω, we need to set ω_0; otherwise, we need to configure an inertia object.

Typically, we set c_1 and c_2 before running the search and then leave them as constants. However, as with anything in this field that can be adjusted, someone has looked at the effect of adjusting these values dynamically. Let's review what these values represent and then consider their implications. For the following, we are thinking only of the canonical algorithm; bare bones PSO doesn't use c_1 or c_2.

We use c_1 to weight the cognitive component of the velocity update. This reflects local knowledge, the particle's discoveries. So, if we make $c_1 > c_2$, we'll be telling the particle to spend more time searching around what it knows to be a good place and pay less attention to what the remainder of the neighborhood or swarm is telling it. Likewise, making $c_2 > c_1$

```
class PSO:
    def __init__(self, ...):
    def Results(self):
    def Initialize(self):
    def Done(self):
    def Evaluate(self, pos):
    def RingNeighborhood(self, n):
    def NeighborhoodBest(self, n):
    def BareBonesUpdate(self):
    def Step(self):
    def Optimize(self):
```

Figure 4.1: Skeleton of the PSO class.

does the opposite; the particle thinks less for itself and instead goes along with the crowd (neighborhood). We'll experiment with these options below.

Historically, it was suggested to use $c_1 = c_2 = 2.0$, but careful analysis in [15] led to a recommendation of $c_1 = c_2 = 1.49$, the default values the `PSO` class uses. [16] showed that particle trajectories converge if,

$$\omega > \frac{1}{2}(c_1 + c_2) - 1$$

which our default c_1, c_2, and ω values satisfy.

What about ω? Again, a myriad of experiments have been performed and published. The notion of linearly decreasing ω over the life of the search from about 0.9 to about 0.5 seems, operationally, to be a reasonable thing to do. These are the default values for a framework `LinearInertia` object, so, in our experiments, we'll often use the default constructor in that case. For variety, the framework also includes `RandomInertia`, which uses a randomly selected inertia value on each iteration, but the range is restricted to $[0.5, 1)$. When we test the `PSO` class in Section 4.3, we'll experiment a bit with c_1 and c_2, but we'll leave experimenting with ω as an exercise for the reader.

Let's make PSO concrete. The next section walks through the code for the `PSO` class. Compare the implementation in Section 4.2 with Algorithm 6.

4.2 The PSO Class

Like the `RO` class of Section 3.2, the `PSO` class uses the framework objects of Chapter 2. We'll present a skeleton of the class, the class constructor, and then fill in the skeleton, method by method. The source code for the `PSO` class is in the file `PSO.py`. As before, the listing here is condensed, see `PSO.py` for comments and proper spacing.

The `PSO` class skeleton is in Figure 4.1. The overlap in method names with the other optimization classes, like `RO` in Figure 3.4, is clear. However, while some methods are identical from algorithm to algorithm, others only share a name, the underlying code is different.

The constructor (`__init__`) takes up to 17 possible arguments. The `PSO` class is the most sophisticated class we'll encounter. Much of this is because it implements multiple variants of the PSO algorithm. The possible arguments are in Table 4.1.

Parameter	*Description*
`obj`	Objective function object (required)
`npart`	Number of particles in the swarm (`10`)
`ndim`	Number of dimensions in the search space (`3`)
`max_iter`	Maximum number of swarm iterations (`200`)
`c1`	Cognitive parameter (`1.49`)
`c2`	Social parameter (`1.49`)
`w`	Base inertia parameter value (ω_0 = `0.729`)
`inertia`	Inertia object (`None` means use ω_0)
`bare`	If true, use the bare-bones update rule (`False`)
`bare_prob`	Probability of updating a particle's components (bare-bones only, `0.5`)
`tol`	Tolerance value (`None`)
`init`	Initializer object (`None`)
`done`	Done object (`None`)
`ring`	If true, use a ring topology (`False`)
`neighbors`	Number of particle neighbors for the ring, must be even (`2`)
`vbounds`	Velocity bounds object (`None`)
`bounds`	Bounds object (`None`)

Table 4.1: Arguments to the PSO class constructor.

Many of these arguments are universal to the swarm algorithms we'll develop. The arguments new to the `PSO` class are,

Parameter	*Description*
`c1`	Cognitive parameter
`c2`	Social parameter
`w`	Base inertia parameter value (ω_0)
`inertia`	Inertia object (`None` means use ω_0)
`bare`	If true, use the bare-bones update rule
`bare_prob`	Probability of updating a particle's components (bare-bones only)
`ring`	If true, use a ring topology
`neighbors`	Number of particle neighbors for the ring, must be even
`vbounds`	Velocity bounds object

We see that `c1`, `c2`, and `w` relate directly to c_1, c_2, and ω from the velocity update equation (Equation 4.1). If not supplied, these values default to 1.49, 1.49, and 0.729, respectively. These are the recommended values in [15]. In most cases, you'll want to adjust ω as the swarm evolves from iteration to iteration. The framework supports this via the `inertia` parameter accepting an instance of an inertia class like `LinearInertia` (see Section 2.6).

Set `bare` to `True` to use the bare bones update rule instead of canonical PSO. If desired, one can adjust `bare_prob` to a value in $[0, 1]$ to control the probability of selecting a new value for the current particle's components or using the same component from the best position the current particle knows.

The `PSO` class supports two swarm topologies: global and ring. To select a ring topology, set `ring` to `True` and make `neighbors` an even number to set the number of neighbors, half to the left and half to the right of the current particle. The default is two.

At times, it is desirable to bound the particle velocities. To do this with the `PSO` class, pass a properly initialized `Bounds` object, or subclass, to the `vbounds` parameter. If set, the particle velocities, after the update, are bounded by calling the `Limits` method of the bounds object.

The code for the constructor is quite similar to all the other swarm classes,

```
def __init__(self, ...):
    self.obj = obj
    self.npart = npart
    self.ndim = ndim
    self.max_iter = max_iter
    self.init = init
    self.done = done
    self.vbounds = vbounds
    self.bounds = bounds
    self.tol = tol
    self.c1 = c1
    self.c2 = c2
    self.w = w
    self.bare = bare
    self.bare_prob = bare_prob
    self.inertia = inertia
    self.ring = ring
    self.neighbors = neighbors
    self.initialized = False
    if (ring) and (neighbors > npart):
        self.neighbors = npart
```

where arguments are stored in member variables. If the ring topology is selected and the number of neighbors happens to be higher than the number of particles in the swarm, the neighborhood is set to the number of particles.

As in Chapter 3, we present the remainder of the `PSO` class code in a top-down fashion.

4.2.1 Optimize

All the swarm classes use the same form for this method,

```
def Optimize(self):
    self.Initialize()
    while (not self.Done()):
        self.Step()
    return self.gbest[-1], self.gpos[-1]
```

which follows Algorithm 2 by initializing the swarm (`Initialize`) before entering the `while` loop to do swarm updates and evaluations (`Step`) until done (`Done`). As before, the swarm best objective function value and position are returned.

Let's look at how PSO initializes the swarm.

4.2.2 Initialize

The code for `Initialize` is,

```
def Initialize(self):
    self.initialized = True
    self.iterations = 0
    self.pos = self.init.InitializeSwarm()
```

```
    self.vel = np.zeros((self.npart, self.ndim))
    self.xpos = self.pos.copy()
    self.xbest= self.Evaluate(self.pos)
    self.gidx = []
    self.gbest = []
    self.gpos = []
    self.giter = []
    self.gidx.append(np.argmin(self.xbest))
    self.gbest.append(self.xbest[self.gidx[-1]])
    self.gpos.append(self.xpos[self.gidx[-1]].copy())
    self.giter.append(0)
```

The swarm is marked as initialized (`initialized`), and the iteration counter set to zero (`iterations`). Next, the `InitializeSwarm` method of the initializer object sets the initial positions of the particles (`pos`), see Section 2.4. The initial particle velocities are set to zero (`vel`).

The initial position of each particle is, by default, the best-known position for that particle, so the positions are simply copied to the per particle best positions, `xpos`. As with `pos`, `xpos` is a 2D NumPy array where each row is that particle's best-known position. The particle best objective function values are stored in `xbest` by evaluating the objective function at the initial positions.

The remainder of the code sets up tracking of the swarm best position as described in Section 3.2.2. Each swarm algorithm will track the evolution of the swarm in this way.

4.2.3 Step

The `Step` method implements a single swarm evaluation and update step. The code is,

```
    def Step(self):
        if (self.inertia != None):
 1:         w = self.inertia.CalculateW(self.w,
                                        self.iterations, self.max_iter)
        else:
 2:         w = self.w
        if (self.bare):
 3:         self.pos = self.BareBonesUpdate()
        else:
            for i in range(self.npart):
 4:             lbest, lpos = self.NeighborhoodBest(i)
                c1 = self.c1 * np.random.random(self.ndim)
                c2 = self.c2 * np.random.random(self.ndim)
 5:             self.vel[i] = w*self.vel[i] +                    \
                              c1*(self.xpos[i] - self.pos[i]) +  \
                              c2*(lpos - self.pos[i])
            if (self.vbounds != None):
 6:             self.vel = self.vbounds.Limits(self.vel)
 7:         self.pos = self.pos + self.vel
        if (self.bounds != None):
 8:         self.pos = self.bounds.Limits(self.pos)
 9:     p = self.Evaluate(self.pos)
        for i in range(self.npart):
10:         if (p[i] < self.xbest[i]):
                self.xbest[i] = p[i]
                self.xpos[i] = self.pos[i]
11:         if (p[i] < self.gbest[-1]):
                self.gbest.append(p[i])
```

```
                self.gpos.append(self.pos[i].copy())
                self.gidx.append(i)
                self.giter.append(self.iterations)
12:     self.iterations += 1
```

A swarm iteration step starts by setting the value of ω. If an inertia object is been supplied, its `CalculateW` method is called `(1)`. The arguments are ω_0 (`w`), the current swarm iteration number (`iterations`) and the maximum number of iterations to perform (`max_iter`), ignoring any early termination. We saw the code for the `LinearInertia` and `RandomInertia` classes in Section 2.6. If no inertia object was passed to the `PSO` class, ω_0 is used for each swarm update `(2)`. If using bare bones PSO, the `BareBonesUpdate` method is called `(3)`. We'll see the definition in Section 4.2.7. Otherwise, the canonical swarm update is used.

The canonical update examines each particle in the swarm. First, we locate the local best objective function value (`lbest`) and position (`lpos`) for the neighborhood in which the current particle (`pos[i]`) resides by calling `NeighborhoodBest (4)`. We'll review the code for `NeighborhoodBest` in Section 4.2.6 below. After this, we generate `c1` and `c2` to use for this particle as some random fraction of c_1 and c_2, respectively. Note that `c1` and `c2` are vectors to use different values for each dimension of the particle.

With the preliminaries out of the way, the velocity update for particle `i` is found by implementing Equation 4.1. Note, after the update, we check to see if a Bounds object was given and if so, we call the `Limits` method `(6)`. The velocity vector is used to move the particle to its new position `(7)`.

Once all particles move to a new position, any boundary conditions are applied `(8)` and the new particle positions are evaluated with a call to `Evaluate (9)` (see Section 4.2.8) to return a vector of objective function values (`p`). Each particle is then checked to see if it is either a new particle best position `(10)` or a new swarm best postion `(11)` with appropriate bookkeeping to track the evolution of the swarm. Finally, the swarm update step completes, so the iteration counter is bumped `(12)`.

4.2.4 Done

This method is identical to the `Done` method of the `RO` class. See the code in Section 3.2.4.

4.2.5 NeighborhoodBest

To update the swarm in the `Step` method, we need to know the best location for the neighborhood of the current particle. This is true for both canonical and bare bones variants of PSO. This objective function value and position are returned by `NeighborhoodBest` using the current particle number as the argument. Let's take a look at the code for this method,

```
def NeighborhoodBest(self, n):
    if (not self.ring):
        return self.gbest[-1], self.gpos[-1]
    lbest = 1e9
    for i in self.RingNeighborhood(n):
        if (self.xbest[i] < lbest):
            lbest = self.xbest[i]
            lpos = self.xpos[i]
    return lbest, lpos
```

If the search is not using a ring neighborhood, then the best objective value and position currently known to the swarm are returned. These are stored in the lists `gbest` and `gpos`, respectively. Therefore, the current bests are the last elements in the list (indexed with `[-1]`).

On the other hand, if the PSO object is using a ring neighborhood, we loop over the set of particle indices returned by `RingNeighborhood` (see Section 4.2.6) to find which of them is the best. Notice, the search is not looking for the best position of the neighborhood particles, given where they currently are, but rather the best position that the neighborhood particles have ever found. This is why we look for the smallest `xbest` for the neighborhood particles. When found, the neighborhood best objective value (`lbest`) and position (`lpos`) are returned.

4.2.6 RingNeighborhood

The `RingNeighborhood` method takes the index of a particle (`n`, a row of `pos`) and returns a list of indices representing the set of particles considered to be neighbors of `n`. As we're implementing a ring topology, this means the set of indices that are $m/2$ before and $m/2$ after the given particle number `n` where m is the size of the neighborhood passed to the PSO object when it was constructed.

Before walking through the code for this method, let's see it in action by exercising `RingNeighborhood` directly at the Python command prompt,

```
>>> from PSO import *
>>> p = PSO(obj=None, ring=True, npart=20, neighbors=4)
>>> p.RingNeighborhood(10)
array([ 8,  9, 10, 11, 12])
>>> p.RingNeighborhood(18)
array([16, 17, 18, 19,  0])
>>> p.RingNeighborhood(0)
array([18, 19,  0,  1,  2])
```

The commands above load the `PSO` class and create an instance, `p`. As we are interested in calling the `RingNeighborhood` method directly we create the instance by passing `None` as the objective function, setting `ring` to `True`, setting the number of particles in the swarm to twenty, and the number of neighbors to four.

We now call the `RingNeighborhood` method directly passing in the desired particle number. First, we ask for the indices of the neighborhood of particle 10 and are told that the neighborhood contains particles 8, 9, 10, 11, and 12. There are four neighbors to particle 10, two on each side of it. This follows the ring topology as we defined it in Section 4.1. Next, we want to know the neighborhood of particle 18. This time, the method returns a neighborhood of 16, 17, 18, 19, and 0. The neighborhood of particle 0 is particles 18, 19, 0, 1, and 2. At this point, the pattern is clear: the neighborhood of particle n is particles $n-2$, $n-1$, n, $n+1$, and $n+2$ modulo the number of particles in the swarm.

Putting this observation into code gives us `RingNeighborhood`,

```
def RingNeighborhood(self, n):
    idx = np.array(range(n-self.neighbors//2,n+self.neighbors//2+1))
    i = np.where(idx >= self.npart)
    if (len(i) != 0):
        idx[i] = idx[i] % self.npart
```

```
    i = np.where(idx < 0)
    if (len(i) != 0):
        idx[i] = self.npart + idx[i]
    return idx
```

We create the NumPy array, `idx`, as consecutive indices from $n - m/2$ to $n + m/2$ where n is `n` and m is the size of the number of neighbors (`neighbors`). The values in `idx` may be negative or greater than the number of particles in the swarm (`npart`), so we first ask if any values in `idx` are too large and if so, we replace them with the value modulo `npart`. This wraps neighborhood numbers equal to or above `npart` to the lower end of the range. Then, we check for negative values in `idx`, and if there are any, add `npart` to them to wrap around the other way. Finally, the updated set of neighborhood indices is returned. Note that `n` is always the value in the middle of `idx`. Let's now consider the bare bones swarm update.

4.2.7 BareBonesUpdate

If `bare` is `True`, the `Step` method will call `BareBonesUpdate` to select new swarm positions. The code is,

```
def BareBonesUpdate(self):
    pos = np.zeros((self.npart, self.ndim))
    for i in range(self.npart):
        lbest, lpos = self.NeighborhoodBest(i)
        for j in range(self.ndim):
            if (np.random.random() < self.bare_prob):
                m = 0.5*(lpos[j] + self.xpos[i,j])
                s = np.abs(lpos[j] - self.xpos[i,j])
                pos[i,j] = np.random.normal(m,s)
            else:
                pos[i,j] = self.xpos[i,j]
    return pos
```

The method needs to return an entirely new swarm array (`pos`), so storage for it is created. Each particle is then visited. The bare bones update also uses neighborhoods, so `NeighborhoodBest` is called passing in the current particle number to get the position of the neighborhood best.

Next, each dimension of the current particle is updated. If a random scalar is below `bare_prob`, the update follows Equation 4.3, otherwise it's Equation 4.4 to copy the j-th component of the i-th particle's best-known position. When the loops over particles and dimensions finish, the new swarm positions are returned.

4.2.8 Evaluate

This method is identical to the `Evaluate` method of the `RO` class. See the code in Section 3.2.6.

4.2.9 Results

As for `RO`, the `Results` method returns information related to how the swarm evolved. For the `PSO` class, `Results` returns all the information the `RO` version does, except `eta` and `vpos`, which do not apply to PSO. See Section 3.2.7. Additionally, `Results` returns

PSO-specific values `c1`, `c2`, `w` (ω_0), and `vel`, the velocity vectors for the current particle positions (returned in `pos`).

This completes the `PSO` class. Please look at the full source code in the file `PSO.py`. Now, let's use our new class to see how it behaves, and how it compares with the `RO` class of Chapter 3.

4.3 Testing the PSO Class

We used the `fxy_gaussian.py` code presented in Chapter 3 to demonstrate the behavior of the `RO` class. Let's do the same here with the `PSO` class. Recall, `fxy_gaussian.py` is configured to find the minimum of Equation 3.1, a pair of two-dimensional Gaussians with minimum at $(-2.2, 4.3)$ (see Figure 3.5). All the code is already in `fxy_gaussian.py` and discussed in Section 3.3. The only change to use PSO is the invocation of the `swarm` object,

```
swarm = PSO(obj=obj, npart=npart, ndim=2, max_iter=miter, init=i,
            bounds=b, inertia=LinearInertia(), bare=False,
            bare_prob=0.5, ring=False, neighbors=4)
```

Most of this code is the same as before: the same objective function object (`obj`), the same number of particles (`npart`), dimensions (`2`), iterations (`miter`), initializer object (`i`), and bounds (`b`).

There are five new parameters, `inertia`, `bare`, `bare_prob`, `ring`, and `neighbors`. The first is set to an instance of the class `LinearInertia`, which, as we've discussed, decreases ω linearly on each swarm update from a default starting value of $\omega = 0.9$ down to $\omega = 0.5$ when the search is over. Notice, there is no `tol` keyword, so the swarm runs through all `miter` iterations.

The second and third parameters, `bare` and `bare_prob`, select whether or not to use the bare bones update rule or the canonical update rule and the likelihood of choosing to keep the particle best position for a dimension during the swarm update step, see Section 4.2.7. We'll start with the canonical rule (`bare=False`). The last two parameters decide whether or not to use a ring topology (`ring`) and, if so, how many neighbors each particle has, half on either side of it.

One run of `fxy_gaussian.py` using the default setup,

```
> python3 fxy_gaussian.py 10 100 PSO RI
```

gave,

```
f(-2.20002195, 4.29989850) = -4.9999999326
(20 swarm best updates)
```

which is quite close to the actual minimum, so we have some confidence that the `PSO` class is working. Summarizing ten runs gives (mean $\pm$ SE),

	PSO	*RO*
Minimum found	-4.3983 ± 0.3792	-3.7851 ± 0.8534
Swarm updates	18.4 ± 1.7	38.9 ± 6.2

where the `RO` class test from Section 3.3 is included for comparison.

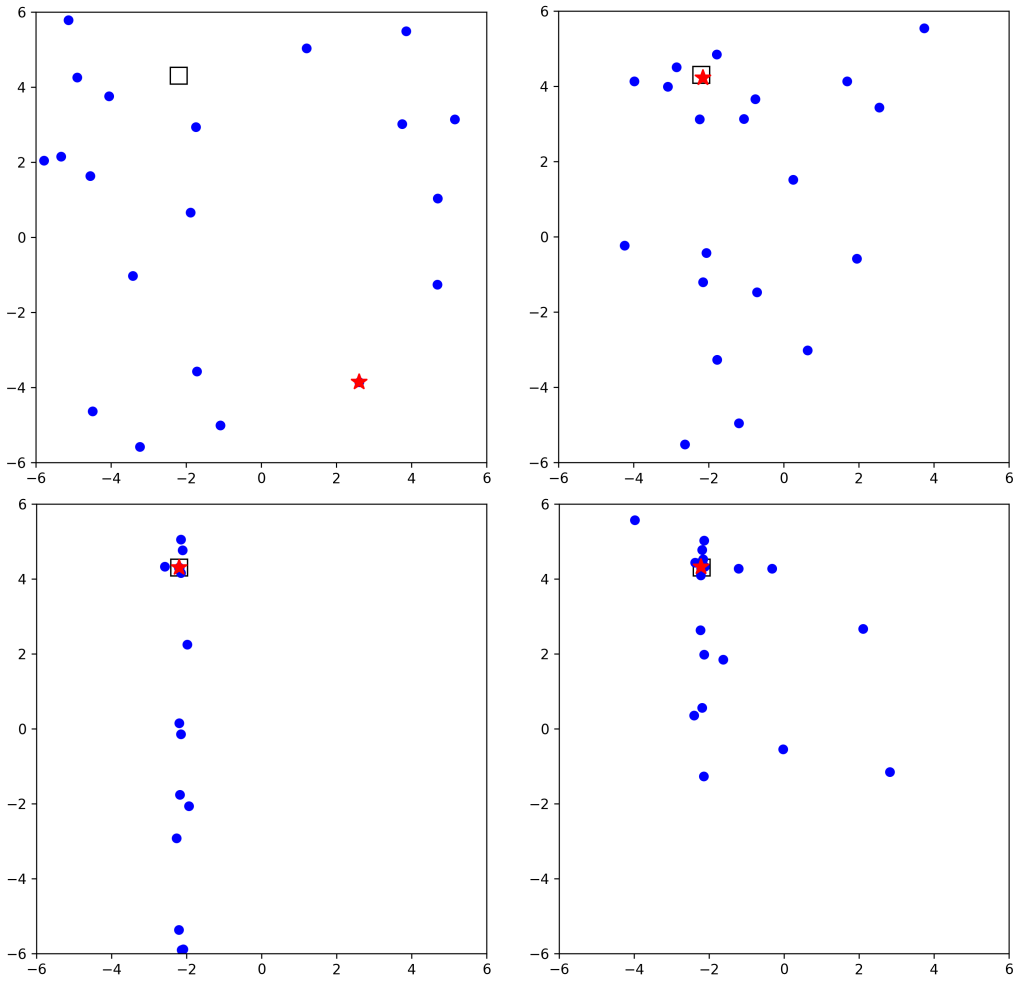


Figure 4.2: Frames from a PSO search with a swarm of twenty particles and 100 iterations. Clockwise from the upper left: initial swarm positions, frame 33, frame 66, and final swarm positions. Swarm particles are circles. The swarm best position is a star and the known global best position is a square.

We see immediately that PSO, for the same size swarm and number of iterations, is doing a better job than RO. A look at the output of the ten PSO runs shows that 8 of the 10 searches ended up at the overall minimum near $(-2.2, 4.3)$ while the remaining two searches found the shallower minimum near $(2.2, -4.3)$. Bumping `miter` from 100 to 1000 to search longer changes the results for ten new runs to,

Minimum found	-4.7000 ± 0.2846
Swarm updates	194.6 ± 46.2

which is misleading because nine of the ten runs found the global minimum of exactly -5.0 while the remaining one found the other minimum of exactly -2.0.

Let's see some frames from the swarm as it searches. For this, we'll go back to 100 iterations and show the same frame numbers as in Figure 3.6. The PSO frames are in Figure 4.2.

Clockwise from the upper left are the initial swarm positions, iteration 33, iteration 66,

and the final positions. The swarm best is the star, and the square is the known minimum. The swarm performed well. The final particle positions were quite different from the initial positions. For RO, the initial and final particle positions were broadly similar as each particle was conducting a local search and was unaware of better positions found by other particles. With PSO, the particles move in response to the movements of the rest of the swarm (remember, we're using a single global neighborhood for the moment). However, something about the overall motion of the PSO swarm might seem a little "off" to you. The minimum was found, and the particles moved towards it, but a peculiar vertical line of particles runs through the minimum. This is due to the Bounds object. A glance at `fxy_gaussian.py` shows that the Bounds object was told to randomly resample particle dimensions that hit the limit. For the 2D space we're searching, this means an out-of-bounds dimension will pick a new position along a vertical or horizontal line depending upon which dimension exceeds the limit. This explains why the swarm is stretched vertically.

Let's regenerate Figure 4.2, but this time we'll set `enforce="clip"` for the Bounds object. Recall, in Chapter 3, we set `fxy_gaussian.py` to use a fixed NumPy pseudorandom number seed. This means every run of the code will produce the same initial sequence of particle positions.

Now, by using clipping, when a particle's dimension exceeds a limit, it will be clipped to the limit value. We'll still get the global minimum, but the overall behavior of the swarm will be somewhat different; see Figure 4.3.

The frame 33 plot (upper right) shows some particles stuck along the upper edge of the search space. This is clipping in action. As swarm iterations proceed, the best is quickly found, and the particles all start to converge on it (lower right). The swarm has collapsed upon the global minimum by the end of the search (lower left). This collapse of the swarm, when clipping, is a characteristic behavior often seen in the canonical algorithm. We'll leave clipping on for the remainder of this chapter.

To this point, our experiments have not adjusted the cognitive (c_1) and social (c_2) scale factors. The default `PSO` class values are $c_1 = c_2 = 1.49$, as indicated above. Setting the values like this balances the tension for a particle between the region near the best position it has found and the overall best position of the neighborhood or swarm itself. The lower left of Figure 4.3 shows us the endpoint for this case.

Let's run two experiments, one where $c_1 > c_2$ and the other where $c_1 < c_2$. As before, we fix the NumPy pseudorandom number seed to get the same initial particle configuration. We need to set specific `c1` and `c2` values. We can do this easily in the constructor or after creating the `swarm` object by using Python's lenient object-oriented abilities. For example,

```
swarm = PSO(obj=obj, npart=npart, ndim=2, max_iter=miter, init=i,
            bounds=b, inertia=LinearInertia(), bare=False,
            bare_prob=0.5, ring=False, neighbors=4)
swarm.c1 = 0.745
swarm.c2 = 1.49
```

will create the `swarm` object and then set `c1` and `c2` directly making $c_1 = c_2/2$. Recall, we want all ω values used to be greater than $(c_1 + c_2)/2 - 1$, and our choices for `c1` and `c2` satisfy this requirement since the smallest ω returned by the `LinearInertia` class is $0.5 > (0.745 + 1.49)/2 - 1 = 0.1175$. Similarly, we can reverse the values to use $c_1 = 1.49$ and $c_2 = 0.745$. Running `fxy_gaussian.py` with these two modifications produces the final swarm configurations shown in Figure 4.4 with $c_1 = 0.745, c_2 = 1.49$ on the left and $c_1 = 1.49, c_2 = 0.745$ on the right.

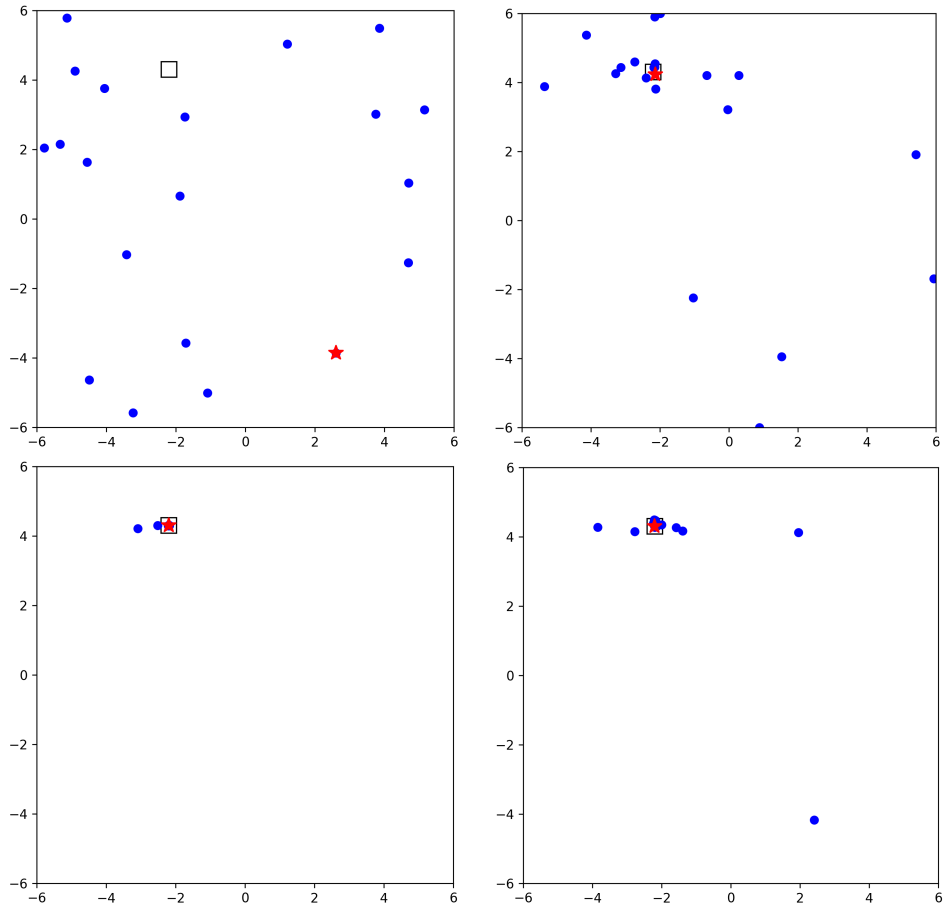


Figure 4.3: Frames from a PSO search that clips when particles move beyond the boundary. Clockwise from upper left: initial configuration, frame 33, frame 66, final configuration. Compare with Figure 4.2.

The left side of Figure 4.4 isn't an error; the entire swarm collapses on top of itself at the wrong minimum. Why? This case has $c_1 = c_2/2$, meaning the cognitive component, the part of the velocity update based on what the particle has learned for itself, is only half the social component's size. The social component is based on the neighborhood, but, for this example, the neighborhood is the entire swarm. Therefore, each swarm update step strongly favors moving in a direction closer to the best position the swarm already knows.

Isn't moving towards the global best position what we want? Yes and no. Here, the fixed random seed used to initialize the swarm places a particle near the $(2.2, -4.3)$ local minimum position and correctly marks it as the initial swarm best position (see upper left of Figure 4.3). Since all particles are now strongly attracted to the swarm best position, on each iteration, the particles move towards this location and further away from the true global minimum we want to find. Without much of a chance for local (cognitive) exploration, particles fall into the trap and move together on top of the local minimum. A quirk of the way the swarm was initialized has resulted in a failure. If the initial swarm positions had landed a particle near the global minimum, the search might have been successful. The emphasis on the social component of the velocity update has virtually removed the possibility that an individual particle, exploring the region it knows best, might find a new

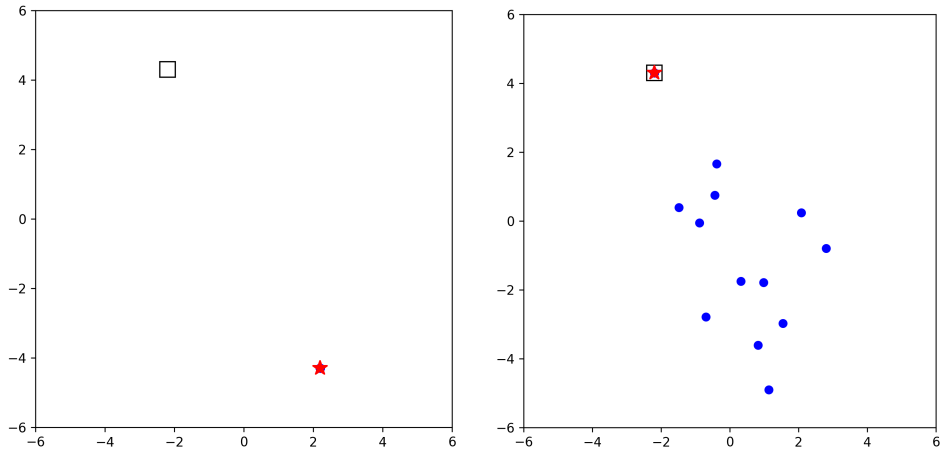


Figure 4.4: Final swarm configurations for $c_1 = 0.745, c_2 = 1.49$ (left) and $c_1 = 1.49, c_2 = 0.745$ (right).

global minimum.

The right side of Figure 4.4 tells a different story. In this case, the global minimum was found, but the swarm did not collapse upon itself, as in the case where $c_1 = c_2$. Here, the relationship between c_1 and c_2 is $c_1 = 2c_2$ so the cognitive component is emphasized over the social component. If you run this example and view the plots frame to frame, you'll see that the swarm wanders around. Overall, it is drawn to where the current best position is, but not as quickly as in the previous case. When an individual particle is near the global minimum, the swarm's best position shifts to it, and the swarm starts moving in that direction. When the search ends after 100 iterations, many particles are still wandering around exploring other regions of the search space, as seen in Figure 4.4.

We see now that letting $c_1 = c_2$ is a reasonable thing to do without any prior intuition as to whether c_1 or c_2 should be made more prominent. The first example shows that too much emphasis on the social component can fail. The swarm is composed of social butterflies who chase after whatever the crowd currently considers to be the best. By way of contrast, if we dare to continue the analogy, the particles in a random search are all rugged individualists who search on their own without influence. Finally, a reasonable balance between c_1 and c_2 reflects a population of well-adjusted individuals who think for themselves while being sensitive to social trends, should those trends be worthy of their attention.

Before concluding this extended example, let's take a cursory look at the behavior of a swarm using the bare bones update rule. We'll run with `bare=True` and change `bare_prob` (p_b) accordingly. However, to avoid being needlessly pedantic, we won't plot the results. Instead, we trust the reader to run the code and view the frames on his or her own.

So, change the code in `fxy_gaussian.py` like so,

```
swarm = PSO(obj=obj, npart=npart, ndim=2, max_iter=miter, init=i,bounds=b,
            inertia=LinearInertia(), bare=True, bare_prob=0.5,
            ring=False, neighbors=4)
```

where we'll set the probability of selecting a new particle position based on the swarm and particle best positions, per dimension, to 0.1, 0.5, or 0.9. For example, when $p_b = 0.1$, for any dimension of a particle, there is only a 10% chance that a new position will be selected

and a 90% chance that the particle's best-known position will be used. Review the code in Section 4.2.7 to refresh your memory of how a bare bones swarm update happens.

If we run the search for each of these probabilities and view the resulting frames, we see each run converges to the global minimum, but they do so in slightly different ways. For $p_b = 0.1$, the particles wander around slowly, almost like a random search, which makes sense as there is little opportunity to explore new places rapidly; the low probability usually results in small change to the current particle position. When $p_b = 0.9$, however, the particles make larger jumps and quickly locate the global minimum. As iterations continue, the swarm collapses upon itself. Finally, when $p_b = 0.5$, each component of a particle's position is updated in a manner balanced between exploring a new location along that dimension or sticking with what is comfortably familiar. We can imagine p_b acting like a slider between c_1 and c_2 in the canonical PSO case: move $p_b \to 1$, and the swarm explores more coarsely and is influenced by the swarm (neighborhood) best. Slide $p_b \to 0$, and the opposite happens, the particles stick with what they already know.

The PSO is a workhorse, but it takes some experimentation to gain intuition about how it behaves. There seems to be no end to the number of new variations appearing in the literature. The two versions we considered in this chapter, the original canonical version and the bare bones version, have parameters that need to be set appropriately. For canonical PSO, we need reasonable c_1 and c_2 values, to say nothing of a sensible ω update schedule. For bare bones, we have p_b. Also, in both cases, there is the topology to consider. We contented ourselves with the simplest topologies, global and ring, but there are others. The von Neumann topology, a lattice, is often considered to be one of the best. We didn't include it in the `PSO` class to keep things simple, but you should consider implementing it yourself if you find PSO is a tool you use frequently.

In the next chapter, we contemplate two newer algorithms selected from the zoo for how well they seem to perform on many problems and for their simplicity in not possessing any adjustable parameters (more or less). So, prepare for victory and watch out for the wolves.

Chapter 5

New Kids On The Block

We learned in Chapter 1 that there is a zoo's worth of new optimization algorithms to contemplate. Many are nature-inspired, at least loosely, while others are not, but all are part of the swarm intelligence zoo.

In this chapter, we'll explore two denizens of the zoo. The chosen denizens are Jaya ([17]) and the Grey Wolf Optimizer (GWO) ([18]). These algorithms were selected because they are relatively new, straightforward to implement, and have no tuning parameters.[1] Specifically, we'll discuss then implement Jaya and GWO (Section 5.1, Section 5.2) and compare them with PSO and RO using our now familiar 2D Gaussian optimization example (Section 5.3).

5.1 Jaya

Rao introduced Jaya in 2016 and followed up with a book, [19]. The book outlines many uses for Jaya and many variants. Here, we'll discuss the original Jaya algorithm. Jaya, Sanskrit for "victory", is a straightforward algorithm with no tuning parameters. The lack of tuning parameters makes Jaya attractive, and the original 2016 paper has been referenced over 1100 times to Fall 2021. Let's describe the algorithm and then its implementation using our framework.

5.1.1 Description

Jaya works with a swarm of positions in a search space, as do all of our algorithms. Many parts of its operation mirror random optimization quite closely, but, unlike RO, Jaya does use the knowledge of the swarm to influence candidate positions for particles. Algorithm 4, which we introduced to describe the operation of RO works just as well for Jaya. The only difference is in how the

Select a new position some random distance away from the current position

line is implemented. For Jaya, candidate positions are selected based on the current particle position's per component magnitude, the current best position in the swarm, and the current worst position in the swarm. A single equation governs the selection of new candidate positions on each swarm update step,

[1] This statement isn't strictly true. GWO has a tunable scale parameter that we'll explore later in the chapter.

```
class Jaya:
    def __init__(self, ...):
    def Results(self):
    def Initialize(self):
    def Done(self):
    def Evaluate(self, pos):
    def CandidatePositions(self):
    def Step(self):
    def Optimize(self):
```

Figure 5.1: Skeleton of the Jaya class.

Parameter	*Description*
obj	Objective function object (required)
npart	Number of particles in the swarm (10)
ndim	Number of dimensions in the search space (3)
max_iter	Maximum number of swarm iterations (200)
tol	Tolerance value (None)
init	Initializer object (None)
done	Done object (None)
bounds	Bounds object (None)

Table 5.1: The usual set of swarm algorithm parameters.

$$\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i + \boldsymbol{r}_1(\boldsymbol{x}_{\text{best}} - |\boldsymbol{x}_i|) - \boldsymbol{r}_2(\boldsymbol{x}_{\text{worst}} - |\boldsymbol{x}_i|) \tag{5.1}$$

Here $\boldsymbol{x}_i$ is the current position of particle i, $\boldsymbol{x}_{\text{best}}$ and $\boldsymbol{x}_{\text{worst}}$ are the current best and worst positions of any particle in the swarm, and $\boldsymbol{r}_1$, $\boldsymbol{r}_2$ are random vectors in $[0, 1)$.

Jaya seeks to move the candidate positions towards the swarm's best ($\boldsymbol{x}_{\text{best}}$) and away from the swarm's worst ($\boldsymbol{x}_{\text{worst}}$). Contrast this with PSO, which never considers the worst, but uses the tension between the particle's knowledge (cognitive) and the swarm's knowledge (social). In Jaya, there is no dichotomy of affinity; there is only attraction to the "good" and repulsion from the "bad".

Let's implement Jaya using our framework. We'll see that, as far as code is concerned, we only need to take the RO class and alter one method.

5.1.2 Implementation

A skeleton for the Jaya class is shown in Figure 5.1. The source code is in Jaya.py.

The constructor (__init__) accepts the usual suspects as keywords, see Table 5.1.

These are the typical set of parameters each swarm class uses. This is reasonable, as Jaya has no tuning parameters, so there are no additional things to think about when using the class.

The Results method returns the expected set of swarm best information (gbest, gpos, giter, and gidx) along with the final swarm position (pos) and the objective function values at those positions (vpos).

The Initialize method is line-for-line identical to the method from RO. See Section 3.2.2. Likewise, the Done, Evaluate, Step, and Optimize methods are identical to

those in RO. See Sections 3.2.4, 3.2.6, 3.2.3, and 3.2.1, respectively.

This leaves a single method to discuss, CandidatePositions. Let's take a look at it,

```
def CandidatePositions(self):
    pos = np.zeros((self.npart, self.ndim))
    f = np.argsort(self.vpos)
    best = self.pos[f[0]]
    worst= self.pos[f[-1]]
    for i in range(self.npart):
        r1 = np.random.random(self.ndim)
        r2 = np.random.random(self.ndim)
        pos[i] = self.pos[i] + r1*(best  - np.abs(self.pos[i])) -  \
                               r2*(worst - np.abs(self.pos[i]))
    if (self.bounds != None):
        pos = self.bounds.Limits(pos)
    return pos
```

The CandidatePositions method fills in pos, an npart by ndim NumPy array. First, it sorts the existing positions (self.pos) by their current objective function values (self.vpos) and selects the best and worst positions. Then, a candidate is generated for each particle by implementing Equation 5.1 using random vectors r1 and r2. As before, the candidate positions are checked for boundary violations by calling Limits on any supplied Bounds object.

In terms of implementation, this is all there is to Jaya. In a sense, it is even simpler than RO because the method for selecting candidate positions is based on simple vector math, no sampling from a Gaussian distribution, nor any need to think about adjusting a scaling factor (η) to the scope of the search space. Jaya handles this on its own by considering the best *and* worst of the swarm.

We'll seek victory with Jaya shortly. For now, let's see what a pack of wolves might do with an optimization problem.

5.2 The Grey Wolf Optimizer

The Grey Wolf Optimizer was introduced by Mirjalili, Mirjalili, and Lewis in [18]. Its popularity has grown since then with nearly 6300 references to date. As the name suggests, the optimizer seeks to model the behavior of a pack of grey wolves. Specifically, it models the social hierarchy of the wolves and through that hunting as searching, encircling, and attacking "prey". Here "prey" is the best position in the search space representing the solution to the problem at hand. As with Jaya, we'll describe the algorithm (Section 5.2.1) and then implement it (Section 5.2.2). Then we can get to the fun stuff in Section 5.3.

Before describing GWO, a clarification is in order. Recently work has demonstrated that GWO, along with several other popular nature-inspired algorithms, are not actually novel at all, but older PSO ideas wrapped in often strained metaphors.[2] That being the case, then, it is fair to wonder why GWO is included here. The emphasis in this book is on practicality and ease of application. GWO is popular and definitely works in terms of providing solutions to problems. In that sense, it doesn't matter if it's new or not. All the same, for the larger optimization field, it is critically important to understand what is

[2]Villalón, Christian Leonardo Camacho, Thomas Stützle, and Marco Dorigo. "Grey wolf, firefly and bat algorithms: Three widespread algorithms that do not contain any novelty." In International Conference on Swarm Intelligence, pp. 121-133. Springer, Cham, 2020.

novel and what is not. I suspect, in the end, that many of the myriads of nature-inspired algorithms will prove to be alternate takes on well-known approaches. But, if GWO, which is easy to try, works, then it works, so we'll keep it in our small collection of algorithms at the risk of alienating genuine optimization researchers.

5.2.1 Description

A wolf pack has an alpha, a lead wolf. It also has subordinate wolves in a hierarchy where the second-in-command, as it were, is the beta followed by the delta. There is a scapegoat wolf, designated omega, but for the GWO algorithm, all wolves (particles) that are not alpha, beta, or delta, are considered omega: they are followers only. The alpha, beta, and delta positions dominate the update for the swarm in that the motions of all particles are influenced by these three. During initialization, the top three best particle positions, those with the three smallest objective function values, are designated the initial alpha, beta, and delta. During the swarm update step, if any new position proves to be a better alpha, beta, or delta, then the corresponding leader is deposed and pushed down the hierarchy making the previous delta an omega, like the rest of the pack.

If you read [18], you'll encounter a full presentation of the rationale, however strained, behind the different equations used to model the behavior of grey wolves. For our purposes, we'll explain the practical: the parts of GWO pieced together to implement the search so we can evaluate the algorithm's performance.

A successful swarm optimization algorithm balances global and local search, exploration and exploitation. For GWO, this is accomplished via two equations used to define the $\boldsymbol{A}$ and $\boldsymbol{C}$ vectors used in the update process. Note, for GWO, we'll use bold uppercase letters as vectors to match how the algorithm is presented in [18].

The $\boldsymbol{A}$ vector depends upon a scalar, a, which itself depends upon a scale factor that we'll call η. The scalar a is similar to ω in PSO: it decreases linearly as the swarm searches, i.e., a is a function of the number of iterations performed and runs from η down to zero. The default value for η, indeed the only value for η used in most GWO implementations, is two.

The swarm update step first sets a, then, for each particle, calculates $\boldsymbol{A}$ and $\boldsymbol{C}$ vectors, one each for the alpha, beta, and delta. These are used to define temporary vectors $\boldsymbol{X1}$, $\boldsymbol{X2}$, and $\boldsymbol{X3}$. The mean of $\boldsymbol{X1}$, $\boldsymbol{X2}$, and $\boldsymbol{X3}$ is used to update the particle's position.

So, as a set of equations, the fraction of the update for particle $\boldsymbol{x}_i$ based on the position of the alpha wolf is,

$$a = \eta - \eta(j_{\text{current}}/j_{\max}) \tag{5.2}$$

$$\boldsymbol{A} = 2a\boldsymbol{r}_1 - a \tag{5.3}$$

$$\boldsymbol{C} = 2\boldsymbol{r}_2 \tag{5.4}$$

$$\boldsymbol{D} = |\boldsymbol{C}\boldsymbol{\alpha} - \boldsymbol{x}_i| \tag{5.5}$$

$$\boldsymbol{X1} = \boldsymbol{\alpha} - \boldsymbol{A}\boldsymbol{D} \tag{5.6}$$

where Equation 5.2 sets a for this iteration with j_{current} the current iteration number and $j_{\max}$ the maximum number of iterations. We then use a to calculate $\boldsymbol{A}$ with $\boldsymbol{r}_1$ a random vector drawn uniformly from $[0, 1)$ (Equation 5.3). Since $\boldsymbol{r}_2$ is a random vector in $[0, 1)$, Equation 5.4 makes $\boldsymbol{C}$ is a random vector in $[0, 2)$.

```
class GWO:
    def __init__(self, ...)
    def Results(self):
    def Initialize(self):
    def Done(self):
    def Step(self):
    def Optimize(self):
```

Figure 5.2: Skeleton of the GWO class.

The $\boldsymbol{C}$ vector is used in Equation 5.5 to define $\boldsymbol{D}$ as an offset from the current position of the alpha wolf, $\boldsymbol{\alpha}$. The offset is then used in Equation 5.6 to calculate the first temporary position vector, $\boldsymbol{X1}$. Notice that the offset is multiplied by $\boldsymbol{A}$ which in turn is itself set by a. The mode of the search is set by $\|\boldsymbol{A}\|$ and $\boldsymbol{C}$. When $\|\boldsymbol{A}\| > 1$, the algorithm is exploring the search space, but when $\|\boldsymbol{A}\| < 1$, the mode switches to attack, the swarm converges on the prey (best location).

The entire process that lead to $\boldsymbol{X1}$ is repreated using the beta ($\boldsymbol{\beta}$) and delta ($\boldsymbol{\delta}$) wolf positions to calculate $\boldsymbol{X2}$ and $\boldsymbol{X3}$, respectively. Then, finally, the mean of these vectors is used as the new position for particle i,

$$\boldsymbol{x}_i \leftarrow (\boldsymbol{X1} + \boldsymbol{X2} + \boldsymbol{X3})/3 \tag{5.7}$$

With all particles updated, the objective function is evaluated. New alpha, beta, and delta wolves are selected whenever a particle's objective function value is small enough to knock one of the leaders down a notch. The displaced leader takes the spot of the one below and so on. At all times, the best position found by the swarm is recorded and ultimately returned when the search is complete. Now, let's see how all of the above translates into code we can use.

5.2.2 Implementation

A skeleton for the `GWO` class is shown in Figure 5.2. The source code is in `GWO.py`. The skeleton is hauntingly familiar.

The constructor accepts the usual parameters along with `eta` to adjust η from its default value of two, if desired. This is like the `RO` constructor, where η plays a similar role. There are only two methods we need to discuss, `Initialize` and `Step`. All the others match those in `RO` and `Jaya` with corresponding names.

The `Initialize` method is,

```
def Initialize(self):
    self.initialized = True
    self.iterations = 0
    self.pos = self.init.InitializeSwarm()
    self.vpos= np.zeros(self.npart)
    for i in range(self.npart):
        self.vpos[i] = self.obj.Evaluate(self.pos[i])

    self.gidx = []
    self.gbest = []
    self.gpos = []
    self.giter = []
```

```
    idx = np.argmin(self.vpos)
    self.gidx.append(idx)
    self.gbest.append(self.vpos[idx])
    self.gpos.append(self.pos[idx].copy())
    self.giter.append(0)

    idx = np.argsort(self.vpos)
    self.alpha = self.pos[idx[0]].copy()
    self.valpha= self.vpos[idx[0]]
    self.beta  = self.pos[idx[1]].copy()
    self.vbeta = self.vpos[idx[1]]
    self.delta = self.pos[idx[2]].copy()
    self.vdelta= self.vpos[idx[2]]
```

The first section of code sets the iteration counter, initializes the particle positions (`pos`), and their objective function values (`vpos`). The second block of code selects the particle with the minimum objective function value and sets it as the initial swarm best.

To set up the wolf pack, we need an alpha, beta, and delta. So, we sort all the objective function values and select the smallest three to serve as initial `alpha`, `beta`, and `delta` positions along with their respective objective function values, `valpha`, `vbeta`, and `vdelta`.

The bulk of the GWO implementation is in `Step`. There are two main loops. The first moves each particle to a new position based on the position of the three leaders. The second loop then evaluates all the new positions, tracks the swarm best, and if there should be any change in pack leadership. In code,

```
    def Step(self):
1:      a = self.eta - self.eta*(self.iterations/self.max_iter)
        for i in range(self.npart):
            A = 2*a*np.random.random(self.ndim) - a
            C = 2*np.random.random(self.ndim)
            Dalpha = np.abs(C*self.alpha - self.pos[i])
2:          X1 = self.alpha - A*Dalpha
            A = 2*a*np.random.random(self.ndim) - a
            C = 2*np.random.random(self.ndim)
            Dbeta = np.abs(C*self.beta - self.pos[i])
3:          X2 = self.beta - A*Dbeta
            A = 2*a*np.random.random(self.ndim) - a
            C = 2*np.random.random(self.ndim)
            Ddelta = np.abs(C*self.delta - self.pos[i])
4:          X3 = self.delta - A*Ddelta
5:          self.pos[i,:] = (X1+X2+X3) / 3.0
        if (self.bounds != None):
            self.pos = self.bounds.Limits(self.pos)
6:      for i in range(self.npart):
            self.vpos[i] = self.obj.Evaluate(self.pos[i])
7:          if (self.vpos[i] < self.valpha):
                self.vdelta = self.vbeta
                self.delta = self.beta.copy()
                self.vbeta = self.valpha
                self.beta = self.alpha.copy()
                self.valpha = self.vpos[i]
                self.alpha = self.pos[i].copy()
8:          if (self.vpos[i] > self.valpha) and
               (self.vpos[i] < self.vbeta):
                self.vdelta = self.vbeta
```

```
                self.delta = self.beta.copy()
                self.vbeta = self.vpos[i]
                self.beta = self.pos[i].copy()
 9:          if (self.vpos[i] > self.valpha) and
                (self.vpos[i] < self.vbeta) and
                (self.vpos[i] < self.vdelta):
                self.vdelta = self.vpos[i]
                self.delta = self.pos[i].copy()
10:         if (self.valpha < self.gbest[-1]):
                self.gidx.append(i)
                self.gbest.append(self.valpha)
                self.gpos.append(self.alpha.copy())
                self.giter.append(self.iterations)
        self.iterations += 1
```

First, `a` is set according to the current iteration of the swarm. Recall, a goes from η down to zero over the number of iterations `(1)`. Then we loop to update each particle position based on the the current alpha, beta, and delta wolf locations. We calculate a new, temporary position, $\boldsymbol{X1}$, according to Equation 5.3 for A, Equation 5.4 for C, and Equation 5.5 for D_{alpha} `(2)`. These calculations are repeated for the two other leaders, beta `(3)` and delta `(4)`. Finally, the new particle position is set as the mean of the three temporary positions `(5)`. With all particles in their new positions, and after the boundary conditions are enforced by a call to `Limits`, the objective function is evaluated `(6)`.

While updating the objective function values for each particle, we check to see if the current particle is the new alpha `(7)`, beta `(8)`, or delta `(9)`. If any of these conditions are true, we make the current particle the corresponding leader and shift the displaced leaders down. So, if there is a new alpha, the old alpha becomes beta, and beta becomes delta. Similarly, if there is a new beta, the old beta becomes delta and likewise for a new delta. At the same time, we check to see if the current particle position is also the new global best `(10)`. To do this, we only need to consider if there is a new alpha as any new swarm best location will also be a new alpha. The swarm update step is now complete, so we bump the iteration counter to ensure a is set correctly for the next iteration.

Let's put both Jaya and GWO to the test comparing their performance with RO and PSO.

5.3 Testing Jaya and GWO

We'll continue using `fxy_gaussian.py`. We'll introduce a new level of testing here, as well, in part to put the new algorithms through their paces, and in part to illustrate additional ways to evaluate swarm optimization algorithms.

We know the code in `fxy_gaussian.py` is already configured to select the algorithm from the command line. The instantiations of the `swarm` object are the only parts specific to Jaya and GWO. They are as we might expect them to be at this point,

```
swarm= Jaya(obj=obj, npart=npart, ndim=2, max_iter=miter, init=i,bounds=b)
```

and,

```
swarm= GWO(obj=obj, npart=npart, ndim=2, max_iter=miter, init=i, bounds=b)
```

Resampling:

Algorithm	*S*	*F*	*W*
RO	96.20	0.04	3.76
PSO	98.20	0.00	1.80
Jaya	99.30	0.00	0.70
GWO ($\eta = 2$)	77.04	0.00	22.96
GWO ($\eta = 4$)	99.20	0.00	0.80

Clipping:

Algorithm	*S*	*F*	*W*
RO	96.10	0.00	3.90
PSO	77.24	0.00	22.76
Jaya	96.74	0.00	3.26
GWO ($\eta = 2$)	73.48	0.00	26.52
GWO ($\eta = 4$)	89.46	0.00	10.54

Table 5.2: Percentage of 5000 searches resulting in success (S), failure (F), or the wrong (W) minimum for resampling at the boundary (left) or clipping (right).

They set up the objective function object (`obj`), swarm size (`npart`), maximum number of iterations (`miter`), initialization object (`i`) and bounds (`b`). The simplicity of Jaya means that there are no additional parameters to set. For GWO, we'll keep $\eta = 2$ for the time being.

Let's do some test runs without fixing the seed value for the NumPy pseudorandom generator. We'll set the Bounds object to resample as well. The command lines to use are,

```
> python3 fxy_gaussian.py 10 100 JAYA RI
> python3 fxy_gaussian.py 10 100 GWO RI
```

The mean for each algorithm over ten runs is,

	Minimum found	*Swarm updates*	*Failures*
Jaya	-4.9999 ± 0.0000	15.8 ± 1.5	0
GWO	-3.4999 ± 0.4743	11.8 ± 0.8	5
PSO	-4.3983 ± 0.3792	18.4 ± 1.7	2
RO	-3.7851 ± 0.8534	38.9 ± 6.2	2

Note, in the table we've added the number of failed searches over the ten runs. A failed search did not find the global minimum of the Gaussian function at $(-2.2, 4.3)$. From this single example, Jaya is having a good day while GWO is not. It's also clear that the algorithms do not converge at the same rate. We'll plot this in a bit, but first, let's repeat the search 5000 times and track the proportion of failures. Recall, the NumPy pseudorandom number seed is not fixed in this case.

5.3.1 Success or Failure?

We'll count three different outcomes for each of the 5000 searches: success, failure, and wrong. A successful search finds the global minimum to within a distance of 0.3 from the true minimum. This means that the position returned by the search is no further away from the global minimum than that. A wrong result means that the final swarm best position was within 0.3 of the *other* local minimum. In that case, the swarm converged, only it converged in the wrong place. It was trapped in the local minimum and did not escape. Finally, any result that wasn't at either minimum is considered a failure. The code for this experiment is in `fxy_failures.py`. We'll run the test twice. The first time, we'll set the Bounds object to resample, and the second time, we'll clip instead. Table 5.2 shows us how the algorithms faired.

Consider only the left side of Table 5.2 for the moment. We see immediately that three of the algorithms faired quite well overall. None were perfect, but all were able to find the global minimum better than 96% of the time, and Jaya was victorious nearly 99% of the time. Also, when RO, PSO, and Jaya were not able to find the global minimum, they still converged on the other minimum, the only exception being RO a handful of times. The odd man (wolf) out in this case was GWO. If the default η value was used, the correct minimum was located only 77% of the time. However, increasing η moved GWO up into second place, right behind Jaya.

Now, compare the resampling results with the clipping results on the right side of Table 5.2. Here we see a different story. The RO algorithm performs the same, though, naturally, if you run `fxy_failures.py` yourself, you'll see slightly different results. Surprisingly, PSO is now performing on par with GWO in the resampling case. Jaya's performance drops to match RO and GWO is even worse. Clipping is not to the liking of many algorithms, at least for this example, a result worth keeping in mind when setting up your searches.

5.3.2 Dispersion

In Chapters 3 and 4, we fixed the pseudorandom number seed in `fxy_gaussian.py` to generate frames showing the evolution of the swarms as they attempted to find the global minimum. You may have noticed an extra file appeared in the frames directory, `dispersion.npy`. We ignored this file then, but let's make use of it now.

If you look at the code in `fxy_gaussian.py`, you'll see that when writing frames to disk there is an extra function called,

```
def Dispersion(swarm, i, d):
    x,y = swarm.pos[:,0], swarm.pos[:,1]
    dx = x.max() - x.min()
    dy = y.max() - y.min()
    d[i] = (dx + dy) / 2.0
```

The `Dispersion` function accepts a swarm object, an iteration number (`i`) and a vector, `d`. The point of this function is to calculate a measure of how dispersed throughout the search space the swarm currently is. We're calling this the "dispersion." It is nothing more than the mean of the swarm range in each direction. The more compact the swarm is, the smaller this number will be. The `Dispersion` function updates `d` for the current iteration of the swarm. When the search is complete, the `d` vector is written to disk in `dispersion.npy`. If we fix the pseudorandom number seed to the same value as before, 8675309, we can generate dispersion files for RO, PSO, Jaya, GWO ($\eta = 2$), and GWO ($\eta = 4$). We can then plot the dispersion of the swarms as a function of iteration number (see the file `dispersion_plot.py`).

Figure 5.3 shows how each swarm behaves for both resampling and clipping boundary conditions. The plots show the trend line and a symbol every fifth iteration.

How should we interpret these plots? Let's start with resampling on the top. We immediately notice both GWO plots converge on zero. In these cases, the swarms did collapse; each particle was virtually on top of all the others. Moreover, there is a definite phase transition, as it were. The swarms were oscillating between more or less diverse, then, suddenly, they quickly converge. Both RO and Jaya avoid collapse and show a smooth, shallow decrease in dispersion over the entire search. PSO is similar, but oscillates, much like GWO does. Still, there is no collapse at the end of the search.

Now consider the bottom of Figure 5.3, the clipping case. The RO curve is identical, as expected because no RO particle hits the boundary, so, given the fixed NumPy seed, the searches will match. The other searches, however, all collapse to zero or close to it. PSO oscillates, as in the resample case, but shows a strong negative slope down to virtually zero by search's end. GWO ($\eta = 4$) roughly follows PSO: oscillations and a steady decrease in swarm diversity. GWO ($\eta = 2$) drops diversity rapidly and then declines to zero. However, in this case, we must remember that GWO ($\eta = 2$) is converging on the *wrong* minimum. One particle starts near the wrong minimum, and, due to the η value, the swarm doesn't escape; it all falls into the trap. Finally, Jaya shows a rapid drop in dispersion around iteration 38. This marks the point where the last particle far from the global minimum made the jump to join the rest of the swarm.

5.3.3 Convergence

How the swarms converge to the global minimum as a function of iteration, algorithm type, and initial configuration is our next adventure. We're interested in how well and how quickly the swarm locates the global minimum without concern about how dispersed the swarm is. We'll use the same seed as before to plot the current global best objective function value as a function of the iteration. We'll create two plots: one for resampling and the other for clipping. Again, we show the trend line and a symbol every fifth iteration.

The code for the convergence plots is in the file `fxy_convergence.py`. Each algorithm tracks the evolution of the swarm and returns the list of swarm best updates by calling the `Results` method. From this list, we can generate a per iteration sequence like so,

```
res = swarm.Results()
g = np.zeros(miter)
for j,i in enumerate(res["giter"]):
    g[i:] = res["gbest"][j]
```

where `miter` is the number of swarm iterations. For every swarm best update, we noted the iteration number when it happened (`giter`). So, if we set every output value in `g`, starting with that index, to the new swarm best that occurred, we'll fill `g` appropriately until the next swarm best update happened, in which case, we update `g` from that point onward. In the end, `g` contains the curve we want to plot. Running `points_plot.py` generates Figure 5.4 with the convergence curves for each algorithm, the resampling case on the top, the clipping case on the bottom.

Both Jaya and PSO behave similarly and converge to the global minimum reasonably quickly, by iteration 35 when resampling and even earlier, around iteration 15, when clipping. Random optimization is identical regardless of boundary rules and converges smoothly as the search progresses, but with some sudden changes in slope.

As before, the starkest difference is with GWO. For $\eta = 2$, the resampling curve rapidly converges to the correct value, but for this same η, the clipping curve is trapped in the other minima. When $\eta = 4$, both resampling and clipping locate the correct minimum.

5.3.4 Precision

Our convergence plot, Figure 5.4, is visual and only shows 100 iterations of the swarms. One good metric to use when comparing swarm algorithms is the precision, which we'll

Algorithm	*Error (mean± SE)*
RO	$0.0000103433957914 \pm 0.0000016743641739$
PSO	$0.0000000000000000 \pm 0.0000000000000000$
Jaya	$0.0000190824476841 \pm 0.0000030181708582$
GWO ($\eta = 2$)	$0.0000003834092728 \pm 0.0000000476242994$
GWO ($\eta = 4$)	$0.0000026417375218 \pm 0.0000005018126146$

Table 5.3: Error between the known minimum value, -5.0, and the minimum found by the swarm for 40 searches (20 particles, 1000 iterations).

define as the error between the known actual minimum of the test function and the final minimum value found by the swarm.

We can calculate this precision for the Gaussian we've been using by randomly initializing the swarms and running the search for many more iterations. Repeating searches many times allows us to report the mean and standard error of the final best objective function value found. The code to use is in `fxy_precision.py`. It's run in much the same way as `fxy_gaussian.py`, but accepts one more command line parameter, the number of times to search before reporting the overall mean and standard error of the absolute difference between the final position and the known minimum value of exactly -5.0. Note, we're only interested in searches that succeed, so the code loops until the desired number of successful searches have happened. Those that fall into the other minimum are ignored.

We'll run each search with 20 particles but use 1000 swarm updates before looking at the minimum found. And, we'll repeat each search 40 times. The results are in Table 5.3.

Table 5.3 is shown without scientific notation to make it visually obvious that the algorithms show differences in precision, for this sample problem, of up to several orders of magnitude in some cases. When run longer, PSO zeros in on the actual minimum position with as much precision as Python can give for a floating-point number. Python uses IEEE 754 binary64 format (C type `double`). This format stores numbers internally as binary floating-point values using a 53-bit significand. This implies full possible precision in the value returned by PSO to approximately 16 decimals. The best we could hope for.

The next most precise result is GWO for $\eta = 2$, though this result is some nine orders of magnitude bigger. The $\eta = 4$ result is an order of magnitude larger than the $\eta = 2$ result. So, GWO found the minimum position, but whether the location is "good enough" depends on the problem. In many practical cases, it certainly will be good enough, but the difference is too significant for some applications. The RO result is an order of magnitude larger still than the GWO $\eta = 4$ result, and the Jaya result is twice the RO result making Jaya the least precise algorithm for this experiment. Again, the application decides if Jaya is good enough.

We fixed the number of iterations in Table 5.3 and looked at the precision of the global best when that many iterations were complete. Let's switch things up a bit now and see how many iterations, on average, it takes for the swarm to converge to a particular error level. The code leading to these results is in `fxy_precision2.py`, and the results themselves are in Table 5.4.

Much is happening in Table 5.4. Let's break it down. For the most significant error, 10^{-4}, we're able to run all of the algorithms. This is a fairly significant error, though possibly adequate for some problems. RO is the winner in this case, as it requires about 150 updates on average. Jaya comes next, followed by PSO. So far, so good for RO and

Algorithm	*Error*	*Iterations (mean± SE)*
RO	10^{-4}	148 ± 25
PSO		1238 ± 361
Jaya		466 ± 68
GWO ($\eta = 2$)		7749 ± 1039
GWO ($\eta = 4$)		10964 ± 1588
RO	10^{-5}	2420 ± 876
PSO		1318 ± 464
Jaya		1593 ± 209
RO	10^{-6}	15126 ± 5588
PSO		1752 ± 456
Jaya		3018 ± 585
RO	10^{-7}	78114 ± 17141
PSO		2516 ± 571
Jaya		13175 ± 1888

Table 5.4: Mean number of iterations over ten searches with 20 particles to locate the global minimum to less than the given error (mean± SE).

Jaya. GWO performs poorly on this test. Many iterations are needed to get the error down to less than 10^{-4}.

We need to clarify something before describing the remainder of Table 5.4. Both PSO, through the `LinearInertia` class, and GWO, when it adjusts a, depend upon the maximum number of iterations specified when the swarm object is created. This requirement poses a bit of a problem when we're interested in how many iterations it takes for the swarm to reach a given error rate, on average. We cannot merely iterate the swarm forever without wanting to set an upper limit on the number of iterations. Setting such a limit, say to 1,000,000, means that ω (PSO) and a (GWO) will decrease at a painfully slow rate. So, what are we to do? In `fxy_precision2.py`, the maximum number of iterations *is* set to 1,000,000 for RO and Jaya. However, for PSO, it's set to 10,000 and for GWO 20,000. The latter two values are empirical guesses, and they do affect the results.

For example, in Table 5.4, PSO needed nearly 1300 iterations on average to reach an error below 10^{-4}. However, in Table 5.3, PSO was at zero error after 1000 iterations. This isn't a bug, it's due to the effect of the maximum number of iterations on the value of ω and the rate at which it decreases. For example, let's seek the number of iterations, on average, leading to an error of less than 10^{-8} for PSO as we change the maximum number of possible iterations, see `fxy_pso_precision.py`. We get these means over ten runs for each iteration maximum,

Maximum Iterations	*Iterations to error* $< 10^{-8}$
100	120 ± 7
500	236 ± 10
1000	606 ± 82
5000	1376 ± 374
10000	3324 ± 724
50000	5719 ± 2548
100000	22614 ± 5098

This effect explains the PSO results shown in Table 5.4. PSO can converge faster, if we take the time to set the maximum number of iterations to the smallest value necessary, assuming we're using a linearly decreasing ω. What is "the smallest value necessary"? That's a good question, only experimentation will tell, or the use of a smarter way to schedule ω as the search progresses.

The GWO results in Table 5.4 stop after the 10^{-4} case because when the desired error is lower, the swarm doesn't reach it before exhausting the 20,000 maximum number of iterations. The takeaway here is that GWO, for this example, is not particularly precise. Again, whether it is precise enough depends upon the application.

Let's finish discussing Table 5.4. As we increase our demand for precision, we see the behavior we might expect from RO: the unguided search requires ever-increasing iterations to reach the desired error threshold. The same is true of PSO and Jaya, but the rate at which the number of iterations grows is less. Still, a precision of 10^{-7} is quite high, and Jaya can find it in a reasonable number of iterations for many applications.

Precision matters, setting swarm parameters intelligently matters, too, but, as always, experimentation and intuition come into play. PSO did quite well on these tests, as did Jaya. RO performed surprisingly well, overall, but GWO was disappointing. Let's conclude this section by taking a cursory look at the clock time associated with each of the algorithms.

5.3.5 Runtime

It's fair to ask whether one algorithm is faster or slower than another in terms of clock time for the same number of swarm iterations. Naturally, the runtime is highly dependent upon the implementation, and, for pedagogical purposes, we've spared all expense at highly efficient implementations. Therefore, it's with some trepidation that we dare to look at how fast the algorithms are. We'll do so, with the understanding that the relative clock time is only a hint of how efficient the algorithms are and with the defense that, because of our framework, many implementations are similar in structure. A full evaluation of the algorithms' actual performance, from a computer science perspective, is beyond what we're after here.

With the above disclaimers in place, we can measure the mean runtime for a fixed number of swarm updates. The code we're using is in the file `fxy_runtime.py`. Calculating the mean runtime, in seconds, for ten runs of a search of 20 particles and 5000 iterations leads to,

Algorithm	*Runtime (seconds, mean± SE)*
RO	1.4113 ± 0.0014
PSO	4.0503 ± 0.0170
Jaya	2.5692 ± 0.0049
GWO ($\eta = 2$)	6.9775 ± 0.0057
GWO ($\eta = 4$)	7.1043 ± 0.0080

Both RO and Jaya run quickly. PSO is in the middle, and GWO is significantly slower than any of the others. The implementations use multiple loops over the particles to select new positions and then to evaluate the objective function at each position. RO is likely faster because candidate positions are chosen by a single-line call to a NumPy function, thereby eliminating one of the slower `for` loops in plain Python. The remaining runtime differences are likely to be overcome by more efficient implementations. The runtimes above should be used as a guide for these specific implementations only.

5.3.6 Evaluation

In this section, we tested Jaya and GWO and compared them to the swarm algorithms we already know, RO and PSO. We looked at how often a randomly initialized search succeeded, how the swarms changed as they searched (dispersion), how quickly they converged to the global minimum (convergence), how well they located the minimum (precision), and how fast the algorithms searched (runtime). Granted, we did this for a single, two-dimensional search space, but the results were illustrative all the same.

What should we make of these new algorithms? Jaya is simplicity itself. It performed admirably, and its complete lack of any tuning parameters makes it attractive; just set it up and go. The Grey Wolf Optimizer did well enough once we adjusted its one parameter, the one we called η, but was less than sterling when fine precision was required. As mentioned above, most implementations fix η at two and do not adjust it, nor let users adjust it. We didn't even consider adjusting the scale of C, as suggested in [18]. Some reviews of GWO have been critical, both of the nature-based inspiration as yet another example of a strained metaphor, and of a perceived ineffective update rule.[3] Is the criticism of GWO unwarranted? Only time will tell, but we'll continue to use GWO, and Jaya, for all our experiments going forward, especially for those in the second part of this book.

For now, let's seek victory elsewhere and forget about who's the leader of the pack. We'll shift paradigms and consider the first of two evolutionary algorithms we'll sneak into our framework. We do this in part because we can, and, even more so, because they are useful tools and we should be more than just one-trick ponies. One might be tempted to consider these evolutionary algorithms wolves in sheep's clothing, but that would be taking things a bit too far. Regardless, let's explore the wonderful world of evolution, at least as far as we can use it to solve difficult optimization problems.

[3] See the pagmo2 documentation, `https://esa.github.io/pagmo2/docs/cpp/algorithms/gwo.html`.

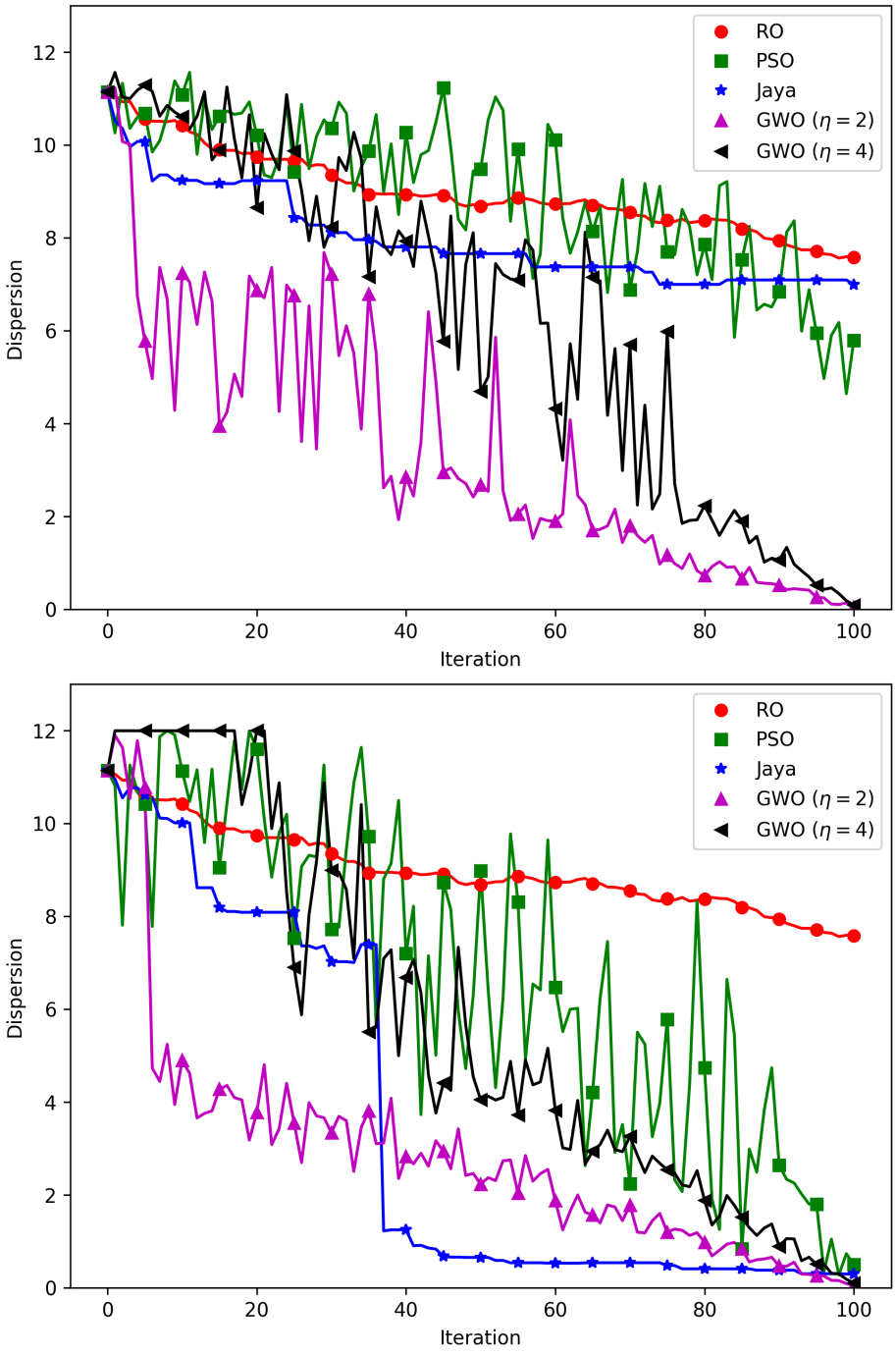


Figure 5.3: Dispersion as a function of swarm iteration number for resampling (top) and clipping (bottom). Note, the GWO ($\eta = 2$) case converged on the wrong minimum in the clipping case.

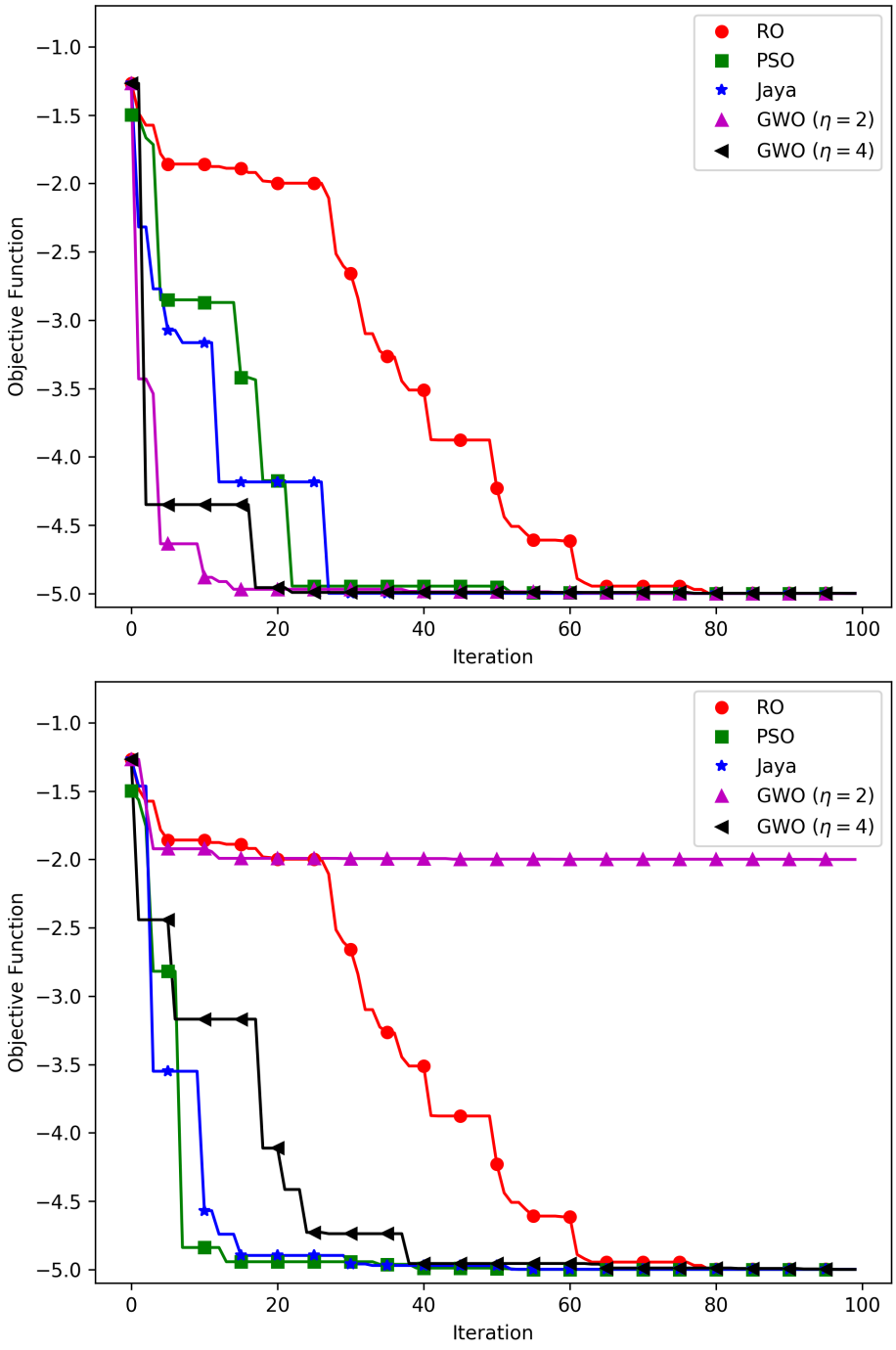


Figure 5.4: Swarm best as a function of iteration number. (Top) resampling bounds, (bottom) clipping bounds. The pseudorandom number seed was fixed so each swarm evolved from the same initial configuration.

Chapter 6

Genetic Algorithm

The algorithms we've developed so far fit solidly within the swarm intelligence family. This chapter, and the one that follows, sneak in two other algorithms that are not strictly "swarm intelligence," but are instead evolutionary algorithms. However, their architecture is amenable to our framework, so we feel justified in including them, doubly so because of their practical utility.

In this chapter, we'll discuss and present code for the genetic algorithm class. In simplest terms, a genetic algorithm employs concepts from biological evolution and uses them to evolve a population of agents, "particles" in our language, to solve a problem. In biological evolution, there is no overall goal or direction; organisms evolve because that's what they do as a consequence of factors like mutation and in response to natural selection. For a genetic algorithm, evolution is highly directed. An overlord is watching and selecting who gets to breed, and how often, and what kind of mutations occur and when, all in an attempt to minimize the objective function.

Section 6.1 describes our implementation of the genetic algorithm, one that plays nicely with our framework. In Section 6.2, we detail the algorithm code as we have for all the previous algorithms. As before, we expound on the differences knowing many methods of the `GA` class are identical to those in previous classes.

Finally, in Section 6.3, we test the `GA` class against our other algorithms. We'll persist with our running test example of 2D Gaussians and then expand it to five dimensions.

6.1 Making Darwin Proud

Darwin's publication of "On the Origin of Species" in 1859 was a watershed moment for the biological sciences and, indeed, for all of humanity. Darwin didn't discover all the ideas in the book, Wallace deserves some credit, and credit is due to others who came before Darwin, but in the end, the synthesis was mainly his. The book introduced the concept of natural selection, often, somewhat incorrectly, simplified to "survival of the fittest."

For our purposes, we'll pare down what evolution is and reduce it to the phrase above. After all, we have an explicit way of measuring fitness, which is only implicitly measured in biological evolution by observing the gene pool of future generations. We'll simplify the drivers of evolution and keep only random mutation and crossover to simulate the process of mating between agents. To be consistent, we'll persist in calling the agents "particles" – only this time the particles can breed with each other like biological agents.

Along with random mutation and crossover due to breeding, which we'll be explicit about

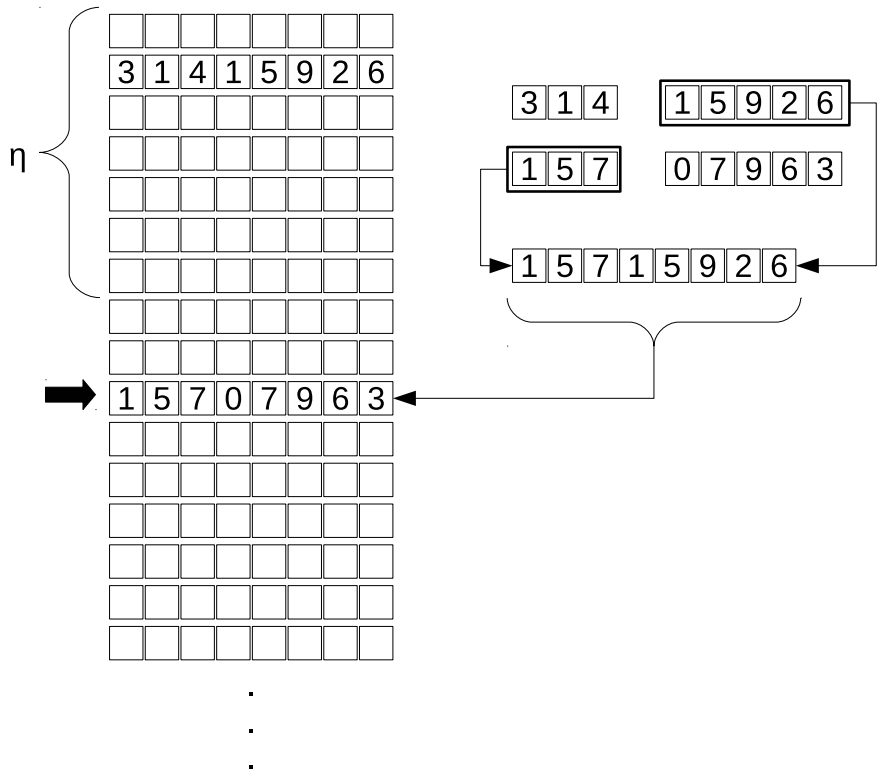


Figure 6.1: Implementing crossover to replace a parent particle with a child.

momentarily, we have an additional knob we can turn. We'll control who the particles may breed with in much the same way as a dog breeder. Turning this knob affects the rate at which the swarm converges.

The title of this section is "Making Darwin Proud." Naturally, we don't know if he would be proud of such an application of his theory, and, perhaps, he might frown on it as not being based in actual biology. However, once he understood the utility, I suspect he would be pleased with the parallel and happy to learn that his core concepts are more widely applicable than he might have first supposed.

Most presentations of a genetic algorithm jump down to the level of genes and talk about mutation or crossover as in chromosomal crossover during conception. We'll be more abstract than that so we can continue to use our framework.

We represent particles as floating-point vectors of some dimensionality, the dimensionality of the problem space we are exploring. We'll continue to do so for the genetic algorithm. Mutation becomes quite simple in this situation: we pick a dimension of the particle's current position at random and change it. The new value is bounded by the `Limit` method of any supplied `Bounds` object. Mutation is little different from resampling when a boundary violation occurs.

Crossover is similarly straightforward in our case. Unlike a genetic programming algorithm that needs to select specific places in the evolved code when an expression subtree is replaced with another from a separate individual, we have no structure to our particles, only locations along different dimensions with the particle representing a point in $\mathbb{R}^n$. There-

Algorithm 7 Genetic algorithm.

Input: An objective function, bounds, and initialization type
Output: The best position found by the swarm

```
for each particle do
    Select an initial position within the bounds of the search space
    Evaluate the objective function at this position
end for
Store the best initial particle position as the swarm global best position
while not done do
    for each particle do
        Evolve the particle
        Evaluate the new position
        if new position fitness < swarm best position fitness then
            Store the new global best
        end if
    end for
    Increment the iteration counter
end while
```

fore, to crossover two individuals, we select a dimension at random for the first individual and replace all values after that dimension with the corresponding values from the second individual. This process is quite analogous to chromosomal crossover.

Figure 6.1 shows visually how we'll implement crossover. The current particle, marked with the large arrow on the left, is to be updated. A member of the top-performing particles (η) is selected as a mate. The right side of the diagram illustrates crossover. A dimension is selected, here dimension three, and the child is formed by keeping elements zero through two of the first parent and updating the rest with the corresponding elements of the second parent. Finally, the child immediately replaces the parent in the swarm.

While we have no specific name for it, the knob deciding which particles the current particle may breed with is implemented as a fraction of the swarm when ranked by objective function value. Only the top N particles, where $N = \lfloor \eta n_{\text{particles}} \rfloor$, η a fraction, are available as mates, though all particles have an opportunity to breed. Doing this lets particles in the lower part of the swarm, those with larger objective function values, have an opportunity to improve by incorporating "better genes" into their offspring's genetic code.

When particle i breeds, the child replaces the parent in the next iteration of the swarm. There are many other ways to handle the approach to breeding, but we'll keep this one throughout the book. As an experiment for the reader, it might be fun to think of other ways to accomplish this. The main point is that more fit members of the swarm should propagate to the next iteration in some fashion. Furthermore, we'll persist the best performing individual in each swarm iteration and move it to the next iteration intact.

Algorithm 7 shows our approach to the genetic algorithm; it should look quite familiar. In fact, Algorithm 7 is nearly identical to Algorithm 4 for the `RO` class. However, instead of a line reading,

```
Select a new position some random distance away from the current position
```

we have,

Evolve the particle

We discussed above what we mean by "evolve the particle," and we'll develop the code in Section 6.2 below. All other parts of the algorithm are directly analogous to the other swarm algorithms we've examined. This is by design, so we can introduce the genetic algorithm and use it with our framework.

The GA class has three parameters. How should we set them? The short answer is "it depends". The best set of parameters, if there is one, is likely specific to the problem. However, general guidelines are possible. For example, the fraction of the swarm available for breeding, η, defaults to the upper 50%. If we set $\eta = 1$, the entire swarm is now available for breeding, meaning there is no preference given for particles that are better performing. Is this a bad thing to do? Perhaps not, it's easy to imagine that two poorly performing particles might mate and produce an excellent performer. What if we set $\eta = 0.1$ or even $\eta = 0.05$? We'll force all particles to breed with top performers, but the gene pool, as it were, will be correspondingly smaller. One imagines such a swarm might become stuck in a local minimum and be unable to escape for lack of genetic diversity. As a rule of thumb, then, $\eta = 0.5$ seems like a good starting point, though smaller η and a larger swarm might prove useful. As with all swarm algorithms, experimentation is required. For many practical applications, we need only one good solution, after that, our use for the swarm is over, so experimentation to get that one good solution is not usually much of an issue unless the objective function is expensive to calculate.

What about crossover probability? In the United States, by age 50, roughly 85% of women have had at least one child. By this, we might say evolution has provided a default crossover probability for us of about 0.85. Indeed, perusing the literature shows that the crossover probability is often around 80%, which we'll use as our default. Again, any ideal value will likely be problem-specific. We do want crossover to be rather high to encourage diversity in the swarm from generation to generation. However, if it is too high, breeding might wash out beneficial "genes", here meaning individuals that might have spread better solutions to the rest of the swarm. On the other hand, a tiny crossover probability means most of the particles will barely change from iteration to iteration, and we might expect the search to be painfully slow. Finally, what about mutation? The effect of mutation as a driver for evolution is well understood. However, most mutations are not beneficial, and evolution by mutation alone is painfully slow; witness the billions of years needed for prokaryotes to evolve into eukaryotes, let alone sexual reproduction. We set the default mutation rate to 5%, meaning on each swarm update, a particle has a 5% chance of mutating. When a particle is selected for mutation, the particle's affected dimension is chosen from a uniform distribution. Doing this makes sense in a generic system where the position vector's numerical values are interpreted as just that, values along some dimension in a high dimensional space. However, the entire point of using a swarm algorithm is to map the solution of a problem into movement through this high-dimensional space to some best location. This implies some dimensions of the particle's position might be more critical to the solution than others. If that's the case, is the best option for selecting which parameter to mutate a uniform distribution? Might it be that another distribution would be better? I am unaware of investigations along these lines, probably because evolution is purely random, but there are strongly conserved portions of the genome that are seldom mutated. Regardless, it's entertaining to consider the possible effect of applying a non-uniform distribution to the selection of which dimension is mutated. However, such a distribution is most definitely problem-specific and requires the infusion of prior knowledge

```
class GA:
    def __init__(self, ...):
    def Results(self):
    def Initialize(self):
    def Done(self):
    def Evaluate(self, pos):
    def Mutate(self, idx):
    def Crossover(self, a, idx):
    def Evolve(self):
    def Step(self):
    def Optimize(self):
```

Figure 6.2: Outline of the GA class.

or intuition by the practitioner configuring the swarm search. For our purposes, we'll stick with uniform selection of the mutated "gene" and use a low mutation rate most of the time.

Let's look at the code for the `GA` class emphasizing its differences from the other optimization classes.

6.2 The GA Class

The outline of the `GA` class is in Figure 6.2. Many methods are identical to those we've seen before. The `Step` method is altered slightly, and there are three new methods: `Evolve`, `Mutate`, and `Crossover`. The source code for the `GA` class is in the file `GA.py`.

The `GA` class constructor accepts all the parameters we've used before, see Table 5.1. Additionally, it accepts three new parameters,

Parameter	*Description*
`CR`	Crossover probability (`0.8`)
`F`	Mutation probability (`0.05`)
`top`	Top fraction to breed (`0.5`)

We use `CR` to set the probability of a particle breeding during a swarm update step (crossover). If a random float is less than `CR`, the particle breeds on this iteration with a mate selected from the `top` performing fraction of the swarm when sorted by current objective function value. Finally, in addition to the opportunity to breed, a particle may experience a random mutation with probability `F`. Using "CR" and "F" is intentional. We'll see these same quantities again in Chapter 7 on differential evolution. Notice, on any swarm update, a particle may or may not breed. If it does breed, the child immediately replaces the parent. Also, a particle may or may not randomly mutate. So, on any iteration, one of these occurs: the particle does not breed or mutate, the particle breeds and replaces itself with a child, the particle mutates, or the particle breeds, and the resulting child immediately mutates.

6.2.1 Step

The `Step` method implements a single swarm update. The GA `Step` method is similar to the RO `Step` method except instead of proposing possible new positions for each particle, it calls `Evolve` to update the swarm position vectors directly. In code, then,

```
def Step(self):
    self.Evolve()
    self.vpos = self.Evaluate(self.pos)
    for i in range(self.npart):
        if (self.vpos[i] < self.gbest[-1]):
            self.gbest.append(self.vpos[i])
            self.gpos.append(self.pos[i].copy())
            self.gidx.append(i)
            self.giter.append(self.iterations)
    self.iterations += 1
```

The swarm positions are evolved *en masse* and the new positions evaluated. Then, each particle is examined to determine if it represents a new swarm best position. If the particle is a new swarm best, it's kept, along with the objective function value (`vpos`). As usual, the iteration counter is incremented to signal that the swarm update is complete.

6.2.2 Evolve

The `Evolve` method is called during each swarm update step. Its purpose is to give the particles an opportunity to breed or mutate. The code is a simple loop over the particles,

```
def Evolve(self):
    idx = np.argsort(self.vpos)
    for k,i in enumerate(idx):
        if (k == 0):
            continue
        if (np.random.random() < self.CR):
            self.Crossover(i, idx)
        if (np.random.random() < self.F):
            self.Mutate(i)
    if (self.bounds != None):
        self.pos = self.bounds.Limits(self.pos)
```

First, we get the sequence of indices into the current particle positions and objective function values in order from smallest to largest. Then we loop over this sequence. The very first particle in the sequence, the current best position, but not necessarily the swarm best, is left as-is to propagate it intact to the next iteration. For all other particles, we ask whether or not to breed the particle. If so, `Crossover` is called. Next, we ask if we want to mutate the particle. Again, if so, `Mutate` is called. Finally, a call to `Limits` on any supplied `Bounds` object ensures that the evolved particles stay in bounds.

6.2.3 Mutate

The code for `Mutate` is,

```
def Mutate(self, idx):
    j = np.random.randint(0,self.ndim)
    if (self.bounds != None):
        self.pos[idx,j] = self.bounds.lower[j] +
            np.random.random()*(self.bounds.upper[j]-self.bounds.lower[j])
    else:
        lower = self.pos[:,j].min()
        upper = self.pos[:,j].max()
        self.pos[idx,j] = lower + np.random.random()*(upper-lower)
```

The particle index is passed in (`idx`), and a dimension along the particle's position is selected at random (`j`). If there is a `Bounds` object, an arbitrary position within the bounds for the selected dimension is used to update the particle. This is precisely what happens when a dimension exceeds a given boundary. If no `Bounds` object was supplied, a random position among the range of the selected dimension for the current swarm is used.

6.2.4 Crossover

The code for `Crossover` is,

```
def Crossover(self, a, idx):
    n = int(self.top*self.npart)
    b = idx[np.random.randint(0, n)]
    while (a==b):
        b = idx[np.random.randint(0, n)]
    d = np.random.randint(0, self.ndim)
    t = self.pos[a].copy()
    t[d:] = self.pos[b,d:]
    self.pos[a] = t.copy()
```

To implement crossover, the given particle (index in `a`) is bred with a randomly selected particle residing in the `top` fraction of particles sorted by current objective function values (`idx`). Since breeding a particle with itself is nonsensical and wouldn't change anything, we ensure that `b` is not `a`.

With the two parents selected, we pick a crossover point (`d`) and set up a child (`t`) where values from zero to `d-1` come from the first parent (`a`) and values from `d` to the end of the position vector come from the second parent (`b`). Finally, the first parent is replaced by the new child vector.

This completes our whirlwind tour of the `GA` class. Let's put it to work, learn something about how it behaves when we tweak the CR, F, and η parameters, and compare its performance to that of other swarm algorithms using our test function, Equation 3.1.

6.3 Testing the GA Class

Locating the global minimum of Equation 3.1 is our goal as it has been throughout the book to this point. We'll see how GA does here, too, but we also want to go beyond this simple two-dimensional example. For now, let's see how we do with Equation 3.1. We know the `fxy_gaussian.py` code is already configured for all the algorithms we'll investigate, so we need only run it to use GA with its default parameters. This command line will do the trick,

```
> python3 fxy_gaussian.py 20 100 GA RI
```

We'll use 20 particles, 100 iterations, and random initialization as we did for the other swarm algorithms.

6.3.1 Modifying Population Size and Generations

Naturally, each run produces a unique output. Let's run 21 times and see what sort of performance we get. Doing so gives us 14 successful searches, 5 wrong searches, and 2

failures. Recall, a success is finding the global minimum, or at least being quite near it. A wrong search ended up in the other local minimum, and a failure was near neither minimum. A success rate of 67% isn't much to write home about.

The successful GA searches found an average minimum of -4.5378 ± 0.2271. The true minimum is -5.0, so successful searches were not all that close. The mean number of swarm best updates for successful searches was 4.6 ± 0.5.

What should we make of this lackluster performance? The GA algorithm lands somewhere between RO and the other swarm algorithms we've considered in terms of how much the performance of particles affects the activities of other particles. In RO, all particles go their own way, while for the other swarm algorithms, particles pay close attention to each others' performance. In GA, the driver for improvement is less overt. There is no principle behind how crossover is applied, and mutation is purely random and unrelated to the goal of the swarm, so the only driver is η since η is related to the current performance of the swarm via the objective function value. A weak driver might mean greater flexibility in the long run, but it also means slower movement of the swarm through the search space.

If you run `fxy_gaussian` again using the command line above, but add a final command-line option to output frames and then step through the frames in sequence, you'll see the swarm move in a strange, grid-like, jumping fashion. This is an effect of crossover on a two-dimensional search space. Often, particles don't move at all and then jump. The most significant driver of searching, in this case, is mutation. Compared to the other swarm algorithms, there is no smooth flow through the search space towards the global minimum.

So, we might hypothesize that the dimensionality of the search space matters for GA, and that few dimensions might make it hard for evolution to have much to work with. We'll return to this thought in a bit. We might also think that a population of twenty individuals is too small, so let's see what happens when we increase the population size from 20 to 100. We'll leave the number of iterations (generations) at 100 and tally the results over 21 runs of the search. Running the search gives us 21 successes with an overall minimum value of -4.9523 ± 0.0131 and 6.4 ± 0.6 swarm best updates, on average.

These results are encouraging. Not only were more searches successful, indeed, all were successful, but the final minimum values found were closer to the actual minimum and clustered more tightly around it, as evidenced by a much smaller standard deviation than the twenty particle case. Still, other swarm algorithms found a better minimum value with far fewer particles. For example, with only ten particles and 100 iterations, Jaya found the minimum with virtually no error.

What happens if we return to only 20 particles but search for a longer period of time? For this experiment, let's bump the 100 generations to 1000. In this case, we have 16 successful searches with 5 wrong and no failures. However, when the searches were successful, the average minimum was -4.9947 ± 0.0013, significantly better than the case of only 100 iterations. The average number of swarm best updates was 11.4 ± 1.1, more than we had with 100 iterations, but that shouldn't surprise us with ten times as many generations as before.

Some trends show themselves with this example as we alter the population size and the number of iterations/generations. First, larger populations offer more raw material for the evolutionary process and lead to more consistent performance. However, more generations also play into the process, hence the better solutions found by the smaller population that was allowed to evolve longer. These results beg the question: what happens if we have a larger population and more generations? We know what sort of answer to expect, but let's experiment all the same. We'll run a population of 100 particles for 1000 generations. As

anticipated, we have 21 successful searches with no wrong or failed searches. Our overall minimum value is -4.9919 ± 0.0056 with 9.0 ± 0.7 swarm best updates, on average.

The rule of thumb regarding the GA seems clear: larger populations and more iterations lead to more accurate solutions. There always seems to be a price to pay. It may be some problems are quite difficult for other algorithms to solve because they are too invested in paying attention to moving the swarm rapidly towards a solution, while something like the genetic algorithm and it's less overt control mechanism might be able to find a better solution given enough time and resources. This is not unlike biological evolution, where time is a primary ingredient.

6.3.2 Modifying CR, F, and η

The tests above constrained themselves to modify the population size and number of generations (iterations). Let's look now at changing the crossover probability (CR), mutation probability (F), and the fraction of the population available for breeding the next generation (η). We'll fix the population size and number of generations at 100 each for all of the following tests.

Let's start by varying CR, the crossover probability. Recall, CR sets the likelihood of a particle breeding with one of the top-performing particles. For all tests, the top fraction is 0.5, the default. The lower CR is, the less likely it is that the particle will breed on a swarm update. If CR is zero, no breeding happens, and only mutation alters the swarm. Here, the mutation probability is the default of 0.05. If CR is one, every particle breeds on every iteration.

The code is in `fxy_gaussian_ga_cr.py`. The code uses swarms of 100 particles run for 100 iterations. For each CR value tested, the result reported is the mean and standard error over 100 runs. The display also tells us how many of the searches succeeded in locating the global minimum, how many found the wrong minimum, and how many failed to find either minimum. For example, one run produced,

```
CR=0.00, min= -4.5252360 +/- 0.0455524 (success=100, wrong= 0, fail= 0)
CR=0.02, min= -4.6299699 +/- 0.0338846 (success=100, wrong= 0, fail= 0)
CR=0.04, min= -4.6957286 +/- 0.0339823 (success=100, wrong= 0, fail= 0)
CR=0.06, min= -4.8023189 +/- 0.0224232 (success=100, wrong= 0, fail= 0)
CR=0.08, min= -4.8123653 +/- 0.0208867 (success=100, wrong= 0, fail= 0)
CR=0.10, min= -4.7826065 +/- 0.0272041 (success=100, wrong= 0, fail= 0)
CR=0.12, min= -4.8305728 +/- 0.0202203 (success=100, wrong= 0, fail= 0)
CR=0.14, min= -4.8439194 +/- 0.0277094 (success=100, wrong= 0, fail= 0)
CR=0.16, min= -4.8310437 +/- 0.0272780 (success=100, wrong= 0, fail= 0)
CR=0.18, min= -4.8818124 +/- 0.0192430 (success=100, wrong= 0, fail= 0)
CR=0.20, min= -4.8670177 +/- 0.0276245 (success=100, wrong= 0, fail= 0)
CR=0.30, min= -4.9203102 +/- 0.0179200 (success=100, wrong= 0, fail= 0)
CR=0.40, min= -4.9169339 +/- 0.0179931 (success=100, wrong= 0, fail= 0)
CR=0.50, min= -4.8760211 +/- 0.0282628 (success= 99, wrong= 1, fail= 0)
CR=0.60, min= -4.8892758 +/- 0.0293400 (success=100, wrong= 0, fail= 0)
CR=0.70, min= -4.8435547 +/- 0.0257399 (success=100, wrong= 0, fail= 0)
CR=0.80, min= -4.8375457 +/- 0.0344338 (success= 99, wrong= 1, fail= 0)
CR=0.90, min= -4.8460925 +/- 0.0308933 (success= 99, wrong= 1, fail= 0)
CR=1.00, min= -4.8689705 +/- 0.0264315 (success= 99, wrong= 1, fail= 0)
```

We see that no searches failed, and only a handful found the wrong minimum. Note, the wrong minimum searches show up when CR is about 0.5 or higher. This pattern seems consistent from run to run of the test code. Whether this effect is conditioned on the

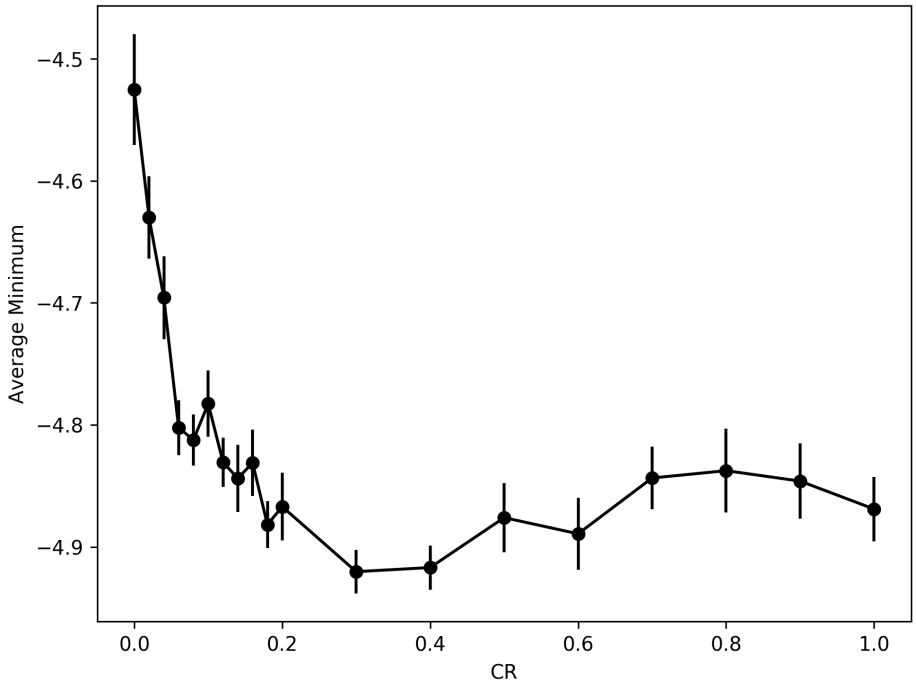


Figure 6.3: Mean minimum found as a function of CR (mean ± SE).

specific problem we're solving or not is unclear.

A plot of this data is in Figure 6.3. For each CR, we plot the mean of all successful searches and the standard error. When CR is zero, only mutation drives the search, and the results are not as good as we would like, though still good enough to consider each search successful. As CR increases, the mean minimum found improves, as we expect, since particles have more opportunity to breed on each iteration. A CR in the range $[0.2, 0.5]$ delivers the best performance. As CR continues to increase, performance decreases slightly up to a CR of one when each particle breeds on each iteration. The error bars' size gives us confidence that the effect we see in the plot is real.

Fixing the NumPy pseudorandom generator seed at 73939133 lets us plot the mean swarm best position and dispersion values as a function of generation for specific crossover probabilities ensuring the two plots represent the same sequence of swarm updates.[1] The code for these plots is also in `fxy_gaussian_ga_cr.py`. Recall, by "dispersion" we mean a measure of how spread out the swarm is in x and y. If the swarm is collapsing, the dispersion goes down.

Figure 6.4 presents the evolution of the swarm best (left) and the dispersion of the swarm (right) as functions of the generation. For the swarm bests, we see distinct differences between different CR values. When CR is zero, mutation only, the swarm best changes slowly compared to other CR values. The other CR values track each other closely, though one might argue that CR=1 does not perform quite as well as a CR of 0.5 or 0.8. Recall, CR=0.8 is the default value for the `GA` class.

Note, because we fixed the pseudorandom seed value, each curve in Figure 6.4 starts at the same initial swarm configuration. However, the first point plotted in the graph is the

[1]The number 73939133 is the largest right-truncatable prime, meaning 73939133, 7393913, 739391, 73939, 7393, 739, 73, and 7 are all prime. We use it here simply for fun.

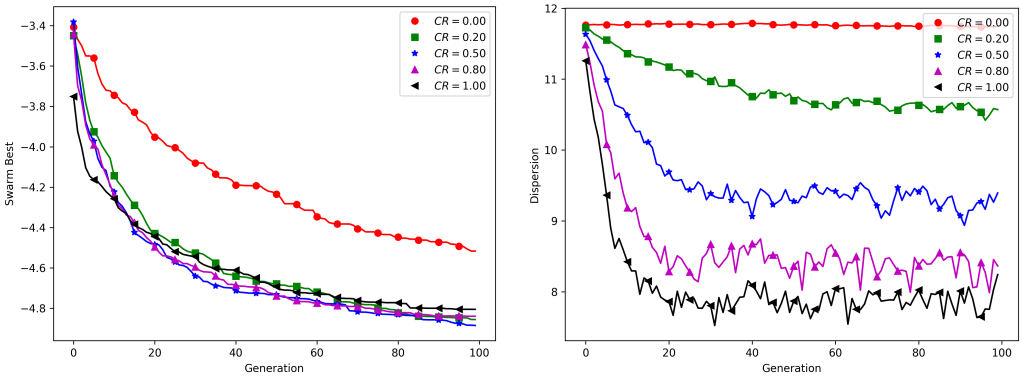


Figure 6.4: (Left) Mean swarm best position and (right) mean swarm dispersion as functions of generation and CR.

swarm *after* one iteration; therefore, the swarm has been updated once and affected by the selected CR value. This accounts for the different starting positions of the curves.

The right side of Figure 6.4 tells the story of the swarm's coverage of the 2D search space. We immediately notice that CR=0, a mutation-only swarm with a probability of mutation set at 5%, barely changes from iteration to iteration. This effect is to be expected. If only some five particles of the 100 in the swarm change at all in any given generation, and only in a single dimension (gene), then we should expect only minor effects on the dispersion of the swarm, doubly so when averaging over 100 searches.

The dispersion for the remaining CR values tracks inversely with increasing CR. Again, this seems a reasonable result. If CR=0.2, then on average, only 20 of the 100 particles breed on any iteration, so particle positions will not change rapidly. However, when CR=0.8 or CR=1.0, most or all particles breed on every iteration forcing the swarm to reach the level of uniformity reflected in Figure 6.4. That these effects are so apparent in a 2D search is somewhat impressive.

Let's turn our attention now to modifying F, the mutation rate. The code for the tests is in `fxy_gaussian_ga_f.py`. We'll make plots similar to those above for changing CR. Our first plot tracks the global minimum as a function of F from $[0, 1]$. If F is zero, there is no mutation, evolution is restricted to crossover only. This approach mixes the existing swarm gene pool, the set of particles from the random initializer, but that's all. As we increase F, the likelihood of a particle mutating during swarm updates increases. When F is one, mutation happens to all particles on every iteration.

Can mutation alone drive the swarm to the global minimum? We can test this if we set CR to zero. Therefore, let's make two curves tracking the mean swarm minimum as a function of F. The first for CR=0.8, our default, and the second for CR=0.0, the mutation-only case. The result is Figure 6.5.

Figure 6.5 tells us when mutation is not present (F=0), the swarm fails to converge on the minimum even though some 90% of the swarm searches are heading towards the global minimum. If F is low, 0.05, the presence of crossover matters and contributes to the effectiveness of the search. Increasing F from its minimum value to about 0.3, combined with the default crossover probability of 0.8, is the most effective combination in this case. As in biological evolution, the most effective approach is sexual reproduction combined with mutation over time. Do bear in mind, however, that our example is 2D. A mutation changes as much of the particle's genome as crossover does, though the probabilities involved are

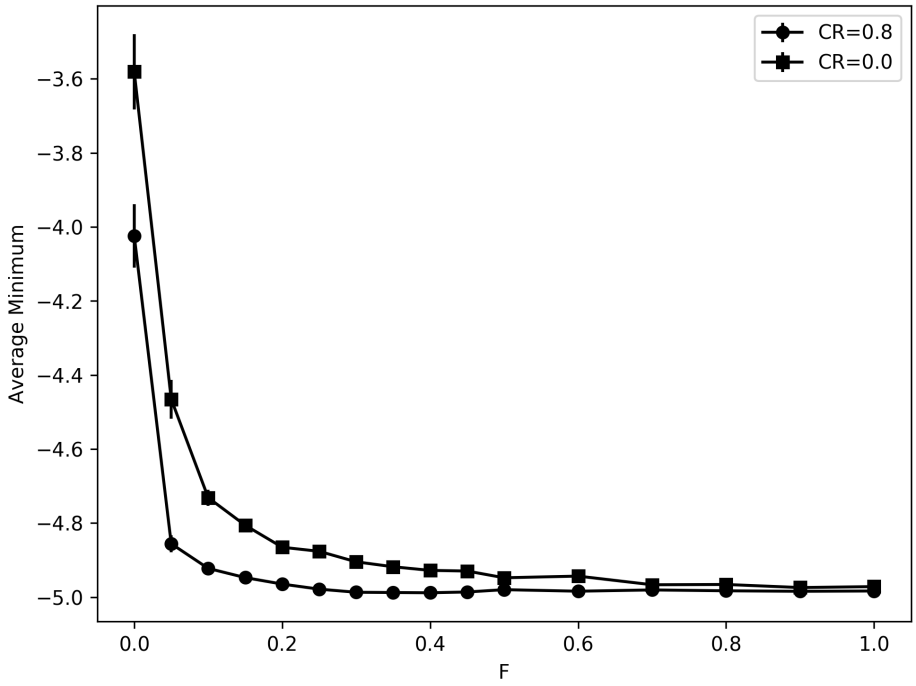


Figure 6.5: Average minimum found as a function of F, the mutation probability.

different.

There is one more GA parameter we can manipulate, η. Let's vary the fraction of best particles available for breeding and see how this affects the rate of convergence of the swarm, the dispersion, and the overall minimum found. The code for the tests is in `fxy_gaussian_ga_eta.py`. Running the code produces figures akin to those we've already seen.

Figure 6.6 tells us that as η increases, two things happen. First, for the fixed number of iterations, the overall minimum found gets worse. Second, the scatter around the mean increases, as evidenced by the larger error bars. For small η, the error bars are virtually invisible, and the minimum found is much closer to the true minimum of -5.0. For the Gaussian function, $\eta = 0.2$ seems quite reasonable.

What of the convergence and dispersion of the swarm as a function of iteration and η? Figure 6.7 shows $\eta = 0.2$ converging quickly while maintaining a reasonable dispersion. When $\eta = 0.99$, meaning virtually all particles are free to breed with any other particle, the dispersion of the swarm remains higher, but the average minimum found is also higher. Results like these for $\eta = 0.2$ and $\eta = 0.99$ make sense. A swarm allowing individuals to breed with only top-performing individuals will have decreased diversity (dispersion), but will also drive the swarm towards better positions in the search space. In the extreme case of $\eta = 0.01$, meaning in essence for our test case, all particles breed with the top performer only, dispersion is at its lowest, but overall performance in terms of minimum found is no worse than most other η values. Surprisingly, however, $\eta = 0.2$ is still better, a likely sweet spot for this particular problem.

The analysis of the Gaussian test problem points us towards a set of parameters we might expect to work well: $\eta = 0.2$, $F = 0.4$, $CR = 0.3$. If we run these parameters 100 times, we get a final minimum value after 100 generations of -4.99421967, which is better

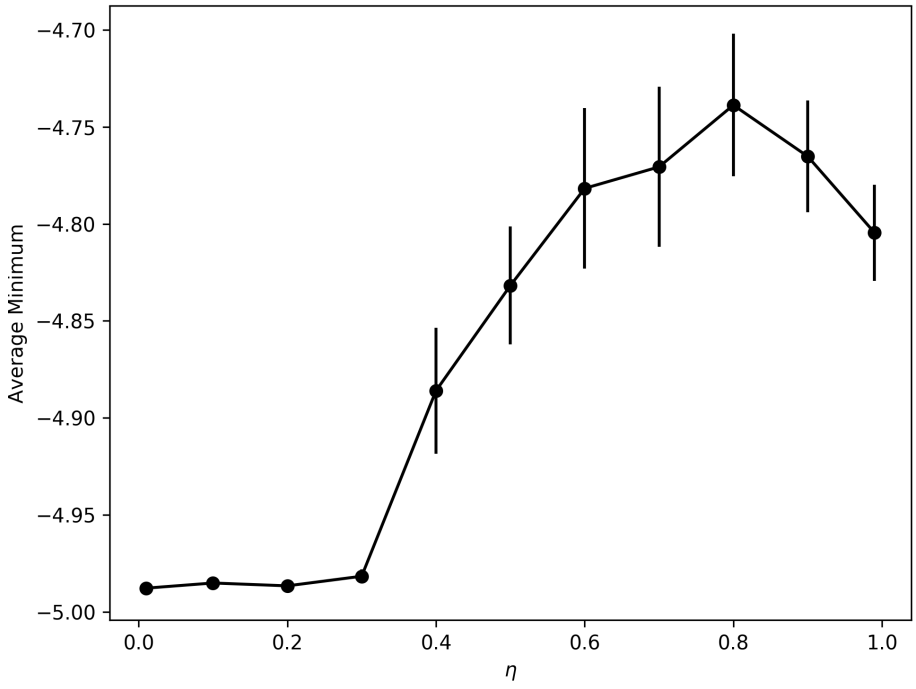


Figure 6.6: Average minimum found as a function of η, the fraction of the population available for breeding.

than any of the previous results in this section. We'll use this set of values for the remainder of the chapter.

6.3.3 Comparison with Other Algorithms

Now that we understand the effect of the GA parameters, let's compare against the other algorithms. We'll track the convergence of the swarm for GA, RO, PSO, Jaya, and GWO simultaneously to see what conclusions we can draw from the results.

The code is in the file `fxy_gaussian_algs.py` and is structurally quite similar to most of the code used above. We'll run 100 searches per algorithm type and plot the mean of the current swarm best objective function value as a function of iteration. The result is Figure 6.8, where we see that PSO converges most rapidly, followed by Jaya and GWO running neck and neck. Next, comes GA followed by RO. The ordering of GA and RO is intuitively sensible. RO has no mechanism for sharing information between particles. For GA, the sharing of information is implicit in crossover when portions of one particle's position are used to update another particle.

6.3.4 Higher-Dimensional Searches

In the second part of the book we'll explore optimizations involving many more dimensions than just the two we've explored so far. However, some readers may be wondering if we're too generous to the GA algorithm by giving it a simple two-dimensional search. After all, in that case, crossover is either copy one dimension from each parent or both from one. Mutation is similarly dramatic as it changes 50% of the particle's genome when it happens. Let's see how GA fares searching a higher-dimensional space. We'll continue with function

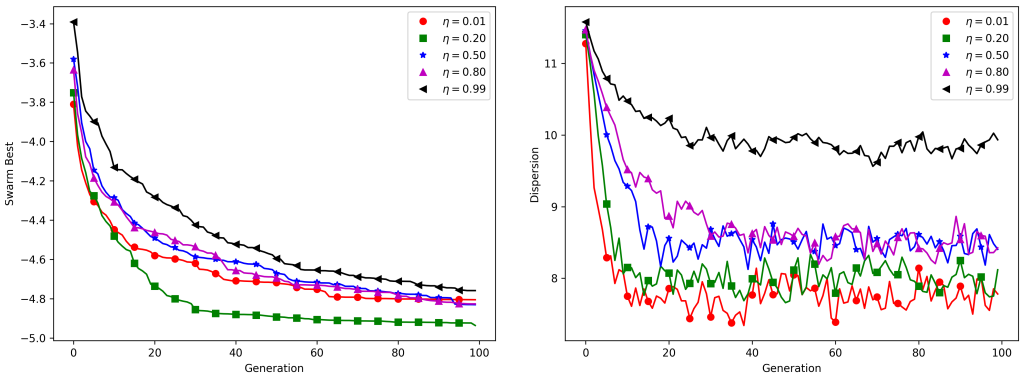


Figure 6.7: (Left) Mean swarm best position and (right) mean swarm dispersion as functions of generation and η.

optimization, with locating the minimum of a Gaussian, but instead of two dimensions, we'll expand to five. Naturally, we cannot plot the function, but the form is a simple extension of Equation 3.1 by adding three more dimensions,

$$f(\boldsymbol{x}) = -5\exp\left(-\frac{1}{2}\left(\frac{(x_0+2.2)^2}{0.4}+\frac{(x_1-4.3)^2}{0.4}+\frac{(x_2+3.1)^2}{0.4}+\frac{(x_3-1.2)^2}{0.4}+\frac{(x_4+0.7)^2}{0.4}\right)\right)+ \tag{6.1}$$

$$-2\exp\left(-\frac{1}{2}\left(\frac{(x_0-2.2)^2}{0.4}+\frac{(x_1+4.3)^2}{0.4}+\frac{(x_2-3.1)^2}{0.4}+\frac{(x_3+1.2)^2}{0.4}+\frac{(x_4-0.7)^2}{0.4}\right)\right) \tag{6.2}$$

This function has a global minimum of -5.0 at $(-2.2, 4.3, -3.1, 1.2, -0.7)$.

The code we need is in `fxy_gaussian_ga_multi.py`. We'll run the search for various swarm sizes and number of iterations. In each case,we'll perform the search 50 times and track the number of times we end up near the global minimum (success), the other minimum (wrong), or neither (failure). For successful searches, we'll report the mean minimum value found. The code takes some time to run and produces output similar to Table 6.1.

The first number on each line is the size of the swarm. The second is the number of iterations. We immediately notice that small populations do not do well. They often fail to converge to the real minimum and even at times fail to converge to either minimum. The frequency of wrong minimums decreases as the swarm size increases, as we might expect. Evolution needs a larger population to work with. We also see that the mean result of a search improves as the number of generations increases for fixed population size. Evolution needs time, as well. However, of practical importance for using the GA approach, we see there's little difference between the minimum found after 1000 generations and that found after 2000, though, as always, it depends on the application as to whether or not the difference is meaningful.

Consider the results for a swarm of 2000 particles. The mean minimum position found improves with the number of iterations, but the relative improvement decreases as the number of iterations increases. There is significant improvement between 100 and 500 iterations, less between 500 and 1000, and so on. High precision results might be difficult to achieve with the genetic algorithm.

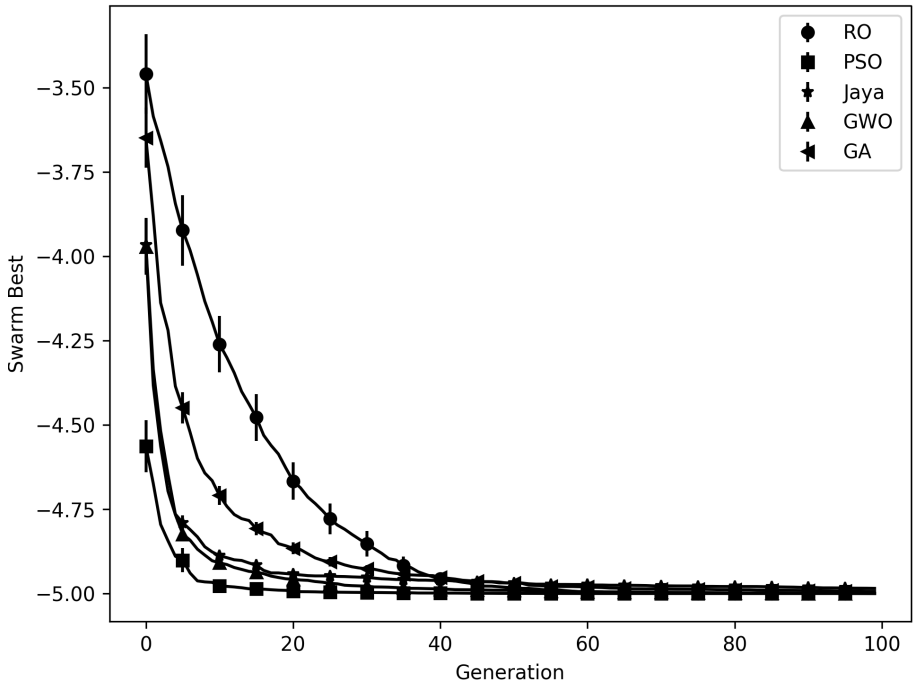


Figure 6.8: Mean swarm best as a function of iteration for each algorithm type.

For comparison purposes, a tweak in the code for `fxy_gaussian_ga_multi.py` replaces the instantiation of a `GA` object with a `PSO` particle swarm using `LinearInertia`. If we rerun the code and wait overnight because our code is not parallelized, we get output like Table 6.2.

Table 6.2, at first glance, looks quite nice. The particle swarm converges more rapidly than the genetic algorithm, and when the number of particles or iterations is high is sometimes perfect for all 50 runs. However, a second look at Table 6.2, especially when compared to the GA results in Table 6.1, reveals a distinct weakness in the PSO search relative to GA.

For GA, as the population increases, the probability of landing in the wrong minimum decreases until we never end up in the wrong place. For the particle swarm, the number of wrong searches also decreases with increasing swarm size, but slowly and erratically, and it never becomes as clean as the GA result. The price to pay for the particle swarm's precision when it finds the correct minimum seems to be an increased likelihood of not finding it at all.

We leave the 5D Gaussian comparison with RO, Jaya, and GWO as an exercise for the reader. Only straightforward changes to the code in `fxy_gaussian_ga_multi.py` are needed. We'll revisit this example in the next chapter and compare the GA and PSO results to those found by differential evolution, to which we now turn.

(particles, iterations)	mean minimum ($n = 50$)	(success, wrong, fail)
(20, 100)	-3.1994460± 0.1715789	(22, 22, 6)
(20, 500)	-4.8024402± 0.0317703	(31, 19, 0)
(20, 1000)	-4.9401292± 0.0243199	(23, 27, 0)
(20, 1500)	-4.9778330± 0.0037183	(39, 11, 0)
(20, 2000)	-4.9882480± 0.0016098	(30, 20, 0)
(100, 100)	-3.7911285± 0.1136029	(32, 18, 0)
(100, 500)	-4.8939450± 0.0136274	(39, 11, 0)
(100, 1000)	-4.9590089± 0.0080581	(31, 19, 0)
(100, 1500)	-4.9779871± 0.0050738	(37, 13, 0)
(100, 2000)	-4.9747370± 0.0113306	(38, 12, 0)
(500, 100)	-4.2072364± 0.0707933	(48, 2, 0)
(500, 500)	-4.8126808± 0.0361144	(49, 1, 0)
(500, 1000)	-4.9181503± 0.0278864	(45, 5, 0)
(500, 1500)	-4.9524024± 0.0092210	(48, 2, 0)
(500, 2000)	-4.9647868± 0.0086705	(48, 2, 0)
(1000, 100)	-4.1254405± 0.0936570	(48, 2, 0)
(1000, 500)	-4.7889888± 0.0359093	(50, 0, 0)
(1000, 1000)	-4.9088754± 0.0213418	(50, 0, 0)
(1000, 1500)	-4.9598445± 0.0051796	(50, 0, 0)
(1000, 2000)	-4.9622745± 0.0076301	(50, 0, 0)
(2000, 100)	-4.2631505± 0.0733849	(50, 0, 0)
(2000, 500)	-4.8383683± 0.0293259	(50, 0, 0)
(2000, 1000)	-4.9283615± 0.0094757	(50, 0, 0)
(2000, 1500)	-4.9603370± 0.0057764	(50, 0, 0)
(2000, 2000)	-4.9764850± 0.0026534	(50, 0, 0)

Table 6.1: Mean minimum found for GA and the 5D Gaussian. Successful searches only.

(particles, iterations)	mean minimum ($n = 50$)	(success, wrong, fail)
(20, 100)	-4.8427106± 0.0504383	(33, 17, 0)
(20, 500)	-4.9999178± 0.0000703	(23, 27, 0)
(20, 1000)	-4.9994031± 0.0005821	(30, 20, 0)
(20, 1500)	-4.9783888± 0.0213171	(37, 13, 0)
(20, 2000)	-4.9532370± 0.0459497	(29, 21, 0)
(100, 100)	-4.9998833± 0.0000556	(31, 19, 0)
(100, 500)	-5.0000000± 0.0000000	(32, 18, 0)
(100, 1000)	-4.9592468± 0.0400682	(30, 20, 0)
(100, 1500)	-4.9471347± 0.0521650	(38, 12, 0)
(100, 2000)	-4.9999997± 0.0000003	(35, 15, 0)
(500, 100)	-4.9999966± 0.0000024	(37, 13, 0)
(500, 500)	-5.0000000± 0.0000000	(35, 15, 0)
(500, 1000)	-4.9998467± 0.0001514	(41, 9, 0)
(500, 1500)	-5.0000000± 0.0000000	(42, 8, 0)
(500, 2000)	-5.0000000± 0.0000000	(44, 6, 0)
(1000, 100)	-4.9999999± 0.0000000	(35, 15, 0)
(1000, 500)	-4.9999994± 0.0000006	(37, 13, 0)
(1000, 1000)	-4.9522592± 0.0471951	(44, 6, 0)
(1000, 1500)	-5.0000000± 0.0000000	(49, 1, 0)
(1000, 2000)	-5.0000000± 0.0000000	(47, 3, 0)
(2000, 100)	-5.0000000± 0.0000000	(34, 16, 0)
(2000, 500)	-4.9897808± 0.0100938	(41, 9, 0)
(2000, 1000)	-5.0000000± 0.0000000	(48, 2, 0)
(2000, 1500)	-5.0000000± 0.0000000	(50, 0, 0)
(2000, 2000)	-5.0000000± 0.0000000	(50, 0, 0)

Table 6.2: Mean minimum found for PSO and the 5D Gaussian. Successful searches only.

Chapter 7

Differential Evolution

Differential Evolution (DE) ([20]) is the second evolutionary algorithm we'll sneak into our toolkit. It's a good one to have and to consult frequently. DE is of the same vintage as PSO, the mid-1990s, and both have proved themselves time and again. Our toolkit would be incomplete without it.

This chapter follows the format of the previous chapters. We'll present differential evolution in Section 7.1, and discuss the code in Section 7.2. As before, we'll put DE through its paces in Section 7.3, with both our simple Gaussian example and the 5D version introduced in Section 6.3. DE is the last of our swarm algorithms. With it, our toolkit is complete, and we move to the second part of the book – the experiments.

7.1 Unnatural Mutation

We discussed GA in Chapter 6 and saw that it worked by simplifying evolution, reducing it to just crossover (breeding) and random mutation. This approach made sense. For DE, we continue with the evolution metaphor but alter it slightly. As we will see, changing it slightly has a significant impact on its effectiveness for many problems.

In GA, mutation affects a single dimension of the particle's position. A random element is updated. Crossover involves breeding two individuals, the current one and some member of the swarm, possibly restricted to a top-performing member of the swarm. A crossover point is selected, and all values up to that point for the current particle are kept with the remainder coming from the selected breeding partner.

In DE, mutation and crossover still apply, but the step where each is performed is named differently from what we might expect. In [20], mutation refers to the generation of a position in the search space built from *three* donor vectors, three other particles. Specifically, mutation finds $\boldsymbol{v}$ from three other swarm vectors, excluding the current swarm vector, using,

$$\boldsymbol{v} = \boldsymbol{v}_1 + F(\boldsymbol{v}_2 - \boldsymbol{v}_3) \tag{7.1}$$

Here $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, and $\boldsymbol{v}_3$ are the three donors to $\boldsymbol{v}$ and F, typically $[0, 2]$, is the amplification factor on $\boldsymbol{v}_2 - \boldsymbol{v}_3$, the differential variation, an offset to $\boldsymbol{v}_1$. Using the difference leads to the algorithm name, "differential evolution". Equation 7.1 creates the mutation vector, but not the final candidate. While [20] uses "mutation" in reference to Equation 7.1, it is, in a sense, breeding, a mixing of three individuals instead of a random error applied during the creation of a new individual. We might consider Equation 7.1 to be unnatural mutation

– unnatural because, in nature, mutation does not involve another individual, let alone three of them. Rather, Equation 7.1 is similar to Equation 5.1, where some portion of the difference between existing particles is used to update the position of another. For Jaya, the difference is explicit in that the swarm's best and worst positions are used. For DE, the selection is random (usually, we'll see a variant below).

After mutation creates $\boldsymbol{v}$, we apply crossover. However, unlike crossover in GA where a single break point is selected and two entire portions of the position vector are merged, DE selects element by element. The goal is to generate the candidate vector, $\boldsymbol{u}$, where the elements of $\boldsymbol{u}$ are selected one at a time using,

$$\boldsymbol{u}_i = \begin{cases} \boldsymbol{v}_i, & \text{if } r < CR \text{ or } i = I \\ \boldsymbol{x}_i, & \text{otherwise} \end{cases} \tag{7.2}$$

In Equation 7.2, i is the current index, CR is the crossover probability, r is a random number $[0, 1]$, $\boldsymbol{x}$ is the current particle position, and I is a randomly selected index $[0, n_{\text{dim}})$. The purpose of I is to ensure that at least one element of $\boldsymbol{u}$ comes from $\boldsymbol{v}$, the mutation vector.

Algorithmically, DE follows Algorithm 4 in Chapter 3, but replaces,

Select a new position some random distance away from the current position

with,

Apply DE mutation and crossover to generate a new position

where the candidate is derived from both Equation 7.1 and Equation 7.2.

The mere fact that we have a *candidate* vector ($\boldsymbol{u}$) distinguishes DE from GA and makes DE more like RO. In GA, evolution happens, period. If the result is helpful, great. If it isn't, we're stuck with it. There is a weak overlord, the overlord who decides who gets to breed, but the direct involvement of members of the swarm, beyond the single mate, is missing. In DE, the mutation vector, the actual mate of the current vector, is built from the swarm using entire vectors or differences of vectors. Additionally, crossover injects more diversity because it acts element-by-element instead of two disjoint segments merged. As we'll see below, the mutation rule, and highly-mixed crossover, make DE a powerful technique.

Equation 7.1 uses three randomly selected vectors to build the mutation vector: $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, and $\boldsymbol{v}_3$. DE is highly customizable, like most metaheuristic algorithms, and many variants have been explored leading to a nomenclature for specifying which version is being discussed. The version in Equation 7.1 is known as,

DE/rand/1/bin

Let's decode the designation. Beyond the obvious meaning of "DE", the first field, "rand", means that $\boldsymbol{v}_1$ is a randomly selected swarm position vector. The "1" refers to the number of differentials used in creating the mutation vector. Equation 7.1 has only one, $\boldsymbol{v}_2 - \boldsymbol{v}_3$. Finally, "bin" refers to how crossover is implemented. Equation 7.2 selects elements of the final candidate vector, $\boldsymbol{u}$, by performing individual Bernoulli experiments, essentially flipping a weighted coin. Recalling that a Bernoulli distribution is a special case of a binomial distribution explains the "bin" label.

Our implementation supports three versions of the first field and two versions of the second. Beyond using a randomly selected $\boldsymbol{v}_1$, we support "best" to use the current swarm

best position as $\boldsymbol{v}_1$ and, new for this text as far as can be discerned, "toggle," which on each swarm update toggles between "rand" or "best". Therefore, temporarily ignoring "toggle", we can configure the search to use,

DE/rand/1/bin	random $\boldsymbol{v}_1$, coin flipping crossover
DE/best/1/bin	swarm best $\boldsymbol{v}_1$, coin flipping
DE/rand/1/GA	random $\boldsymbol{v}_1$, GA-style crossover
DE/best/1/GA	swarm best $\boldsymbol{v}_1$, GA crossover

We'll experiment with each of these options, and "toggle", in Section 7.3.

Our DE implementation will not support anything other than a single differential, but researchers have experimented with additional differentials. For example, DE/rand/2/bin uses two differentials and changes Equation 7.1 to,

$$\boldsymbol{v} = \boldsymbol{v}_1 + F(\boldsymbol{v}_2 + \boldsymbol{v}_3 - \boldsymbol{v}_4 - \boldsymbol{v}_5)$$

with all $\boldsymbol{v}_i$ vectors randomly selected swarm particles.

The original DE paper ([20]) has been referenced over 27,000 times as of 2021. This alone is clear evidence of DE's utility. Is it the best technique? Clearly, no, it isn't as we know there is no "best" optimization algorithm for all cases [21].

DE's strengths include rapid convergence. We'll see this in the experiments that follow. However, later in the book, we'll also see rapid convergence is sometimes a liability. The price we pay for rapid convergence is a tendency to become trapped in local minima. There are also claims of DE not scaling well as the dimensionality of the search space increases. Such claims must be interpreted in the context that led to them; there will doubtless be practical situations where DE scales appropriately for the problem.

7.1.1 Configuring DE

DE is not parameter-free. We need to specify both F and CR (for the "bin" variant). What values should we use? As expected when discussing swarm optimization, the answer is "it depends on the problem." Explorations in [22] imply for a given population size and CR that F should not be less than,

$$F_{\text{crit}} = \sqrt{\frac{1 - \frac{CR}{2}}{n_{\text{particles}}}}$$

Other researchers find different thresholds. For example, in [20] we get $F \in [0.5, 1]$ and $CR \in [0.8, 1]$. Our implementation defaults to $F = 0.8$ and $CR = 0.5$, values recommended in [23]. We are in the general ballpark, and we'll experiment with F and CR in Section 7.3.

For now, let's detail the essential features of the `DE` class and then jump into basic experiments.

7.2 The DE Class

The `DE` class is outlined in Figure 7.1. The common set of methods are present. We'll walk through the `Candidate`, `CandidatePositions`, and `Step` methods. The others are identical or virtually identical to those in the other swarm classes. The source code for the `DE` class is in the file `DE.py`.

```
class DE:
    def __init__(self, ...):
    def Results(self):
    def Initialize(self):
    def Done(self):
    def Evaluate(self, pos):
    def Candidate(self, idx):
    def CandidatePositions(self):
    def Step(self):
    def Optimize(self):
```

Figure 7.1: Outline of the DE class.

The `DE` constructor accepts the same set of arguments as the `GA` class constructor, including `CR` and `F`. We'll discuss those when we use them, though they have much the same meaning for DE as they do for GA. Additionally, the constructor accepts `mode` and `cmode` arguments. The `mode` is a string, either "rand" (default), "best" or "toggle". The mode determines which variant of DE we'll be using to select $\boldsymbol{v}_1$. We'll get to the mode in Section 7.2.3.

The `cmode` parameter specifies the version of crossover used. Allowed values are "bin" (default) or "GA". The default follows the original DE algorithm. It creates the candidate vector by merging the current position vector and mutation vector element-by-element, rolling a weighted die to decide which element to copy. The "GA" option mimics crossover in the `GA` class. Here a single position is selected, and all values up to that position are retained from the current position vector and all values after that position come from the mutation vector.

Let's walk through the relevant methods starting with `Step` and working our way up to `Candidate`. The DE-specific code resides mainly in the `Candidate` method, so that's where we'll spend most of our energy.

7.2.1 Step

The `Step` method performs a single DE swarm update. The code is straightforward and modified only slightly from that of the other swarm algorithms,

```
def Step(self):
    new_pos = self.CandidatePositions()
    p = self.Evaluate(new_pos)
    for i in range(self.npart):
        if (p[i] < self.vpos[i]):
            self.vpos[i] = p[i]
            self.pos[i] = new_pos[i]
        if (p[i] < self.gbest[-1]):
            self.gbest.append(p[i])
            self.gpos.append(new_pos[i].copy())
            self.gidx.append(i)
            self.giter.append(self.iterations)
    self.iterations += 1
```

First, new candidate positions are created for each particle in the swarm (`new_pos`). We do this with a call to `CandidatePositions` (Section 7.2.2). As with RO and GA, the

candidate positions are evaluated (`Evaluate`) with the objective function values retained in `p`.

The loop over particles examines the candidate's objective function value. If the new position's objective function value is less than the current position's (`vpos[i]`), we move to the new position. If not, the particle remains where it is until the next iteration.

We also ask if the new position is a new swarm best. If it is, we make it the new swarm best by appending it to the list of swarm bests (`gbest`, `gpos`). We then increment the iteration counter and start the next iteration of the swarm.

7.2.2 CandidatePositions

This simple method generates new candidate positions for the entire swarm. In code,

```
def CandidatePositions(self):
    pos = np.zeros((self.npart, self.ndim))
    for i in range(self.npart):
        pos[i] = self.Candidate(i)
    if (self.bounds != None):
        pos = self.bounds.Limits(pos)
    return pos
```

where we set up space for the new swarm positions (`pos`) and then loop over the particles to generate a candidate position (`Candidate`) for each. After the update, if there is a `Bounds` object, we enforce boundary and other conditions. The entire set of new positions is returned to `Step`. This method echoes the `CandidatePositions` method of the `RO` class and serves the same function. See Section 3.2.5.

7.2.3 Candidate

The `Candidate` method implements DE mutation and crossover to return a candidate position. If the candidate position is better than the current particle position, the candidate position is kept. In code we have,

```
def Candidate(self, idx):
    k = np.argsort(np.random.random(self.npart))
    while (idx in k[:3]):
        k = np.argsort(np.random.random(self.npart))
    v1 = self.pos[k[0]]
    v2 = self.pos[k[1]]
    v3 = self.pos[k[2]]
    if (self.mode == "best"):
        v1 = self.gpos[-1]
    elif (self.mode == "toggle"):
        if (self.tmode):
            self.tmode = False
            v1 = self.gpos[-1]
        else:
            self.tmode = True
    v = v1 + self.F*(v2 - v3)
    u = np.zeros(self.ndim)
    I = np.random.randint(0, self.ndim-1)
    if (self.cmode == "bin"):
        for j in range(self.ndim):
            if (np.random.random() <= self.CR) or (j == I):
                u[j] = v[j]
```

```
            elif (j != I):
                u[j] = self.pos[idx,j]
    else:
        u = self.pos[idx].copy()
        u[I:] = v[I:]
    return u
```

where `idx` is the index of the current swarm particle. The goal of mutation in DE is to select three other particles which do not contain `idx` and use them to build a donor vector (`v`). The three particles are in `v1`, `v2`, and `v3`. We get them by generating a random ordering of the particles (`k`) and ensuring that none of the first three selected contain the current particle. If the `mode` of the `DE` class is "best", the first particle selected is replaced by the current swarm best position. If the `mode` is "toggle", each call to `Candidate` toggles between "rand" and "best" based on the state of `self.tmode` which is initialized in `__init__` to `False`.

A direct implementation of Equation 7.1 uses the three donor vectors to generate `v`: `v1` plus the mutation fraction, F, times the difference between the other two donors.

With `v`, it is now possible to create the candidate vector, `u`. If `cmode` is "bin", the elements of `u` are set according to Equation 7.2 with `I` a fixed index. The `if` statement checks CR and `I` ensuring at least one element of `u` comes from `v`. Otherwise, the new element comes from the corresponding element in the current position the particle.

If `cmode` is "GA", GA-style crossover is used instead. In that case, `I` is the position where the split and merge takes place with current position values from index zero to $I-1$ retained and values from I onward replaced with the corresponding portion of $\boldsymbol{v}$.

The essence of DE is in the `Candidate` method, the remainder of the `DE` class is nearly identical to the other swarm algorithms we've encountered. Let's put DE to the test, first against our 2D Gaussian and then the 5D Gaussian from Section 6.3.

7.3 Testing the DE Class

To put the DE class through its paces, we'll first do basic runs of the 2D Gaussian we've used consistently throughout the preceding chapters (Equation 3.1). Next, we'll snapshot how the swarm behaves during one of these searches by plotting the swarm positions in 2D space. After that, we alter DE parameters to see the effects on convergence and swarm diversity thereby echoing Section 6.3.2 where we did the same for GA. Finally, we compare DE to all the previous algorithms using our 5D Gaussian (Equation 6.2).

7.3.1 Experiments with a 2D Gaussian

Previously, we started our experiments using a new algorithm by running a search for the minimum of Equation 3.1. Let's continue that tradition by running 21 searches each with our four DE variants and tracking the number of successful searches, wrong searches, and failures. The code we want is in `de_gaussian.py` and run with command lines like this,

```
> python3 de_gaussian.py 20 100 RI rand bin
```

This command runs a search using 20 particles, 100 iterations, random initialization, and DE/rand/1/bin. Configuring a script for each DE variant and 21 searches produces the following,

DE/rand/1/bin	success: 21, wrong: 0, fail: 0
DE/rand/1/GA	success: 21, wrong: 0, fail: 0
DE/best/1/bin	success: 20, wrong: 1, fail: 0
DE/best/1/GA	success: 17, wrong: 3, fail: 0
DE/toggle/1/bin	success: 21, wrong: 0, fail: 0
DE/toggle/1/GA	success: 21, wrong: 0, fail: 0

Clearly, DE performs well on this simple search. In particular, DE/rand/ and DE/toggle/ succeed on all 21 trials. Using DE/best/ produces a few wrong results, worse when using DE/best/1/GA. The increased number of wrong results when using GA crossover is not too surprising; there is less diversity generated by GA-style crossover than the coin-flipping approach. Add in the restriction of using only the current swarm best for $\boldsymbol{v}_1$ when creating the mutation vector, and diversity decreases still further. Of course, recall that these results are for a 2D search space.

Altering the command line to,

```
> python3 de_gaussian.py 20 100 RI rand bin frames
```

creates a `frames` output directory with the current swarm state after each iteration. We'll fix the pseudorandom seed to 8675309 so you can run the code yourself to produce the same output. Setting the seed also lets us compare the plots to previous chapters (e.g., Chapter 4).

A few of the 100 frames in the `frames` directory are shown in Figure 7.2. Running clockwise from the upper left, we have the initial swarm configuration followed by iteration 3, iteration 18, and iteration 40. The known global best position is marked with an open square. The current swarm best position is a star. Compare these frames, especially the iteration numbers, with those of Figure 4.2 using PSO.

The DE algorithm located the global minimum in only a handful of iterations. As the swarm continued to evolve, it split into two groups (iteration 18 in Figure 7.2), one around the global minimum and the other around the other minimum. Further updates to the swarm result in it collapsing on the global minimum by iteration 40. By way of contrast, after 100 iterations, the PSO swarm was still widely dispersed.

The `frames` directory contains an additional file, `dispersion.npy`. This file tracks the dispersion, the mean difference between the minimum and maximum value along each dimension of the swarm, as a function of iteration. Let's plot this to see how the swarm evolved.

Figure 7.3 shows the dispersion of the swarm for DE variants including DE/toggle/1/bin. In each case, there is a rapid drop in dispersion marking when the swarm starts to collapse into the global minimum. Recall, the pseudorandom seed is fixed, so each swarm starts from the same configuration. The timing of the swarm collapse depends on the $\boldsymbol{v}_1$ mode.

As we might expect, "best" leads to a rapid collapse of the swarm once the swarm best locates the global minimum. This is true for either "bin" or "GA" crossover. Pulling the swarm together around the current swarm best should lead to rapid convergence. Therefore, the "best" curves make sense, exploitation of the minimum takes priority.

The other extreme is found for the "rand" searches. These, regardless of crossover strategy, allow the swarm to explore for a longer period before collapsing. Again, the stochastic nature of $\boldsymbol{v}_1$ selection implies this behavior, so the curves are sensible.

Finally, and somewhat satisfyingly, the "toggle" curve lies between "best" and "rand". The swarm is drawn towards the swarm best but not so rapidly that it cannot continue its

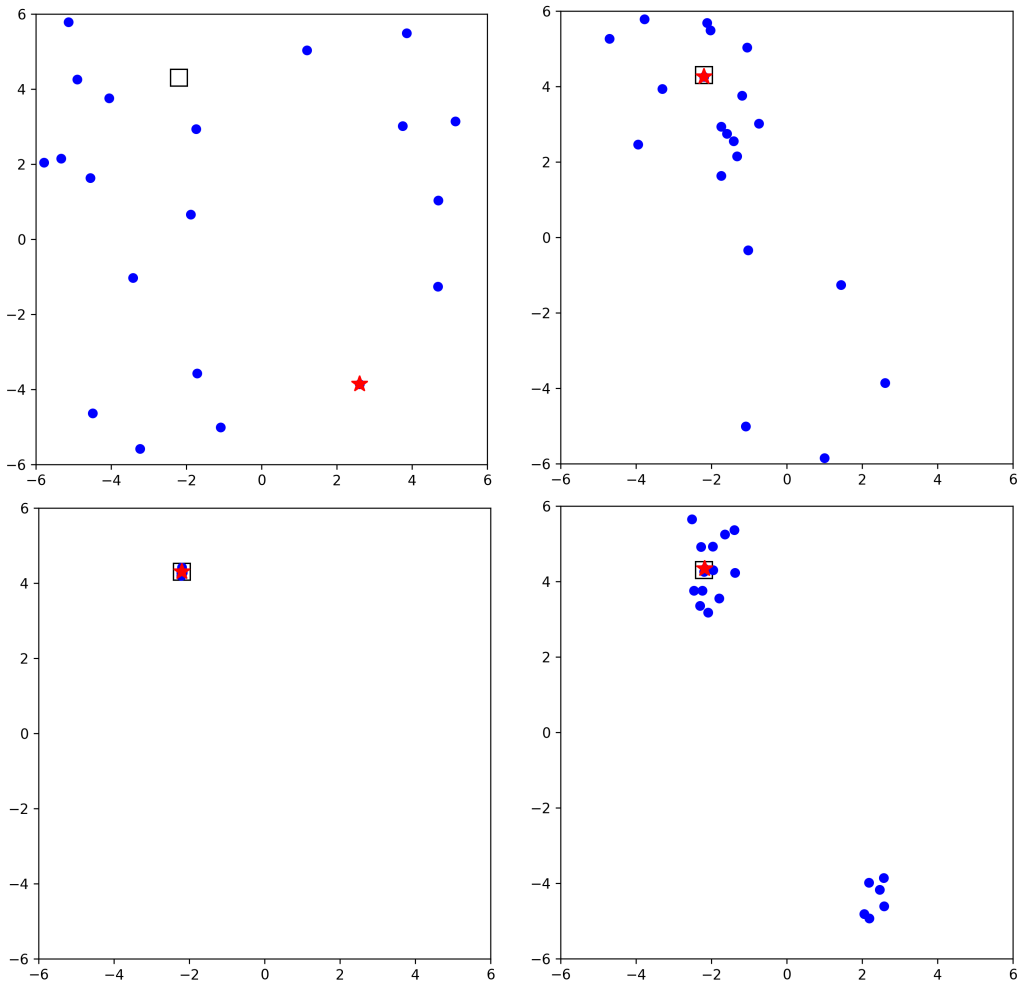


Figure 7.2: DE swarm positions clockwise from upper left: initial, iteration 3, iteration 18, and iteration 40.

exploration of the search space. The reader is encouraged to run `de_gaussian.py` with the fixed pseudorandom seed to produce frames for DE/rand/1/bin, DE/best/1/bin, and DE/toggle/1/bin modes. Stepping through the frames for the first mode shows exploration with a small portion of the swarm spending many iterations around the other minimum. For the second, the swarm converges rapidly on the global minimum, with no exploration around the other minimum. Finally, toggle shows convergence to the global minimum with a small portion of the swarm initially exploring the other minimum, precisely as expected.

7.3.2 Modifying CR and F

In Section 6.3.2, we modified CR and F for the genetic algorithm. Let's do the same here for DE. We'll restrict ourselves to DE/rand/1/bin and leave other combinations as experiments for the reader.

Recall, changing CR for GA implies changing the probability that a particle will breed at all during a swarm update. For DE/rand/1/bin, changing CR alters the likelihood that a component from the mutation vector will be used in the candidate vector. Therefore, the experiment that follows is not strictly a direct comparison, but the CR parameter plays a

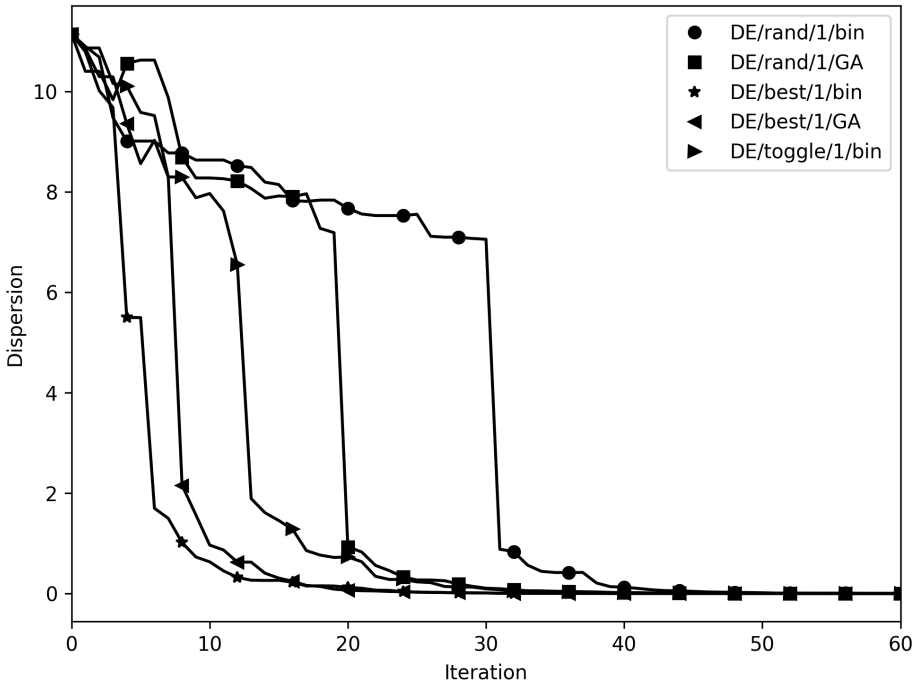


Figure 7.3: Dispersion of different DE searches as a function of iteration.

similar role in each algorithm.

The code we need is in the file `de_gaussian_cr.py`. It's a clone of the code used in Section 6.3.2 replacing the call to `GA` with `DE` and altering the names of the output files. In a sense, the code has evolved by a simple mutation of its "DNA", its source. The code generates the mean minimum value found as a function of CR. The swarm has 100 particles and uses 100 iterations.

Running the code produces Figure 7.4, which we should compare to Figure 6.3 showing the results for GA. Note, there is a significant y-axis scale difference between the two figures.

With GA, we found the search sensitive to CR; it varied more from run to run, as illustrated by the size of the error bars. For DE, when CR is any value above a 5% chance of copying components from the mutation vector, we see virtually uniform performance of the search.

When CR is zero, the candidate vector contains only one value from the mutation vector (see Section 7.2.3). For a 2D search, this one value matters, so even then, the search progresses. When CR reaches one, the mutation vector becomes the candidate vector, and the swarm improves via mutation only, there is no crossover.

Figure 6.4 shows the effect of CR on GA as a function of generation (iteration). Figure 7.5 shows the same for DE. DE is, for this particular search, less affected by CR than GA. The left side of Figure 7.5 testifies to this with even $CR = 0.0$ leading to a good minimum value. In that case, the requirement that at least one component of the mutation vector is used in the candidate vector has had a meaningful impact. We might expect this impact to be significantly reduced in a higher-dimensional search.

The right side of Figure 7.5 shows the effect of CR on the dispersion of the swarm. The results make intuitive sense. A DE search with $CR = 0.0$, especially if the dimensionality of the search space is low, is essentially a mutation-only search with GA: the candidate vector

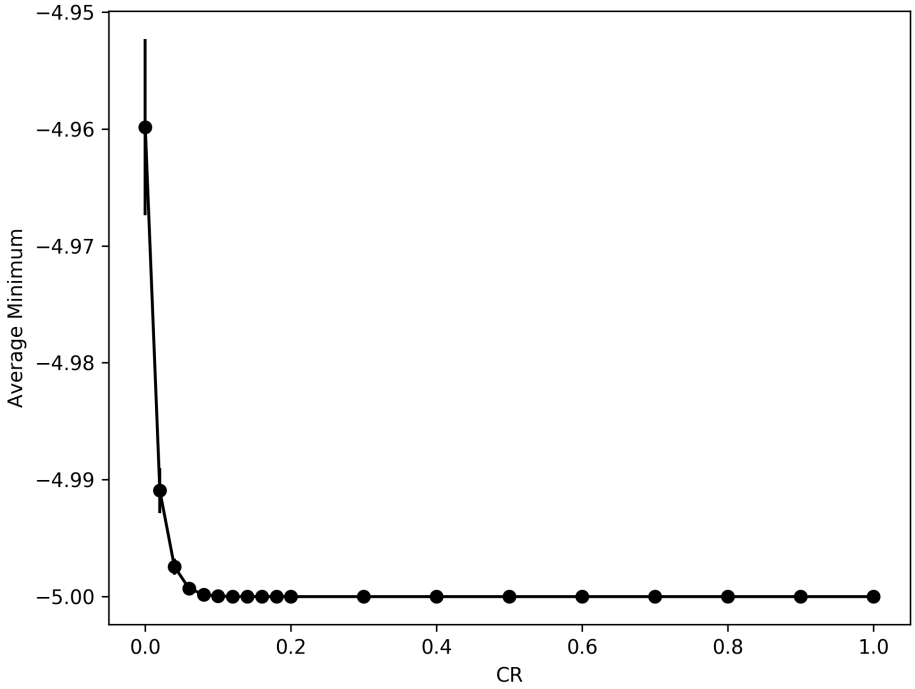


Figure 7.4: Mean minimum found as a function of CR (mean ± SE).

becomes the original position vector with a single (substantially) random mutation. We say "substantially" because the component of the mutation vector used when $CR = 0.0$ is not purely random, it was arrived at in a principled way using existing swarm positions. Still, the variation is small on each swarm update, so we should expect the dispersion to remain high as a function of iteration.

Dispersion falls off more rapidly as CR increases. For the 2D problem we're considering, we expect a rapid decrease in dispersion when the swarm starts to collapse on itself in the well of the global minimum. We saw this effect clearly in Figure 7.3. In Figure 7.5, the collapse into the global minimum is still evident, but smoothed because we are using a swarm of 100 particles instead of only 20. Higher CR values imply a strong tendency for the swarm to pull itself towards the global minimum as diversity goes down. The balance of a DE search between exploration and exploitation can be influenced by CR. One might imagine a more advanced DE search which modifies CR as the search progresses. The situation is not quite so simple, however, as there is a complex interaction between CR and F, and the "ideal" setting for both is highly problem-dependent.

Running the code in `de_gaussian_f.py` produces a plot of the search for two different CR values, Figure 7.6. We immediately notice that when CR is nonzero, we get a good solution for any value of $F \neq 0.0$. Likewise, when $CR = 0.0$, we are bounded and never reach the same level of precision as the $CR = 0.8$ case.

What about the situation when $F = 0.0$? In that case, when $CR = 0.8$ we get *worse* performance than with $CR = 0.0$. Recalling Equation 7.1,

$$\boldsymbol{v} = \boldsymbol{v}_1 + F(\boldsymbol{v}_2 - \boldsymbol{v}_3)$$

we see that $F = 0.0$ implies the mutation vector is only $\boldsymbol{v}_1$ and $\boldsymbol{v}_1$ is a randomly selected swarm vector. Therefore, the mutation vector is a copy of an existing swarm position.

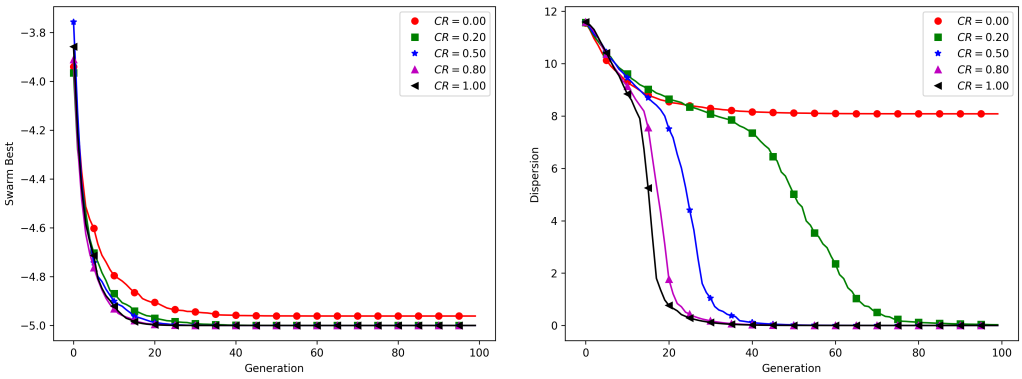


Figure 7.5: (Left) Mean swarm best position and (right) mean swarm dispersion as functions of generation and CR.

Also, when $CR = 0.8$, there is an 80% chance of selecting an element of the candidate vector from the mutation vector. At least one of the mutation vector's two components will be selected with certainty. So, when $F = 0.0$ and $CR = 0.8$, the candidate vectors, with high probability, will be nothing more than an existing swarm particle position. In this case, we shouldn't be surprised to get worse performance. The latter case implies the candidate vectors are exactly one element from $\boldsymbol{v}_1$ and the other from the particle's current position. We have mutation rather than the copying of another swarm vector, so we have more opportunities to create a candidate vector that improves the overall swarm best.

We've likely extracted all the relevant insights we can regarding DE and the simple 2D Gaussian problem. Let's compare DE and all the previous algorithms on the 5D Gaussian example.

7.3.3 Comparing DE to Other Algorithms

Equation 6.2 represents a pair of Gaussians in five dimensions with a global minimum value of -5.0 at $(-2.2, 4.3, -3.1, 1.2, -0.7)$. In Section 6.3.4, we compared the performance of GA and PSO on this function for various combinations of swarm size and iterations. For example, see Tables 6.1 and 6.2.

The test here is straightforward: we desire to plot the mean swarm best value for successful minimization searches as a function of iteration and swarm size. The code we need is in `de_gaussian_5d_plot.py` producing the results in Figure 7.7.

How should we interpret Figure 7.7? The figure shows the mean convergence of the swarms by iteration and algorithm. Here 20 runs were performed and the per iteration mean and SE are shown. Starting with the upper left plot, and going by rows, swarm sizes are 5, 10, 20, 50, 100, and 200 particles. It is important to remember that the plots show only *successful* searches, searches that found or moved towards the global minimum. We'll discuss below how often this happened for each algorithm. Searches continued until 20 were successful. All algorithms used their default parameters, and PSO used the default linear inertia class.

Let's consider RO first. As we've come to expect from RO, even small swarms show a steady movement towards the goal. By the time the swarm contains 50 particles, we are close to reaching the global minimum by iteration 450, the limit used for the plots. As always, "close" is relative, and within 20% might be fine for one application while another

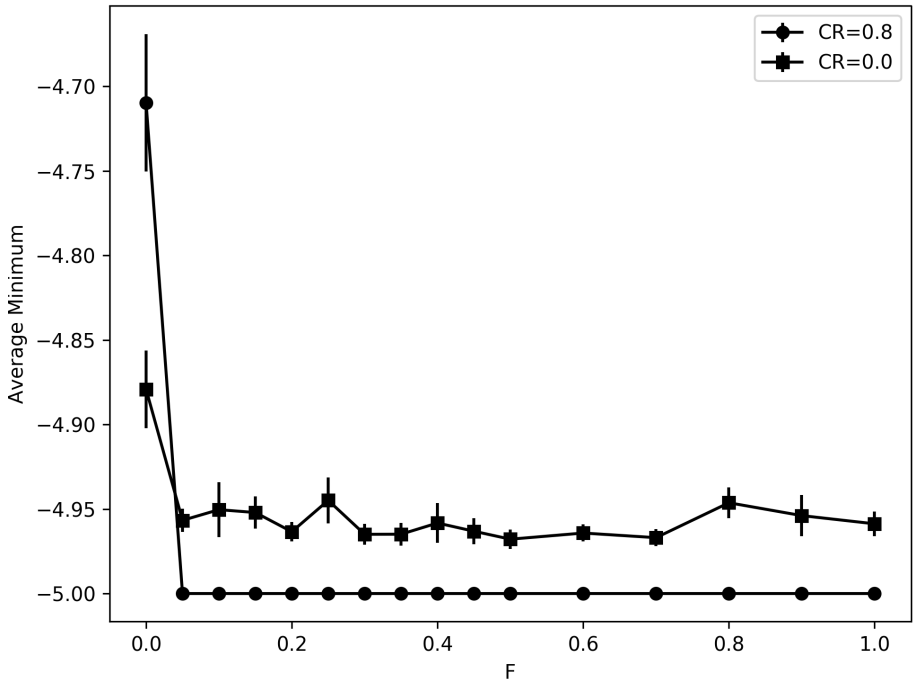


Figure 7.6: Average minimum found as a function of F, the differential scale factor.

wants eight digits of accuracy.

For PSO, we see steady improvement in the convergence rate as the population size increases. A swarm of 100 particles converges to the minimum by around iteration 150. We observe similar performance from GWO, which, for this problem, converges quickly compared to the other algorithms once the number of particles is above twenty. Even GA follows this trend but, as we might expect, performs poorly when the number of particles is small. Improvement happens with iteration, but slowly due to the lack of diversity in the population, behavior that reflects real-world dynamics of species.

Jaya does not perform well on this task, regardless of the size of the swarm. Swarms of five particles or 200 particles behave much the same. Why is not immediately clear – we invoke the "No Free Lunch" theorem and claim this task is simply unsuited to Jaya's approach. We saw Jaya work well on other tasks in Chapter 5.

What is most impressive about Figure 7.7 is DE. Even a tiny swarm of five particles finds the global minimum by iteration 200, and all larger swarms perform equally well. Additionally, the error bars on the DE curves vanish for all but the first several tens of iterations. DE converges quickly and consistently for this problem. Compare with the error bars for RO, which we might expect to remain relatively large as multiple searches proceed given the purely stochastic nature of each search. DE uses a principled application of randomly selected swarm positions while RO uses randomly selected candidate positions, so we might expect run to run variance to be larger.

The code in `de_gaussian_5d.py` produces output reflecting how often each algorithm succeeds in finding the 5D global minimum value, the wrong minimum value, or no minimum (a failure), as a function of swarm size and number of iterations. A run of the code produced,

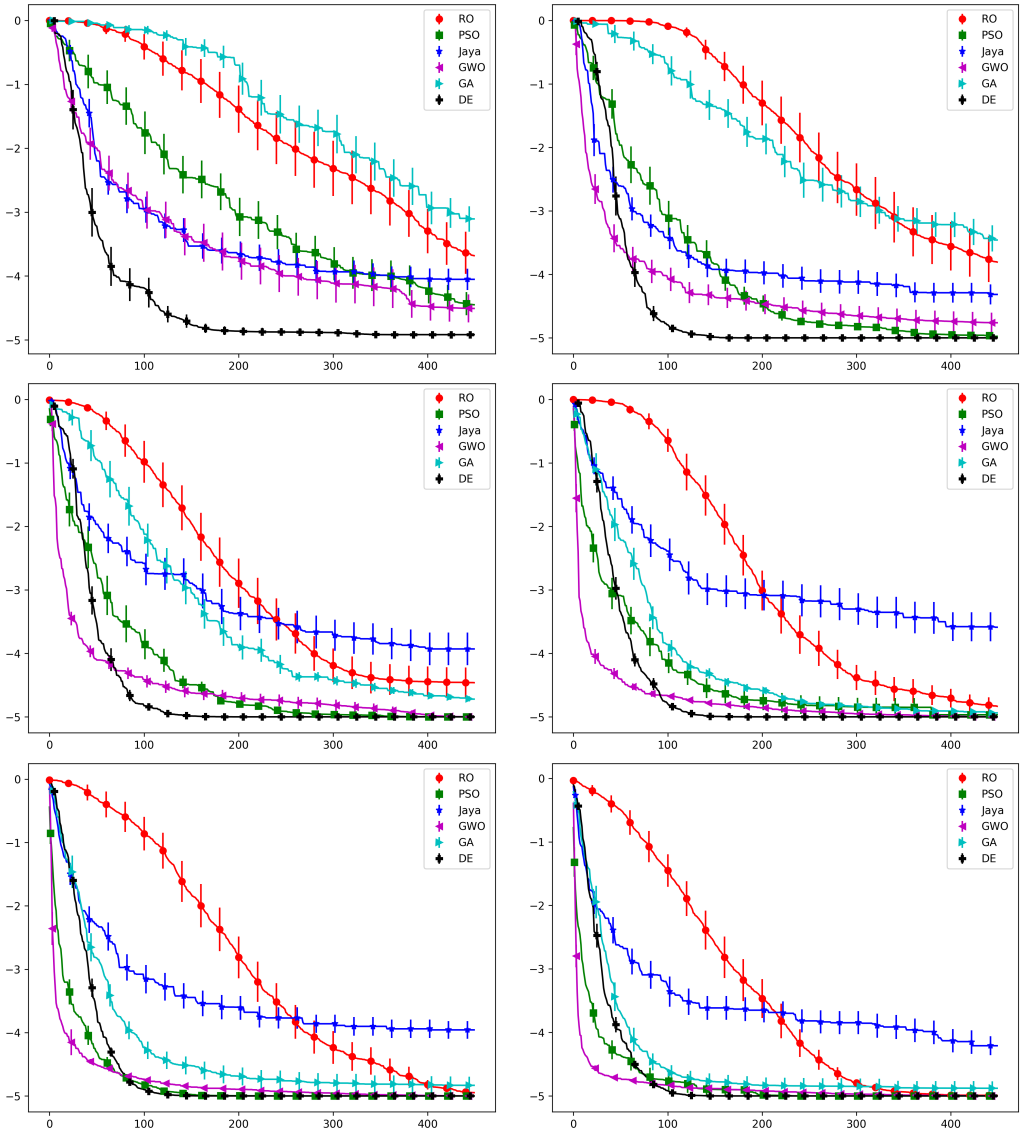


Figure 7.7: Mean swarm best ($n = 20$) by iteration and algorithm type for the 5D Gaussian. By row from the top left: 5 particles, 10 particles, 20 particles, 50 particles, 100 particles, and 200 particles.

(20, 100)	s	w	f
RO	1	0	11
PSO	7	5	0
Jaya	6	5	1
GWO	9	3	0
GA	5	2	5
DE	11	1	0

(300)	s	w	f
RO	3	5	4
PSO	11	1	0
Jaya	11	1	0
GWO	5	7	0
GA	9	2	1
DE	11	1	0

(500)	s	w	f
RO	6	4	2
PSO	5	7	0
Jaya	10	2	0
GWO	9	3	0
GA	5	7	0
DE	12	0	0

(50, 100)	s	w	f
RO	4	1	7
PSO	5	7	0
Jaya	7	4	1
GWO	5	7	0
GA	6	6	0
DE	12	0	0

(300)	s	w	f
RO	10	1	1
PSO	7	5	0
Jaya	11	1	0
GWO	6	6	0
GA	6	6	1
DE	12	0	0

(500)	s	w	f
RO	11	1	0
PSO	8	4	0
Jaya	11	1	0
GWO	5	7	0
GA	5	7	0
DE	12	0	0

(100, 100)	s	w	f
RO	1	3	8
PSO	9	3	0
Jaya	10	2	0
GWO	6	6	0
GA	6	5	1
DE	12	0	0

(300)	s	w	f
RO	12	0	0
PSO	8	4	0
Jaya	11	1	0
GWO	10	2	0
GA	7	5	0
DE	12	0	0

(500)	s	w	f
RO	12	0	0
PSO	8	4	0
Jaya	12	0	0
GWO	6	6	0
GA	8	4	0
DE	12	0	0

where we've condensed and adjusted the format of the output to improve readability. Each row above represents a swarm of 20, 50, or 100 particles and each table shows the successful (s), wrong (w), and failed (f) searches for the given number of iterations: 100, 300, or 500.

The clear star of the show is DE. For even the smallest swarms with the fewest iterations, DE finds the global minimum virtually every time. Again, strong evidence of DE's utility and justification for including it in our pantheon of algorithms. A close second is RO, especially when the size of the swarm is larger.

Jaya puts on a good show in that it moves towards the global minimum once the number of iterations is high enough, but as Figure 7.7 makes clear, the final minimum found is disappointingly short of the goal. Compare Jaya's performance with DE, which often located the global minimum value with complete precision. Even GWO and PSO, more hit-or-miss as to whether the right minimum is selected, locate the minimum with high accuracy when found.

Perhaps Jaya simply needs more time, or rather, iterations. The code in `jaya_5d.py` runs a swarm of 100 particles for 12 runs for each iteration limit, up to 160,000 iterations. The results are,

```
  700: min= -4.2601205 +/- 0.1378435 (success=12, wrong=0, fail=0)
 1000: min= -4.1938387 +/- 0.1247555 (success=12, wrong=0, fail=0)
 1500: min= -4.3806454 +/- 0.1374493 (success=12, wrong=0, fail=0)
 2000: min= -4.2206336 +/- 0.1928849 (success=12, wrong=0, fail=0)
 2500: min= -4.5656279 +/- 0.0965143 (success=12, wrong=0, fail=0)
 5000: min= -4.5995791 +/- 0.0888398 (success=12, wrong=0, fail=0)
10000: min= -4.6535931 +/- 0.0779748 (success=11, wrong=1, fail=0)
20000: min= -4.9029048 +/- 0.0487318 (success=12, wrong=0, fail=0)
40000: min= -4.9909208 +/- 0.0017837 (success=12, wrong=0, fail=0)
80000: min= -4.9962642 +/- 0.0005475 (success=11, wrong=1, fail=0)
```

```
160000: min= -4.9968910 +/- 0.0003000 (success=12, wrong=0, fail=0)
```

The leading number is the iteration limit. With only a few exceptions, the Jaya searches find the correct minimum, but even with a high number of iterations, Jaya fails to converge to the minimum with high precision. DE is the clear overall winner for this task.

Even the basic experiments above make a strong case for making DE a "go to" algorithm, one that you should consult for many, if not most, problems. DE won't always be the best option, but it's likely to put its best foot forward for most applications.

This concludes our investigation of the DE algorithm and with it our toolkit. We have all the swarm optimization algorithms we need for the remainder of the book. Naturally, we shouldn't be completely satisfied when the growing universe of optimization algorithms lies virtually unexplored before us, but we need to start somewhere. Hopefully, the framework is flexible enough to make adding new algorithms a straightforward process, so please do so as you go forward with the book.

Part II

Experiments

Chapter 8

Initial Experiments

This chapter presents a potpourri of experiments intended to familiarize us with the framework we developed in Part I of the book. The goal of the experiments, beyond characterizing the differences between the algorithms for problems more interesting than finding the minimum of an inverted Gaussian, is to work through the typical thought process used when approaching a new swarm optimization problem.

Specifically, we start with standard test functions. You'll see these in papers as you review the metaheuristic literature, so they are a natural place to begin (Section 8.1). Next, we explore a classic computer science problem: the 0-1 knapsack (Section 8.2). Finally, we close out the chapter with nonlinear curve fitting (Section 8.3).

For all but the standard test functions, we'll walk through the problem, what it entails, and how we are mapping a potential solution to the swarm domain. The mapping happens primarily through the objective function and the process of creating and interpreting the meaning of a particle position in the search space. This process needs to be followed for each problem we wish to solve, so we'll get our feet wet here before moving on to more challenging examples in later chapters.

8.1 Standard Test Functions

We know there are many swarm algorithms, likely hundreds, so it shouldn't surprise us to realize that the research community has developed a series of standard test functions for evaluating them. The test functions show up time and again in papers describing new algorithms. We'll use a few of the test functions here to compare the performance of the algorithms, like what we did to test the algorithms in Part I, but here the functions are more challenging.

We'll work with five functions: two 2D and three of arbitrary dimensionality that we'll vary from three to 50 dimensions. The functions are defined in Table 8.1 along with their names and the location and value of their minimums. Plots of the functions in 3D space are given in Figure 8.1.

Each function has its own test script: `sphere.py`, `rastrigin.py`, `rosenbrock.py`, `beale.py`, and `easom.py`. Let's walk through `sphere.py` understanding that the difference between it and the other test scripts is to update the `Objective` method with the proper objective function.

The `sphere.py` script begins by importing necessary modules and the definition of a helper function to calculate the Euclidean distance between the swarm best position and

$f(\boldsymbol{x}) = 10n + \sum_{i=1}^{n}[x_i^2 - 10\cos(2\pi x_i)]$	(Rastrigin)
$f(\boldsymbol{x}) = \sum_{i=1}^{n} x_i^2$	(Sphere)
$f(\boldsymbol{x}) = \sum_{i=1}^{n-1}[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2]$	(Rosenbrock)
$f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$	(Beale)
$f(x, y) = -\cos(x)\cos(y)e^{-((x-\pi)^2+(y-\pi)^2)}$	(Easom)
$f(0, \ldots, 0) = 0$	(Rastrigin)
$f(0, \ldots, 0) = 0$	(Sphere)
$f(1, \ldots, 1) = 0$	(Rosenbrock)
$f(3, 0.5) = 0$	(Beale)
$f(\pi, \pi) = -1$	(Easom)

Table 8.1: (Top) The test functions. The first three are n-dimensional while the final two are 2-dimensional. (Bottom) The location and value of the minimum for each test function.

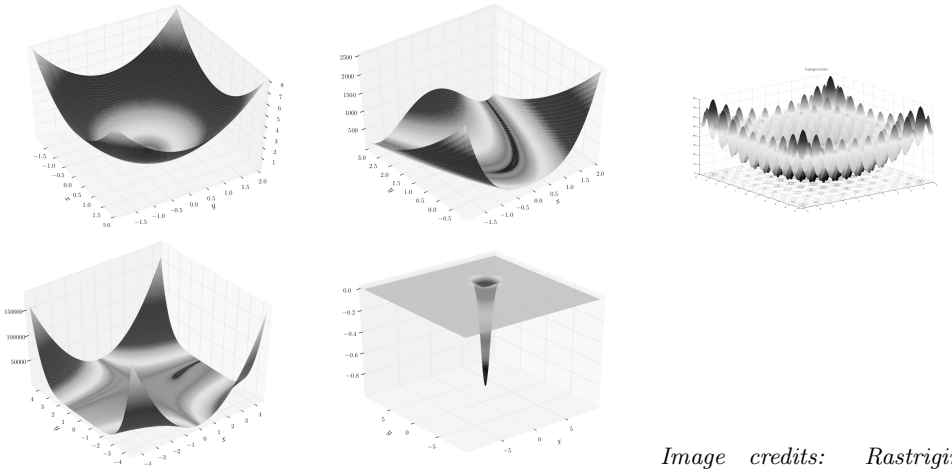


Image credits: Rastrigin public domain, others Creative Commons Attribution-Share Alike (author Gaortizg).

Figure 8.1: Plots of the standard test functions. Top: Sphere, Rosenbrock, Rastrigin. Bottom: Beale and Easom.

the known function minimum, which for the sphere is at $\boldsymbol{x} = \mathbf{0}$,

```
import sys
import time

from PSO import *
from DE import *
from RO import *
from GWO import *
from Jaya import *
from GA import *

from RandomInitializer import *
from SphereInitializer import *
from QuasirandomInitializer import *
from Bounds import *
from LinearInertia import *

def dist(p):
    m = np.zeros(len(p))
    return  np.sqrt(((p-m)**2).sum())
```

The `import` and `from` statements make the framework available. The `dist` function takes the swarm's best position as a vector input (`p`). The length of `p` determines the dimensionality of the search space, and `m` is a zero vector representing the minimum position. To calculate the Euclidean distance, the difference between `p` and `m` is squared, then summed, followed by the square root. We'll use `dist` to calculate how far off the swarm is in space from the known minimum. Subtracting $\boldsymbol{m}$, which is all zeros, from $\boldsymbol{p}$ adds a bit of overhead to the code but makes explicit the fact that we are looking for the distance between two vectors.

The objective function comes next,

```
class Objective:
    def __init__(self):
        self.fcount = 0
    def Evaluate(self, p):
        self.fcount += 1
        return (p**2).sum()
```

Recall, the objective function is an instance of a class with an `Evaluate` method accepting a particle position (`p`). The sphere objective is simply the sum of the squares of the components of the position. Notice that the constructor sets a member variable, `fcount`, to zero and that `Evaluate` increments `fcount` by one each time it's called. We'll use this to track how many times the swarm evaluates the objective function.

The main part of the script is next. In the first part, command line arguments are parsed and the bounds and initializer defined,

```
ndim = int(sys.argv[1])
npart = int(sys.argv[2])
max_iter = int(sys.argv[3])
alg = sys.argv[4].upper()
itype = sys.argv[5].upper()
b = Bounds([-1]*ndim,[1]*ndim)
if (itype == "SI"):
    i = SphereInitializer(npart, ndim, bounds=b)
elif (itype == "QI"):
```

```
    i = QuasirandomInitializer(npart, ndim, bounds=b)
else:
    i = RandomInitializer(npart, ndim, bounds=b)
```

The bounds are set to $[-1, 1]$ for each dimension. The number of dimensions, particles, and maximum iterations is given on the command line.

The objective function and swarm are defined next,

```
obj = Objective()

if (alg == "PSO"):
    swarm= PSO(obj=obj, npart=npart, ndim=ndim, init=i, max_iter=max_iter,
               bounds=b, inertia=LinearInertia())
elif (alg == "DE"):
    swarm= DE(obj=obj, npart=npart, ndim=ndim, init=i, max_iter=max_iter,
               bounds=b)
elif (alg == "RO"):
    swarm= RO(obj=obj, npart=npart, ndim=ndim, init=i, max_iter=max_iter,
               bounds=b)
elif (alg == "GWO"):
    swarm= GWO(obj=obj, npart=npart, ndim=ndim, init=i, max_iter=max_iter,
               bounds=b)
elif (alg == "JAYA"):
    swarm= Jaya(obj=obj, npart=npart,ndim=ndim, init=i, max_iter=max_iter,
               bounds=b)
elif (alg == "GA"):
    swarm= GA(obj=obj, npart=npart, ndim=ndim, init=i, max_iter=max_iter,
               bounds=b)
```

Each algorithm type is supported. We saw similar code in Part I. An instance of the requested algorithm is created passing in the objective function object (`obj`), the number of particles (`npart`), dimensions (`ndim`), the initializer (`i`), the maximum number of swarm updates (`max_iter`), and the bounds object (`b`).

With everything in place, we can now run the search and evaluate the results,

```
st = time.time()
swarm.Optimize()
en = time.time()

res = swarm.Results()
b = res["gbest"][-1]
p = res["gpos"][-1]
count = swarm.obj.fcount

print("fmin = %0.16e at:" % (b,))
print("distance from minimum = %0.16e" % dist(p))
for i in range(ndim):
    print("    {: .16e}".format(p[i]))
print("(%d swarm best updates, %d function evals, time: %0.3f seconds)" %
        (len(res["gbest"]), count, en-st))
```

A single call to `Optimize` performs the search. The `Results` method returns the swarm best objective function value and position. These are reported along with the distance between the swarm best and the known minimum.

To call `sphere` use a command line like,

```
> python3 sphere.py 10 100 2000 RO RI
```

to use a 10-dimensional search space, 100 particles, 2000 iterations, random optimization, and a random initializer. A run of the above produces output similar to,

```
fmin = 1.0234527774198612e-05 at:
distance from minimum = 3.1991448504559172e-03

     9.6543865922902835e-04
     9.7852774626079326e-04
     1.0057781815023493e-03
     1.0181475366500821e-03
    -9.5983273434426637e-04
    -1.2928723766729572e-03
     7.3158773114536757e-04
     1.1750084066928952e-03
     7.4488248369607916e-04
    -1.1105008435513333e-03

(1001 swarm best updates, 200100 function evals, time: 3.742 seconds)
```

The swarm converged reasonably well to the minimum at zero (again, "reasonable" is relative). Notice, there were over 1000 swarm best updates. The number of objective function calls is 100 to initialize the swarm, one for each particle, and $2000 \times 100 = 200,000$ to complete the search.

If we leave everything the same and switch to a different algorithm, in this case GWO, we might get,

```
fmin = 0.0000000000000000e+00 at:
distance from minimum = 0.0000000000000000e+00

    -1.3487853396538051e-162
    -1.3131842656860750e-162
     7.7128119028146570e-163
    -1.1606074164527127e-162
     1.4918179808454143e-162
    -8.1775665831359060e-163
     1.1902081581475020e-162
     4.6277860169243430e-163
    -1.0656099228711197e-162
    -1.3206814865781378e-162

(3476 swarm best updates, 200100 function evals, time: 14.845 seconds)
```

showing GWO to be far superior to RO in this case, as we might expect.

The results above are quite similar to those of Part I, and you can experiment with the other test scripts at your leisure. Likewise, in Part I, we developed many tests of the algorithms using a 2D or 5D Gaussian. Naturally, we can replace the objective function in those experiments with any of the standard test functions, perhaps limited to only two dimensions, and produce similar outputs. We leave that as an exercise for the interested reader.

The file `test_functions.py` contains a script to evaluate a given standard test function using each swarm algorithm for a given swarm size and a maximum number of iterations. If the test function is sphere, Rosenbrock, or Rastrigin, the dimensionality is

varied from three to 50. The output is the mean and standard error of the minimum value found and the distance between the minimum position and the known function minimum over ten searches. The goal of `test_functions.py` is to show the relative performance of the swarm algorithms. The repository contains output files generated by `test_functions.py` with names like `test_sphere_summary.txt`. The script `test_functions` runs the test for all the standard functions using swarms of 30 particles and 2000 iterations. As the dimensionality increases, it is reasonable to believe that the number of iterations of the swarm should also increase.

For example, a run using the Rastrigin function produced the following output,

```
3 dimensions, 2000 iterations:
    RO  : min: 0.0000 +/- 0.0000, dist: 0.0000 +/- 0.0000
    PSO : min: 0.5969 +/- 0.2199, dist: 0.5386 +/- 0.1837
    DE  : min: 0.0000 +/- 0.0000, dist: 0.0000 +/- 0.0000
    GWO : min: 0.0000 +/- 0.0000, dist: 0.0000 +/- 0.0000
    Jaya: min: 0.0000 +/- 0.0000, dist: 0.0000 +/- 0.0000
    GA  : min: 0.0013 +/- 0.0004, dist: 0.0022 +/- 0.0004

10 dimensions, 6666 iterations:
    RO  : min:  2.6719 +/- 0.3251, dist: 1.5207 +/- 0.0877
    PSO : min: 11.0694 +/- 1.7334, dist: 2.4905 +/- 0.1715
    DE  : min:  1.5059 +/- 0.4859, dist: 0.9891 +/- 0.1916
    GWO : min:  0.0000 +/- 0.0000, dist: 0.0000 +/- 0.0000
    Jaya: min: 10.4107 +/- 0.7590, dist: 2.2255 +/- 0.0780
    GA  : min:  0.0062 +/- 0.0013, dist: 0.0052 +/- 0.0006

25 dimensions, 16666 iterations:
    RO  : min: 11.1665 +/- 0.4215, dist: 2.8181 +/- 0.0548
    PSO : min: 58.9343 +/- 1.1008, dist: 5.1038 +/- 0.0321
    DE  : min:  9.4809 +/- 0.6602, dist: 2.9468 +/- 0.1023
    GWO : min:  0.0000 +/- 0.0000, dist: 0.0000 +/- 0.0000
    Jaya: min: 50.6164 +/- 1.3750, dist: 4.6063 +/- 0.0723
    GA  : min:  0.0182 +/- 0.0022, dist: 0.0094 +/- 0.0005

50 dimensions, 33333 iterations:
    RO  : min:  33.2555 +/- 0.7373, dist: 4.4186 +/- 0.0588
    PSO : min: 132.6803 +/- 3.1006, dist: 7.4736 +/- 0.0390
    DE  : min:  52.3571 +/- 4.4522, dist: 5.7280 +/- 0.1212
    GWO : min:   1.1792 +/- 0.7900, dist: 0.4662 +/- 0.3112
    Jaya: min: 127.6932 +/- 4.2923, dist: 7.2405 +/- 0.0648
    GA  : min:   0.0538 +/- 0.0043, dist: 0.0163 +/- 0.0006
```

where the full numeric output has been truncated to four digits.

The results above show most algorithms perform well when the dimensionality is low but begin to struggle as the dimensionality increases, at least for small swarm sizes and number of iterations. Two algorithms stand out, however. The first is GWO, which produced a somewhat poor showing in our Gaussian tests in Part I. Here, GWO produces perfect results for even 25 dimensions. At 50 dimensions, we see some decrease in performance, but the result is still better than virtually any other algorithm.

The second surprise performer is the genetic algorithm. We saw it converge slowly for the simpler examples in the first half of the book, but here it does a reasonable job even with the 50-dimensional case.

The output from `test_functions.py` for the two-dimensional Beale function is,

```
RO  : min: 0.0000 +/- 0.0000, dist: 0.0007 +/- 0.0001
```

```
PSO : min: 0.0796 +/- 0.0796, dist: 0.7035 +/- 0.7035
DE  : min: 0.0000 +/- 0.0000, dist: 0.0000 +/- 0.0000
GWO : min: 0.0796 +/- 0.0796, dist: 0.7036 +/- 0.7035
Jaya: min: 0.0000 +/- 0.0000, dist: 0.0000 +/- 0.0000
GA  : min: 0.1029 +/- 0.0968, dist: 0.6905 +/- 0.5400
```

demonstrating that several algorithms perform very nicely on this function (30 particles, 2000 iterations). The seemingly identical output for PSO and GWO is an illusion caused by truncating the output to four digits.

The output presented above fixed the swarm size and number of iterations. What if, instead, we fixed the algorithm? Let's select DE/rand/1/bin and vary the swarm size and number of iterations. We'll test the 22-dimensional Rosenbrock function for swarm sizes from 10 to 100 particles and iterations from 100 to 50,000. The code we need is in `rosenbrock_de.py`. The main loop is,

```
outname = sys.argv[1]
navg = 6
ndim = 22
results = []

for npart in [10,20,30,40,50,60,70,80,90,100]:
    for max_iter in [100,5000,10000,15000,20000,25000,30000,35000,40000,
                     45000,50000]:
        r = np.zeros(navg)
        for m in range(navg):
            b = Bounds([-1.1]*ndim,[1.1]*ndim)
            i = RandomInitializer(npart, ndim, bounds=b)
            obj = RosenbrockObjective()
            swarm = DE(obj=obj, npart=npart, ndim=ndim, init=i,
                       max_iter=max_iter, bounds=b)
            swarm.Optimize()
            res = swarm.Results()
            p = res["gpos"][-1]
            r[m] = dist(p)
        rmean = r.mean(axis=0)
        rSE = r.std(axis=0, ddof=1) / np.sqrt(navg)
        results.append([npart, max_iter, rmean, rSE])
        print("npart: %3d, max_iter: %6d, dist: %0.16f +/- %0.16f"
            % (npart, max_iter, rmean, rSE), flush=True)

results = np.array(results)
np.save(outname, results)
```

The output is a NumPy array containing the mean and standard error of the distance between the swarm best position and the minimum of the 22-dimensional Rosenbrock function.

We could show the output as a table, but it is more interesting to present it as an image. The output represents samplings of a 2D function where the arguments are the swarm size, number of iterations, and the outputs are the mean distance from the swarm best and the true Rosenbrock minimum. Further, we can use spline interpolation to estimate the performance over a grid of swarm and iteration points. Let's show the code to get this far,

```
import numpy as np
import matplotlib.pylab as plt
from PIL import Image
```

```
from scipy.interpolate import interp2d

d = np.load("rosenbrock_de_results.npy")
x = d[:,0]
y = d[:,1]
z = d[:,2]

func = interp2d(x,y,z, kind="cubic")
xmin, xmax = x.min(), x.max()
ymin, ymax = y.min(), y.max()
X = np.linspace(xmin, xmax, 72)
Y = np.linspace(ymin, ymax, 72)
zimg = np.zeros((len(X),len(Y)))
for i,x in enumerate(X):
    for j,y in enumerate(Y):
        zimg[i,j] = func(x,y)
```

The code imports the standard modules and the function `interp2d` from `scipy.interpolate`. The `interp2d` function accepts a set of input points, which need not be on a grid, and returns an interpolation function using cubic splines to map any x and y to an output, z. For us, x is the swarm size, and y is the number of iterations. That leaves z, the estimated distance between the swarm best and the true minimum.

With `x`, `y`, and `z`, we create the interpolating function, `func`, and use it to fill in a two-dimensional array, `zimg`. The `linspace` function returns a vector from `xmin` to `xmax` in 72 steps (`X`). These are the swarm sizes to interpolate. Similarly, `Y` is a vector of swarm iterations. Looping over all combinations of the swarm size and iterations lets us use `func` to estimate the swarm's performance.

We want to display the estimated swarm performance as an image. To do that, we first take the log of the estimates, adjusting for any estimates that are less than zero. Taking the log brings out subtle differences. The actual distances in `zimg` range from near zero to above four. Finally, we scale `zimg` to $[0, 1]$ and multiply by 255 to turn it into a grayscale image (`img`). In code,

```
zimg += np.abs(zimg.min())
zimg = np.log(zimg+1)
zimg /= zimg.max()
img = (255.0*zimg).astype("uint8")
img = Image.fromarray(img).resize((10*len(X),10*len(Y)))
img = np.array(img)
```

Besides scaling to $[0, 255]$, we also expand the image to 720x720 pixels. All that remains is to display `img`,

```
fig, ax = plt.subplots(figsize=(6,6))
ax.imshow(img, cmap='gray', extent=[100,50000, 100,10], aspect='auto')
plt.xlabel("Iterations")
plt.ylabel("Swarm size")
plt.savefig("rosenbrock_de_results.png", dpi=300)
plt.show()
```

We use `imshow` to set up the image with a grayscale color map and labeling on the axes to match the range of the number of iterations and swarm size.

What does it look like? Each time we run `rosenbrock_de.py`, we'll get different results due to the random nature of swarm initialization. Figure 8.2 shows the result of

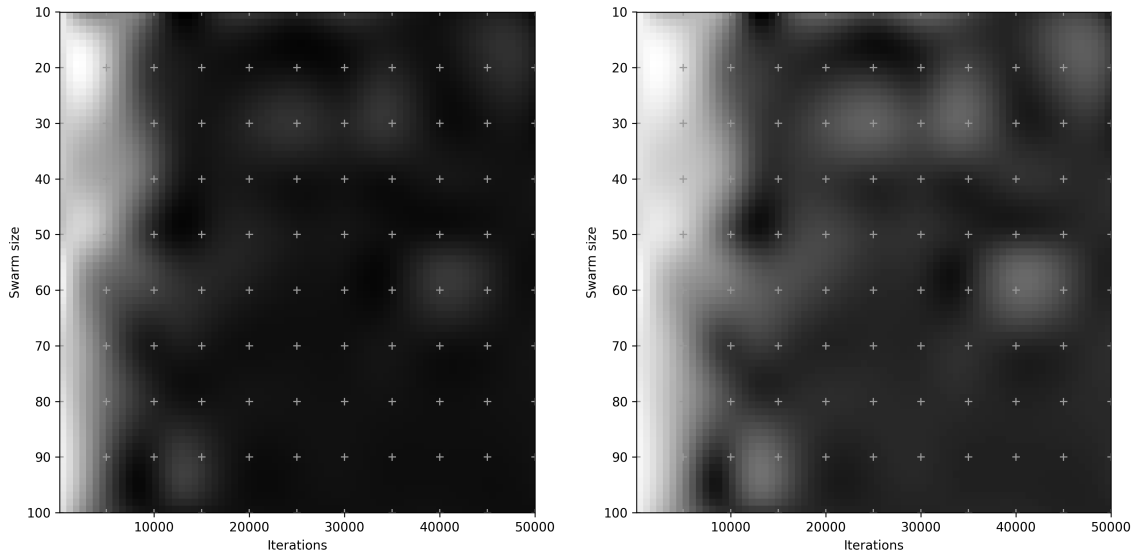


Figure 8.2: A run of the 22-dimensional Rosenbrock DE test code. Linear scaling (left) and log scaling (right). Crosses mark the points sampled. Cubic spline interpolation forms the background image.

one run with a linearly scaled image on the left and a log scaled version on the right. The crosses mark the locations that were sampled.

How should we interpret Figure 8.2? The brighter a region, the further away from the known minimum location. Intuition tells us we should expect small swarms and fewer iterations to do relatively poorly. We see this in Figure 8.2 in the lighter region on the upper left. Similarly, the lower right should show better convergence leading to a darker region. We see this in both the linear and log scaled versions. In general, regardless of swarm size, fewer iterations lead to poorer results, as expected.

Interestingly, even a small swarm with many chances to explore, the upper right of Figure 8.2, does perhaps better than expected. However, the log scaled plot shows that the upper right and lower right regions are still quite different. Visible as lighter and darker "blobs", the variation in final swarm location is likely stochastic with a new run of the test generating a different pattern. However, the overall observations should persist. Viewing the swarm convergence landscape in this way provides an alternative method for building intuition about the entire optimization process.

8.2 The 0-1 Knapsack

We are now ready for our first actual experiment. We have a problem we need to solve, the 0-1 knapsack. We first describe the problem in detail to understand what we need to accomplish (Section 8.2.1). Then we walk through framing the problem as a swarm optimization (Section 8.2.2) and, finally, run the experiment to discover useful solutions (Section 8.2.3). We'll use this format throughout the remainder of the book, if only implicitly at times. Let's get started.

8.2.1 The Problem

The 0-1 knapsack problem is a classic of computer science. It is normally solved via dynamic programming. We'll frame it differently here, so swarm optimization techniques apply. The problem is easy to state:

> We have a knapsack and a collection of items, one of each kind. The knapsack has a fixed weight capacity that we may not exceed. Each of the items has two properties: a weight and a value. What is the best combination of items that maximize the value without exceeding the weight limit?

The "0-1" part of the name comes from having only one instance of each item. Either we place it in the knapsack (1), or we don't (0). As we'll see, this restriction simplifies the setup. We'll avoid speculation as to *why* we desire to fill the knapsack with the most valuable collection of objects possible.

Here's a simple example of the problem and a solution for it. First, we need a list of items with weights and values,

Item	*Value*	*Weight*
platinum bar	24	24
gold bar	20	10
silver bar	15	8
copper bar	5	7

Second, we need to know the maximum weight the knapsack can hold before it breaks. In this case, it's 25. The problem is simple, and the solution is easy to see with a few moments of thought: we select the gold, silver, and copper bars. If we do, our total value is 40, and the weight is 25, the limit of the knapsack. If we select the platinum bar, the weight is 24 leaving only 1 extra. None of the other items weigh so little; therefore, we could only take the platinum bar. That's a value of 24, nowhere near the 40 we get by taking the other three items.

Let's see how to cast the problem in a form where we can search for the solution.

8.2.2 The Setup

To use a swarm, we need to frame the solution to the problem in the form of a vector. The name of the problem gives us a clue: "0-1 knapsack". Either we include an item, or we don't. So, let's use a vector where each element represents one of the items on our list. The number of items available then determines the dimensionality of the search. The value we'll use for each item is whether we include it or not.

We have four possible items for the simple example problem, so our search space would have four dimensions. The solution was to ignore the platinum bar and take the other three. We can write this as a binary vector,

$$\text{platinum, gold, silver, copper} = 0,\ 1,\ 1,\ 1$$

Therefore, we'll use a binary vector to represent the set of items selected. Each particle in the swarm then represents a possible solution, a possible collection of items. Using a binary vector also sets the bounds of the search. For each item (particle dimension), we bound the search to the range $[0, 1]$.

Are we good to go? Not quite. We want binary vectors; we don't know what it means to include an item if the particle position element representing it is 0.344. Additionally, we need to enforce the weight limit. That's an additional bound on the problem, but not one applied to a particular particle dimension. How can we include the discretization and weight requirements?

For our framework, we developed a `Bounds` class that knows how to enforce bounds and check limits. Let's subclass it for the knapsack problem. We'll leave the `Limits` method alone, but now use the previously empty `Validate` method. We'll use `Validate` to make the position vectors binary in each element, either 0 or 1.

We've now set up most of the problem. We know how to represent solutions as binary vectors, and we know how to enforce the binary requirement. We still need to know how to decide if one set of items is a better solution than another. That's the objective function.

As each item has a value, and we want to *maximize* the value of the items in the knapsack, the sum of the values of the selected items is a natural measure, assuming the weight limit of the knapsack is not exceeded. Our framework always minimizes, however, so instead of returning the overall value of the items represented by a particle, we'll return the negation of that value, so more valuable combinations yield smaller numbers.

At this point, we are ready to build the solution in code. Here's how we're setting it up,

Solution representation	Binary vector. Each element is an item.
Boundary conditions	Subclass `Bounds` using `Validate` to discretize.
Objective function	Negative of the sum of the values of selected items.

The complete code for this problem is in the file `knapsack.py`, which is in the `knapsack` directory. We'll start with the objective function, then the subclass of `Bounds`.

The objective function needs to accept the item values and weights along with the maximum weight allowed. Additionally, the `Evaluate` method needs to accept a vector representing a collection of selected items from which it calculates the overall value and weight. If the weight exceeds the maximum, we return a huge positive number. This captures the weight constraint. Otherwise, we return the negative of the overall value of the selected items. The code becomes,

```
class Objective:
    def __init__(self, values, weights, max_weight):
        self.values = values
        self.weights= weights
        self.max_weight = max_weight
        self.fcount = 0
    def Evaluate(self, p):
        self.fcount += 1
        value = (self.values*p).sum()
        weight= (self.weights*p).sum()
        if (weight > self.max_weight):
            return 1e9
        return -value
```

The constructor accepts two NumPy arrays, `values` and associated `weights`. It also accepts the weight limit for the knapsack (`max_weight`). These are simply stored in the

instance along with setting `fcount` to zero. We saw `fcount` in Section 8.1, where it was used to count the number of objective function evaluations during a search.

The `Evaluate` method is straightforward. We count the call by incrementing `fcount`. Then we compute the total value and weight of the candidate solution represented by `p`, a swarm position vector. We know the elements of `p` are either zero or one, so multiplying the item values and weights by `p` and summing gives us the total value and weight. A quick check to see if the weight exceeds the limit or not, and the negative of the overall value is returned. This completes the objective function; let's now build the custom bounds class.

The code for the `KnapsackBounds` class subclasses `Bounds`. The constructor becomes,

```
class KnapsackBounds(Bounds):
    def __init__(self, lower, upper):
        super().__init__(lower, upper, enforce="resample")
```

where the constructor takes the expected `lower` and `upper` bounds on each particle position, here always zero and one. The `lower` and `upper` limits are passed to the superclass. Notice that we are telling the superclass to enforce limits by resampling. For example, see the `Limits` method of the `Bounds` class. Next, we implement our custom `Validate` method,

```
def Validate(self, p):
    p[np.where(p < 0.5)] = 0
    p[np.where(p >= 0.5)]= 1
    return p.astype("float64")
```

Recall, `Validate` is called by the `Limits` method of the `Bounds` class. Here, the first two lines take the floating-point input position vector, which is already bounded to $[0, 1]$, and maps values less than 0.5 to zero and values 0.5 or above to one. This makes `p` a binary vector representing a particular collection of items. Finally, we return the updated position, ensuring it is still a floating-point vector.

The 0-1 knapsack problem is common, so there are many known examples for us to use. The included examples are in the form of lists of items as a value/weight pair along with a maximum weight for the knapsack. For each example, the optimal solution is given to check whether we've found it.

Here's one example, `f1_l-d_kp_10_269`, found in the `low-dimensional` folder,

```
10 269
55 95
10 4
47 60
5 32
4 23
50 72
8 80
61 62
85 65
87 46
```

The first line supplies the number of items (10) and the maximum allowed knapsack weight (269). This is followed by one line for each item as a value/weight pair.

The files `problem_generator.py` and `brute_force.py` can be used together to generate new 0-1 knapsack problems. For example, to generate a new problem with 14 items use,

```
> python3 problem_generator.py 14 examples/example14
```

This creates the problem file. To find the optimum set of items via brute force use,

```
> python3 brute_force.py examples/example14
```

Bear in mind that all possible combinations of items will be examined to find the optimum solution. For example, a brute force search for a 26 item problem took over 30 minutes on an Intel i5 machine. A 27 item search would take about twice as long and so on.

Let's add a function to parse the problem files,

```
def LoadProblemFile(fname):
    lines = [i for i in open(fname)]
    n,wmax = [float(i) for i in lines[0].split()]
    n = int(n)
    values = np.zeros(n)
    weights= np.zeros(n)
    for i in range(n):
        v,w = [float(j) for j in lines[i+1].split()]
        values[i] = v
        weights[i] = w
    return values, weights, wmax
```

`LoadProblemFile` is plain text processing. We load the lines of the file, pull out the number of items (`n`) and the maximum weight (`wmax`). This lets us set up `values` and `weights` vectors to hold the per-item data. A simple loop extracts each item's value and weight. Finally, we return a list of values, weights, and the maximum allowed weight. The optimum set of items is in the `optimum` directory in the file with the same name as the problem file.

Let's put the pieces together to build the complete solution. We already have `Objective`, `KnapsackBounds`, and `LoadProblemFile`, so let's describe the remaining parts of `knapsack.py`. First, we import the necessary modules (not shown) and then set up the problem including parsing the command line arguments and loading the example file,

```
npart = int(sys.argv[2])
max_iter = int(sys.argv[3])
alg = sys.argv[4].upper()
itype = sys.argv[5].upper()

values, weights, max_weight = LoadProblemFile(sys.argv[1])

obj = Objective(values, weights, max_weight)
ndim = values.shape[0]
b = KnapsackBounds([0]*ndim, [1]*ndim)

if (itype == "RI"):
    ri = RandomInitializer(npart, ndim, bounds=b)
elif (itype == "QI"):
    ri = QuasirandomInitializer(npart, ndim, bounds=b)
else:
    ri = SphereInitializer(npart, ndim, bounds=b)

if (alg == "PSO"):
    swarm= PSO(obj=obj,npart=npart, ndim=ndim, max_iter=max_iter, init=ri,
               bounds=b, inertia=LinearInertia())
```

```
elif (alg == "DE"):
    swarm= DE(obj=obj, npart=npart, ndim=ndim, max_iter=max_iter, init=ri,
                bounds=b)
elif (alg == "RO"):
    swarm= RO(obj=obj, npart=npart, ndim=ndim, max_iter=max_iter, init=ri,
                bounds=b)
elif (alg == "GWO"):
    swarm= GWO(obj=obj,npart=npart, ndim=ndim, max_iter=max_iter, init=ri,
                bounds=b)
elif (alg == "JAYA"):
    swarm= Jaya(obj=obj,npart=npart, ndim=ndim, max_iter=max_iter, init=ri,
                bounds=b)
elif (alg == "GA"):
    swarm= GA(obj=obj, npart=npart, ndim=ndim, max_iter=max_iter, init=ri,
            bounds=b)
```

The code above is similar to the code we saw in Part I when testing the swarm algorithms. The command line lets us select the desired number of particles, iterations, algorithm type, and initializer. Notice that `ndim` is set by the number of items (the length of `values`) after reading the example file.

The only code left to write performs the search and reports the results,

```
st = time.time()
swarm.Optimize()
en = time.time()

res = swarm.Results()
fcount = swarm.obj.fcount
weight = (weights*res["gpos"][-1]).sum()

print()
print("Final value=%0.2f, weight=%0.1f/%0.1f, objects:"
        % (-res["gbest"][-1], weight, max_weight))
print()
print(np.array2string(res["gpos"][-1].astype("uint8")))
print()
print("(%d swarm best updates, %d function evals, time: %0.3f seconds)"
        % (len(res["gbest"]), fcount, en-st,))
print()
```

The call to `Optimize` evolves the swarm, and `Results` returns a dictionary detailing how the swarm did. The code in `knapsack.py` is now complete. Let's see how well it works.

8.2.3 The Results

We'll test `knapsack.py` with the 10 item example given above,

```
> python3 knapsack.py examples/low-dimensional/f1_l-d_kp_10_269 20 100 RO RI
```

where we're using random optimization, 20 particles, 100 iterations, and random initialization. The output is,

```
Final value=295.00, weight=269.0/269.0, objects:

[0 1 1 1 0 0 0 1 1 1]

(6 swarm best updates, 2020 function evals, time: 0.270 seconds)
```

We're told the best combination found has a value of 295 and a weight right at the limit of 269. If we look at the `f1_l-d_kp_10_269` file in the `low-dimensional-optimum` directory, we see that 295 is the optimal solution. Good, our code appears to be working.

The example above has ten items. Therefore, the number of possible combinations is $2^{10} = 1024$. A brute force search would need to look at every possible combination, which would evaluate 1024 positions in the search space. We see above that the RO search evaluated the objective function over 2000 times, about twice the number of possible combinations of items. Could it be that we didn't search in a meaningful way but simply found the solution by brute force?

If we add a final argument to the command line, we'll store the search results in a Python pickle file (code for this not shown above to save space). The results are a dictionary containing the list of best positions found by the swarm (`gbest`) and at which iteration it was found (`gidx`). These values tell us that the optimal solution with a value of 295 was found at iteration 10 after 220 positions were evaluated. That's far less than the 1024 needed for a brute force search, a good sign that the code is doing what we want.

Let's test the code against a more complex example using the file `f2_l-d_kp_20_878` which sets up a problem with 20 items. With 20 items, we have a possible search space of $2^{20} = 1,048,576$ combinations. The known optimum solution is a value of 1024.

Let's run again using random optimization but bump the iteration count by a factor of ten,

```
> python3 knapsack.py examples/low-dimensional/f2_l-d_kp_20_878 20 1000 RO RI
```

giving,

```
Final value=1024.00, weight=871.0/878.0, objects:

[1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1]

(8 swarm best updates, 20020 function evals, time: 3.556 seconds)
```

indicating the swarm found the optimum solution and only used 20,000 objective function calls to do it. In actuality, the number was likely less as we ran the swarm to completion in terms of iterations. We could hardly do otherwise as, *a priori*, we do not know what the solution might be.

Random optimization has performed well for us on two examples. Let's continue using the 20-item example and see how the other algorithms fare. We'll run the search five times using each algorithm. We'll vary the swarm size and number of iterations a bit as well. The results are in Table 8.2. The script itself is in the file `test_knapsack`.

From Table 8.2, it's clear that RO and GWO are well-suited to this particular search. PSO sometimes finds the best solution, but it doesn't do so reliably.

We stated above that the 20-item search means there are at most $2^{20} = 1,048,576$ combinations of items. It's reasonable to expect, then, that a search involving more than that many objective function calls will likely find the best combination since each objective function call evaluates a combination of items. Running `knapsack.py` with 20 particles and 60,000 iterations gives us 1.2 million objective function calls. That's nearly 200,000 more than the number of combinations a brute-force search would use. Do we find the best combination regardless of the swarm algorithm in this case? It turns out that we don't.

20 particles, 1000 iterations:	
RO	1024, 1024, 1024, 1024, 1024
PSO	958, 967, 1024, 1016, 1024
DE	1016, 961, 933, 995, 978
GWO	1018, 1024, 1024, 1013, 1018
Jaya	1024, 966, 964, 1004, 958
GA	989, 961, 965, 956, 958
100 particles, 1000 iterations:	
RO	1024, 1024, 1024, 1024, 1024
PSO	1010, 1010, 1016, 1024, 1024
DE	991, 979, 990, 981, 973
GWO	1024, 1024, 1024, 1024, 1024
Jaya	1009, 996, 987, 1004, 1018
GA	1013, 970, 996, 996, 996
20 particles, 5000 iterations:	
RO	1024, 1024, 1024, 1024, 1024
PSO	1013, 1009, 996, 997, 1010
DE	997, 987, 972, 976, 965
GWO	1024, 1024, 1024, 1024, 1024
Jaya	1009, 991, 985, 990, 996
GA	1016, 997, 1009, 1009, 983

Table 8.2: Results for the 20 item knapsack search by algorithm and swarm parameters. The final value found for five runs is shown.

If we run the search five times each for each algorithm, we see that DE fails to find the optimum twice, as does Jaya, and GA fails once. RO, PSO, and GWO always converge on the optimum value.

Each increase in the number of possible items results in a doubling of the number of brute-force comparisons. The file `generated_0` contains a 26 item example created with `problem_generator.py`. A (lengthy) run of `brute_force.py` tells us that the optimal solution has a value of 1058. The search space for this example is $2^{26} = 67,108,864$ possible combinations of items. Let's run the algorithms on this problem five times each, like above. We'll use 100 particles for 15,000 iterations leading to 1.5 million objective function evaluations. That's only 2% of the possible combinations a brute force search would need. The results are illustrative,

RO	1058, 1058, 1049, 1058, 1058
PSO	1038, 1038, 1049, 1030, 1041
DE	1029, 1036, 1006, 1018, 1019
GWO	1058, 1058, 1058, 1058, 1058
Jaya	1008, 1036, 1028, 1014, 1021
GA	1007, 1028, 1002, 1018, 1024

First, compared to the results of Table 8.2, we see that all of the algorithms are closer to the optimal value of 1058 than many of the results in the table; more iterations or a larger swarm helps. As previously, GWO is the star achieving a perfect result on all five runs. With one exception, RO does the same. PSO, DE, Jaya, and GA, however, never

find the optimum solution. Once again, we need to consider multiple algorithms as no one algorithm is the best choice in all circumstances.

As a final test, let's run the algorithms against the file `generation_2`. This example has 30 items meaning the search space has just over 1 billion possible combinations. Let's use 100 particles and 30,000 iterations, leading to some 3 million objective function evaluations or about 0.3% of the total possible. A brute force search revealed that the knapsack's optimum value is 1839 found by selecting 29 of the 30 items. Both RO and GWO find this optimum solution showing again that they are well suited to this task.

We shouldn't be too pleased with our swarm solutions, however. As the problem's size increases, so too will the number of swarm iterations necessary to locate the optimum set of items. The dynamic programming approach to this problem works very well for over 10,000 items. It isn't reasonable to expect our swarms to find the solution in a space of $2^{10,000}$ possible combinations. All the same, that the problem can be approached at all by swarm optimization is exciting and fun.

The 0-1 knapsack problem is an excellent introduction to the remaining experiments. It was straightforward to implement using the framework and easy to follow in terms of its operation. The objective function was simple to conceive, and adding the weight constraint was trivial. The difference in the performance of each algorithm was informative. Old standards like PSO and DE were not particularly good at this task. In some cases, PSO did find the optimum solution, and one might imagine tweaking the PSO parameters to see if that improves things. On the other hand, DE seems to have shown its oft-claimed weakness by converging too quickly, thereby missing out on the exploration necessary to find the optimum set of items for the knapsack. RO's good overall performance backs up this intuition as the swarm's extreme individualism means there is no exploitation, only mindless exploration that stumbles upon the optimum solution.

Let's put our knapsack aside and move on to our second short experiment, curve fitting.

8.3 Curve Fitting

Curve fitting is a frequent task in many scientific and engineering disciplines. Experiments generate data, and we'd like to characterize it by fitting a known function. The function's form is known, but the function's parameters are not and need to be determined from the data.

The standard means for fitting a function to a data set is to minimize the mean squared error between the function using the current set of parameters and the data itself. For linear functions, the best fit line can be calculated precisely. However, for nonlinear functions, some form of optimization is usually used. Typical optimization algorithms require knowledge of the derivative of the desired function, not usually an issue, and a set of initial guesses as to what the parameters might be. It's this latter requirement that makes nonlinear curve fitting sometimes tricky. Without good guesses, the optimization algorithms fail to converge. A typical example of a nonlinear optimization routine is the CURFIT routine found in [24] which requires an auxiliary routine to calculate the function value and partial derivatives at a point.

Our implementation bypasses the need to calculate derivatives. Instead, we'll use the points to be fitted and an objective function minimizing the mean squared error between the points and the fit function. Each particle position represents a possible set of parameters. As before, we'll use three steps to explain the problem, set up the swarm solution, and

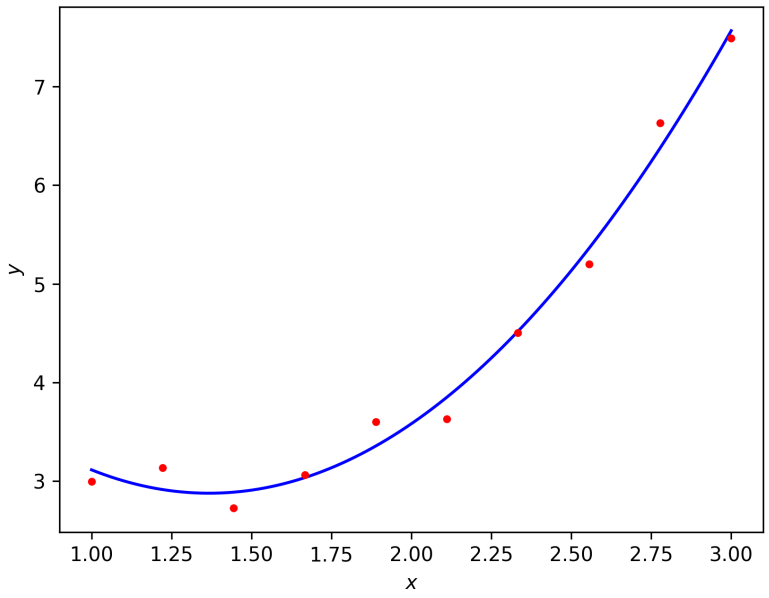


Figure 8.3: A sample dataset and best fit curve to $f(x) = ax^2 + bx + c$.

examine the results.

8.3.1 The Problem

We've run an experiment and collected data in the form of a set of N (x, y) pairs. We know from theory that the data follows a quadratic function, so we want to fit a quadratic to the data. The general form of the function we need to fit is,

$$f(x) = ax^2 + bx + c$$

where a, b, and c are the parameters we need to find to best fit the data. For a specific set of (a, b, c), we can derive a measure of how well the function fits the data by calculating the mean squared error between the y at each x where we have a measurement and the function value. The claim, and it's reasonable, is that the best fit function will have the smallest mean squared error over all the data points. Of course, there need to be enough data points to characterize the function in the first place, but we'll assume our experiment was well designed and we have the necessary information. Therefore, we claim the best set of parameters, (a, b, c), are those where,

$$\text{MSE} = \frac{1}{N}\sum_{i=0}^{N-1}(y_i - f(x_i))^2$$

is as small as possible.

For example, Figure 8.3 shows a set of points and the best fit quadratic found by the code we'll develop in Section 8.3.2. The parameters found were,

$$a = 1.7590716,\ b = -4.810116,\ c = 6.1650996$$

The data points were generated by `make_sample_plot.py` where the base function used $a = 1.5$, $b = -3$, and $c = 4.4$ with random noise of up to 25% added or subtracted from the base y value.

8.3.2 The Setup

Our goal is to find a set of parameters for a function. The number of parameters needed by the function sets the dimensionality of the problem. The example above required three parameters making each particle in the swarm a vector in 3D space. If there were five parameters, the search space would be 5-dimensional.

Are there any special requirements on the types of values the particles can represent? For curve fitting, the answer is no. We want continuous values. What about boundary conditions? Do we need those? Here the answer is yes. We'll experiment with boundaries in Section 8.3.3, but for now, we see that our general rule of thumb holds for curve fitting: the tighter the bounds on the parameters, the better in terms of simplifying the search space. Of course, this sounds a bit like our criticism of other optimization techniques that require an initial guess at the parameters. However, we'll see that we have more freedom to be sloppy and usually get a good solution. The user will set the problem's bounds as inputs to our framework `Bounds` class. No subclass is needed. This leaves the objective function. The natural one to use is the mean squared error, and that's precisely the function we will use. The smaller the MSE, the better the fit.

To put it all together, then, here's the set up,

Solution representation	Particle components represent fit function parameters.
Boundary conditions	Given by the user.
Objective function	The mean squared error between the data and the fit.

However, we're not quite ready to start writing code. We haven't decided on *how* we'll represent the fit function. We know we'll supply a set of data points, but we want to write a generic tool able to fit any set of points to any function. How should we do it?

We're using Python, so why not use the interpreter to help us? We'll pass fit functions to the code via strings and use Python's `eval` function to interpret the string on demand. Doing this frees us from adjusting source code every time we want to fit a different function. We need to decide on a convention for referencing the parameters, and once we do that, we're all set to write code.

Most of the code examples to this point in the book used an objective function where the argument to the `Evaluate` method is `p`, a NumPy vector representing a single particle's position in the search space. Let's continue that tradition here, so we know that the parameters are in `p`, and we get at them by subscripting. This lets us define arbitrary functions as strings. For example, above, we showed a fit to a quadratic. The actual string passed to the implementation was,

```
p[0]*x**2 + p[1]*x + p[2]
```

to represent ax^2+bx+c with `p[0]` for a, `p[1]` for b, and `p[2]` for c. This is how we'll pass fit functions to the code. When this string is evaluated in the `Evaluate` method of the objective class, it's interpreted in the context of that method and returns the function value we seek for the current value of `p`. Note, if the function uses any math functions like `sqrt` or `exp`, we need to use the NumPy versions, `np.sqrt` and `np.exp`, as the arguments are vectors, not scalars.

We're going to pass the fit function as a string. We need to pass the data to be fit as well. Let's combine everything into one text file. The first line is the function to fit. The remaining lines will be the dataset points as y followed by x. For the quadratic example, the input file (`sample.txt`) is,

```
p[0]*x**2 + p[1]*x + p[2]
2.9974516653492604  1.0000000000000000
3.1367763014372589  1.2222222222222223
2.7258038027190272  1.4444444444444444
3.0645761653322778  1.6666666666666665
3.5985181506233666  1.8888888888888888
3.6291980453275348  2.1111111111111112
4.5033314937754909  2.3333333333333330
5.1992874806966309  2.5555555555555554
6.6311609175123172  2.7777777777777777
7.4920293273327463  3.0000000000000000
```

We chose this format because it's compact and because it matches the format used by the NIST curve fitting test functions used below.[1]

Everything's in place; let's build the code. We're using the existing framework's `Bounds` class, so we need only create the custom `Objective` class. The rest of the code uses existing framework pieces. All of the code is in the file `curves.py`.

The objective function class is,

```
class Objective:
    def __init__(self, x, y, func):
        self.x = x
        self.y = y
        self.func = func
        self.fcount = 0
    def Evaluate(self, p):
        self.fcount += 1
        x = self.x
        y = eval(self.func)
        return ((y - self.y)**2).mean()
```

The constructor keeps the data points to be fit in `x` and `y`. These are NumPy vectors. The third argument to the constructor is `func`. It's the string containing Python code to implement the function to be fit. The arguments are stored, and the objective function counter (`fcount`) is set to zero.

The `Evaluate` method bumps the count and then sets the local variable `x` to the fit data. Doing this lets us refer to `x` in the function string. Next, `eval` calculates the set of

[1] NIST is the "National Institute of Standards and Technology", a division of the U.S. Department of Commerce. NIST is deeply involved with standards for all manner of things, including nonlinear curve fitting routines.

output values, `y`. These are the function values, using the parameters in `p`, for the given x positions. It's the squared difference between these y values and those of the fit data we seek to minimize.

The `curves` script accepts many arguments on the command line. Instead of proper parsing, we'll keep it simple and specify the ordering as,

```
curves <data> <lower> <upper> <ndim> <npart> <niter> <tol> <alg> RI|SI|QI
       <plot> <output>
```

Here, `<data>` is a pathname to a text file containing the fit function and the fit points, like the one shown above for the quadratic example. The `<lower>` and `<upper>` arguments are the lower and upper parameter bounds. These are strings with `x` separating the values. We'll see examples below to clarify the syntax. Next comes the number of dimensions, the number of parameters in the fit function, and the desired number of particles. The `<niter>` argument sets the swarm iteration limit. Use `<tol>` to set a tolerance value. We put this in the framework but have not used it until now. If the swarm best objective function value falls below this threshold, stop the search early. Use `<alg>` to specify the swarm algorithm name. Next comes the initializer type. We'll stick with the random initializer (`RI`), but please explore the other options on your own.

The final two arguments are for output. If present, `<plot>` is the file name for an output plot showing the fit data and the swarm's function. We'll see these plots below. Use `<output>` to specify the name of a Python pickle file to contain the results of the swarm search.

As an example, here is the command line that generated Figure 8.3,

```
> python3 curves.py sample.txt -10 10 3 20 1000 1e-7 DE RI
```

This includes the `sample.txt` file shown above. The lower and upper bounds are given as a single value to use the same value for all dimensions. If different ranges are needed for each dimension, separate the bounds with an `x`. Here there are three parameters, each bounded by $[-10, 10]$. The swarm has 20 particles and iterates 1000 times or until the MSE is below 10^{-7}. The DE algorithm is used and initialized randomly.

Before presenting the rest of `curves.py`, we need to define two helper functions. The first reads the data file with the fit function and the data points. The second parses the bounds,

```
def GetData(s):
    lines = [i[:-1] for i in open(s)]
    func = lines[0]
    d = np.zeros((len(lines[1:]),2))
    for i in range(1,len(lines)):
        d[i-1,:] = [float(k) for k in lines[i].split()]
    return d[:,1], d[:,0], func

def GetBounds(s,ndim):
    if (s.find("x") == -1):
        try:
            n = np.ones(ndim)*float(s)
        except:
            n = None
    else:
        try:
            n = np.array([float(i) for i in s.split("x")])
```

```
        except:
            n = None
    return n
```

The `GetData` function reads the entire text file, keeps the first line as the string representing the fit function, and splits the remaining lines, which are (y, x) pairs. The return value is a list of x, y, and the fit function.

The `GetBounds` function interprets `s`, a lower or upper bound string from the command line. If the string does not contain a lowercase "x", the single value is used for all dimensions. Otherwise, the string is split to set per dimension limits. If there is an issue in converting the boundary string, `None` is returned.

The rest of the main function in `curves.py` starts with,

```
X,Y,func = GetData(sys.argv[1])
ndim = int(sys.argv[4])
lower = GetBounds(sys.argv[2], ndim)
upper = GetBounds(sys.argv[3], ndim)
npart = int(sys.argv[5])
niter = int(sys.argv[6])
tol = float(sys.argv[7])
alg = sys.argv[8].upper()
itype = sys.argv[9].upper()

if (type(lower) is type(None)) and (type(upper) is type(None)):
    b = None
else:
    b = Bounds(lower, upper, enforce="resample")

if (itype == "QI"):
    i = QuasirandomInitializer(npart, ndim, bounds=b)
elif (itype == "SI"):
    i = SphereInitializer(npart, ndim, bounds=b)
else:
    i = RandomInitializer(npart, ndim, bounds=b)

obj = Objective(X, Y, func)
```

This format has become familiar: parse the command line, set up the bounds using defaults if `GetBounds` returned `None`, and create the desired initializer object. Finally, create an instance of the `Objective` class passing the data points and fit function string. Notice, this time, we explicitly set the tolerance to the value given on the command line.

Next, we define the desired swarm object passing in the necessary parameters. For PSO, we use the default inertia schedule,

```
swarm= PSO(obj=obj,npart=npart, ndim=ndim, init=i, tol=tol, max_iter=niter,
           bounds=b, inertia=LinearInertia())
```

The other swarm algorithms are initialized similarly.

The actual optimization and reporting of results is straightforward,

```
st = time.time()
swarm.Optimize()
en = time.time()

res = swarm.Results()
```

Filename	*Function*	#
`chwirut1.txt`	$\exp(-p_0 x)/(p_1 + p_2 x)$	3
`eckerle4.txt`	$\frac{p_0}{p_1}\exp(-0.5(x-p_2)^2/p_1^2)$	3
`ENSO.txt`	$p_0 + p_1\cos\left(\frac{2\pi x}{12}\right) + p_2\sin\left(\frac{2\pi x}{12}\right) + p_4\cos\left(\frac{2\pi x}{p_3}\right) + p_5\sin\left(\frac{2\pi x}{p_3}\right)$ $+p_7\cos\left(\frac{2\pi x}{p_6}\right) + p_8\sin\left(\frac{2\pi x}{p_6}\right)$	9
`gauss1.txt`	$p_0\exp(-p_1 x) + p_2\exp(-(x-p_3)^2/p_4^2) + p_5\exp(-(x-p_6)^2/p_7^2)$	8
`gauss2.txt`	$p_0\exp(-p_1 x) + p_2\exp(-(x-p_3)^2/p_4^2) + p_5\exp(-(x-p_6)^2/p_7^2)$	8
`hahn1.txt`	$(p_0 + p_1 x + p_2 x^2 + p_3 x^3)/(1 + p_4 x + p_5 x^2 + p_6 x^3)$	7
`thurber.txt`	$(p_0 + p_1 x + p_2 x^2 + p_3 x^3)/(1 + p_4 x + p_5 x^2 + p_6 x^3)$	7
`sinexp.txt`	$p_0\sin(p_1 x) + p_2\exp(-0.5(x-p_3)^2/p_4)$	5

Table 8.3: The NIST test functions along with the data file name and number of parameters.

```
print("Minimum mean total squared error: %0.9f  (%s)" %
      (res["gbest"][-1],os.path.basename(sys.argv[1])))
print("Parameters:")
for k,p in enumerate(res["gpos"][-1]):
    print("%2d: %21.16f" % (k,p))
print("(%d best updates, %d function calls, time: %0.3f seconds)" %
      (len(res["gbest"]), swarm.obj.fcount, en-st))
```

The minimum mean squared error is reported, followed by each parameter ending with some information about the search. The `curves.py` file includes code to make the plot, if a plot filename was given, and to store the `res` values in a Python pickle file.

This completes the implementation of `curves.py`. Let's take it for a couple of laps around the block.

8.3.3 The Results

To test the curves code, we'll use the NIST nonlinear curve fitting test functions. These are available from `https://www.itl.nist.gov/div898/strd/` with several examples included on the book website. The functions and the datasets are meant for evaluating nonlinear curve fitting routines and include the certified values to many digits of accuracy. We're not looking to match state-of-the-art curve fitting routines, but we can use the functions to test our swarm approaches. The data files are in the `NIST` subdirectory of the `curves` directory. Each function has two files associated with it. The first is a `.txt` file. This is the file to pass to `curves.py`. It contains the fit function and dataset. The second file is a `.dat` file. It's a text file as well, but one that details the origin of the dataset, the fit function, and the NIST certified parameter values along with two examples of initial guesses. Additionally, the `NIST` subdirectory contains the `sinexp.txt` file. This function is not a NIST standard function, but I've used it over the years as an example.

We'll use the NIST test functions shown in Table 8.3 and fit each one five times with each of our algorithms. The code is in `test_functions.py` in the `curves` directory. This script simply sets up repeated calls to `curves.py` with the appropriate command-line arguments.

The result of immediate interest is the minimum MSE found. Aggregate output of `test_functions.py` for a single run is in the file `test_functions_MSE_results.txt`. For each NIST function and algorithm, the MSE is reported for each of the five runs. In general, the results are more or less

consistent from run to run, with a few instances of one run being wildly different from the others. The mean over the five runs gives us MSE's of,

Function	*RO*	*PSO*	*DE*
chwirut1	11.14246058	**11.14241653**	**11.14241653**
eckerle4	0.01447502	**0.00004181**	**0.00004181**
ENSO	5.88368063	4.70200344	**4.69368920**
gauss1	69.48794461	240.96999499	**5.26328897**
gauss2	281.42404091	285.35262667	**4.99011283**
hahn1	0.44159257	1.36283698	**0.00649352**
sinexp	0.21600792	**0.00000002**	**0.00000002**
thurber	484.73325744	1221.31989747	**152.50562814**

and,

Function	*GWO*	*Jaya*	*GA*
chwirut1	12.36848825	**11.14241653**	608.61817149
eckerle4	0.00004182	**0.00004181**	0.00005333
ENSO	5.33949256	4.69369441	4.92459862
gauss1	5.26402186	68.68690417	24.88613718
gauss2	4.99213618	**4.99011283**	14.66047428
hahn1	0.61034623	0.19537979	22.32650942
sinexp	0.67713391	0.00193188	0.00218510
thurber	283.80189992	173.19974512	3069.89358190

The smallest MSE found for each fit function is in **bold** face. The range of MSE values requires many digits at times. Naturally, in many cases, most are not significant.

We see yet again that no one swarm algorithm rules them all. For the knapsack, Section 8.2, RO and GWO were our star pupils. For curve fitting, DE is the clear winner, though it sometimes shares the victory with PSO and Jaya. Interestingly, Jaya does not find the smallest MSE for the `sinexp` function, which is, judging from the tiny MSE found by DE and PSO, the simplest of the test functions.

How do the parameters found compare to the NIST certified values? We'll look at DE's results on `eckerle4` as a best-case example,

Parameter	*DE*	*NIST certified value*
p_0	1.5543827173	1.5543827178
p_1	4.0888321754	4.0888321754
p_2	451.54121844	451.54121844

Here, p_1 and p_2 match the NIST certified value exactly. For p_1, the DE value was only off in the 10-th decimal.

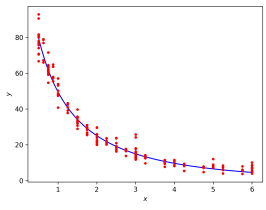
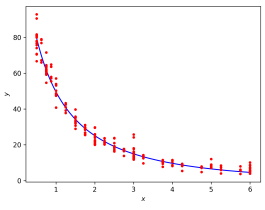
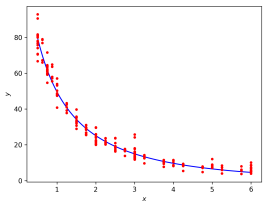
Figure 8.4 through Figure 8.6 show randomly selected fits for each of the NIST test functions and algorithm. In general, the fits are close, but even visually, it's often clear that DE has done the best job overall. In some cases, the fit failed. Still, these examples show the overall utility of using a swarm-based approach to nonlinear curve fitting.

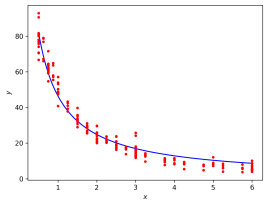
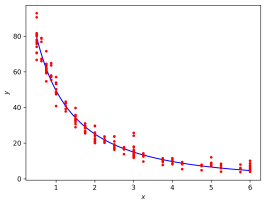
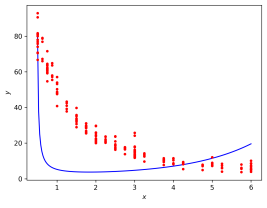
This completes the experiments of this chapter, but before we move on; let's answer the question: what does the search process look like?

The file `curves_plot.py` generates output frames showing the fit function each time the swarm finds a new global best set of parameters.

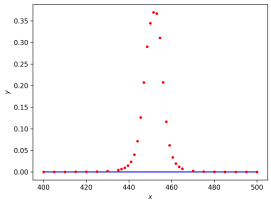
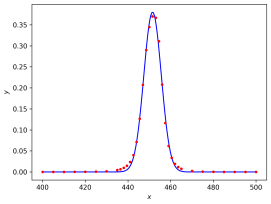
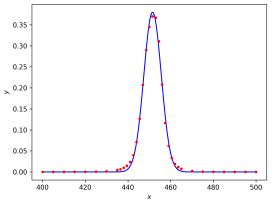
We won't detail the code, it's not relevant to swarm optimization, but we'll show an example of how you might use it. Let's generate frames to watch DE learn the NIST `gauss1` function. First, we create the necessary output files with `curves.py`,

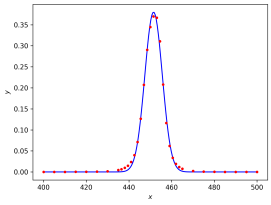
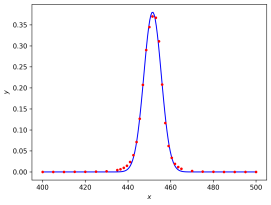
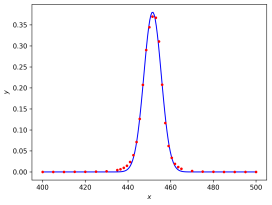
chwirut1:


RO

PSO

DE

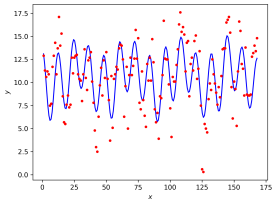
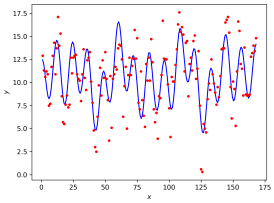
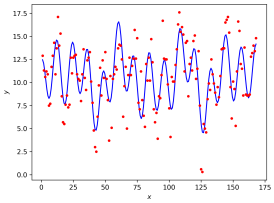

GWO

Jaya

GA

eckerle4:


RO

PSO

DE


GWO

Jaya

GA

ENSO:


RO

PSO

DE

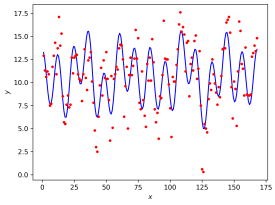
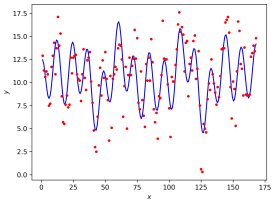
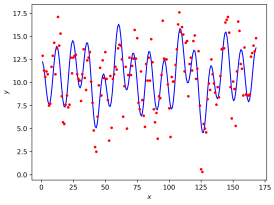
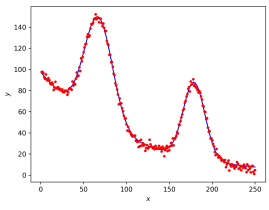
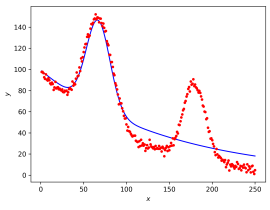
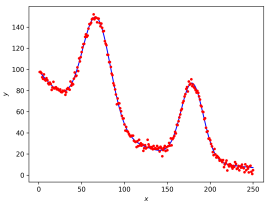
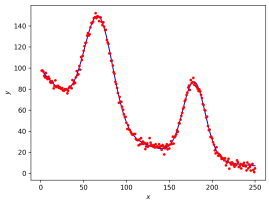
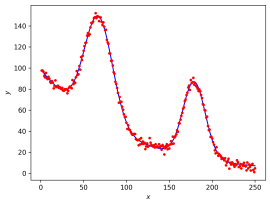
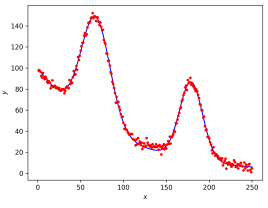

GWO

Jaya

GA

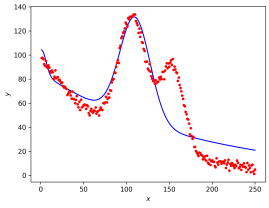
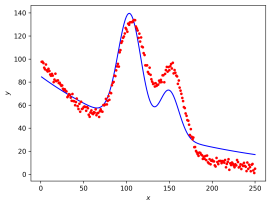
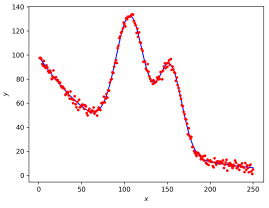
Figure 8.4: Representative fits for the NIST test functions.

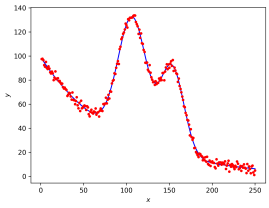
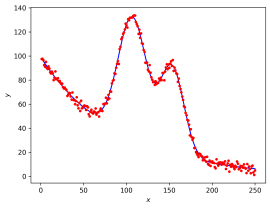
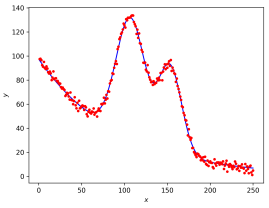
gauss1:


RO

PSO

DE

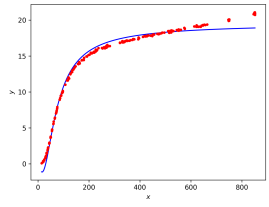
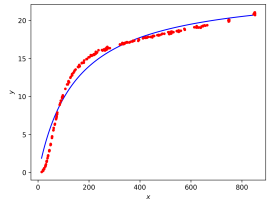
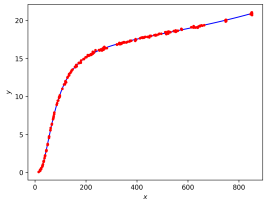

GWO

Jaya

GA

gauss2:


RO

PSO

DE


GWO

Jaya

GA

hahn1:


RO

PSO

DE

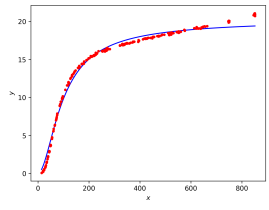
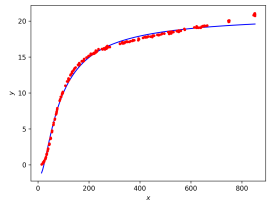
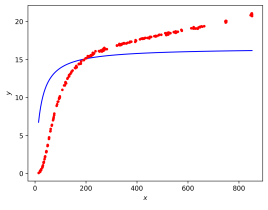

GWO

Jaya

GA

Figure 8.5: Representative fits for the NIST test functions.

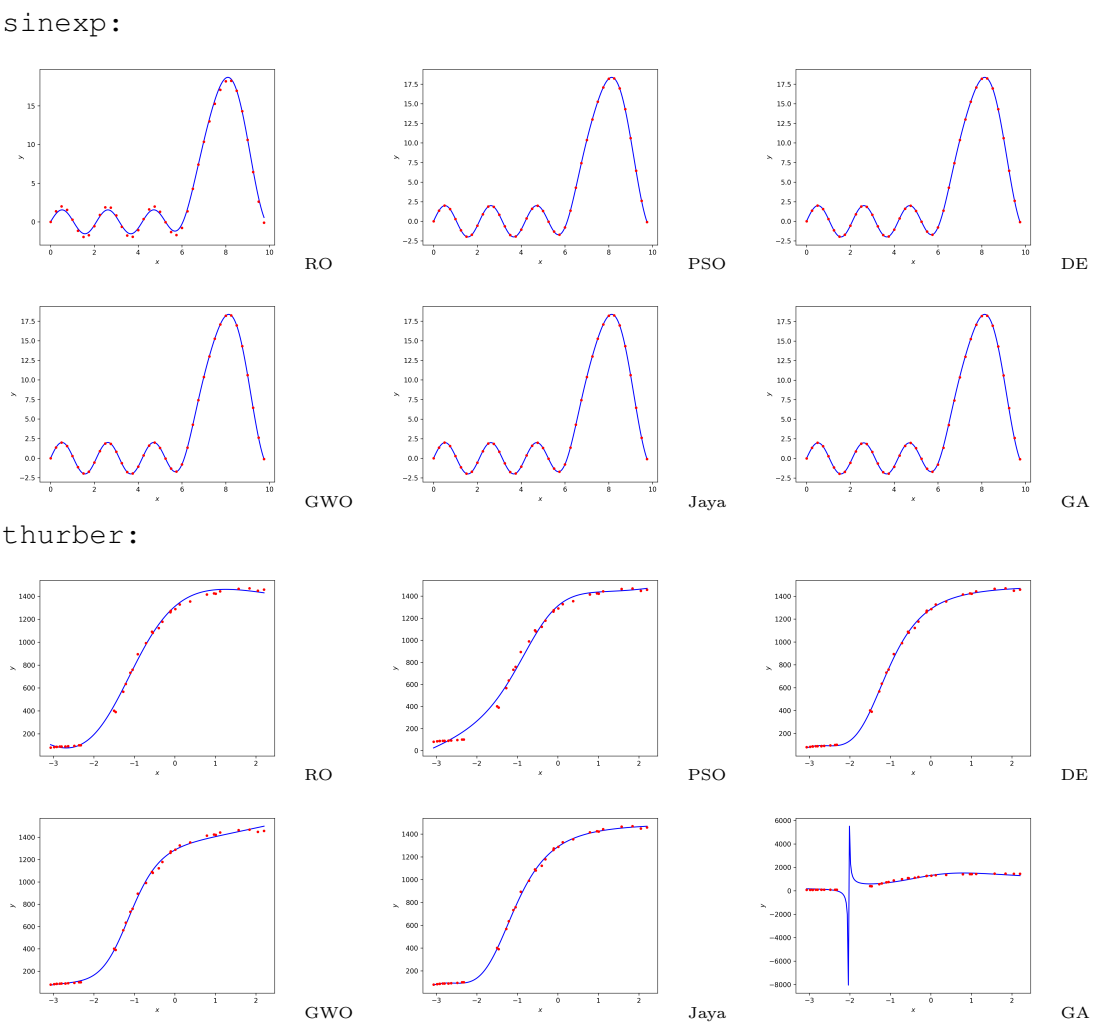


Figure 8.6: Representative fits for the NIST test functions.

```
> python3 curves.py NIST/gauss1.txt 0 100x1x110x70x25x75x200x20 8 20 10000
        1e-7 DE RI gauss1.png gauss1.pkl
```

This creates a plot in `gauss1.png` and stores the search results in `gauss1.pkl`.

Now, generate the frames with,

```
> python3 curves_plot.py gauss1.pkl NIST/gauss1.txt gauss1
```

to create an output directory, `gauss1`, with the frames and a single plot showing the MSE as a function of swarm best updates. For my test run, the swarm converged to a good set of parameters by update 65.

To create an animated GIF from the frames, use ImageMagick,

```
> convert -delay 100 -loop 0 gauss1/frame*.png gauss1.gif
```

to produce `gauss1.gif` thereby animating the search. ImageMagick is typically already installed on most common Linux distributions.

Let's continue our experiments in the next chapter by applying swarms to the training of traditional neural networks.

Chapter 9

Training a Neural Network

In this chapter, we apply swarm optimization to the task of training a traditional neural network. By "traditional" we mean a fully-connected, feedforward model with one or two small hidden layers. Don't worry if none of these terms yet have meaning. We'll describe everything as we go. As before, we present the problem, show how we'll set up the solution using the framework, and then see how we do with several experiments.

9.1 The Problem

A neural network is a model that accepts a vector input and produces an output value interpreted as the likelihood of membership in one of two or more classes. Naturally, neural networks are far more than just this, but we'll restrict ourselves here to traditional, supervised learning with feedforward networks.

Figure 9.1 shows the structure of the kind of network we'll explore. Read the figure from left to right. The boxes on the left are the vector input, the feature vector, here with three elements. Next comes the first hidden layer with four circles. Hidden layers are made of neurons, also called nodes. Each neuron, the circles, receives input from each element of the feature vector and produces a single output value. In this sense, neurons are analogous to biological neurons which accept multiple inputs and produce one output. However, that's where the similarity generally ends. The hidden layer's output becomes the input to the layer to its right, the second hidden layer with three nodes. The process repeats, so this layer's output is the input to the next layer with a single node, the output layer. Therefore, the network accepts a 3-element vector input and produces a single scalar value as output. The scalar value is interpreted as the likelihood of membership in class 1 as opposed to class 0. The figure represents a binary classifier which sorts input feature vectors into class 0 or class 1.

The numeric output is assigned a class label by applying a threshold. If the output is less than the threshold, typically 0.5, the label is "class 0", otherwise it's "class 1". It is straightforward to extend a network like this to produce multiple outputs for multiclass classification.

The power of neural networks comes from the activity of the neurons. For example, the top circle of the first hidden layer in Figure 9.1 receives input from each element of the feature vector, (x_0, x_1, x_2). Each of these values is multiplied by a weight assigned to one of the edges from the input to the node. The weights are not shown, but each line connecting two nodes in the figure has one. Additionally, each node starting with the first hidden layer has an associated bias value, a scalar, also not shown.

The neuron's action is straightforward: multiply each input value by the weight assigned to that edge, sum them, add in the bias value, and pass that as the input to the activation function. The activation function is a nonlinear function producing a scalar output. The scalar outputs of all the nodes in a layer become the input vector to the next layer to the right.

From this description, we see what differentiates one network from another, assuming the layers and nodes per layer are fixed, is the value of the weights and biases. The goal of training a neural network is to find values for the weights and biases that allow the network to produce correct

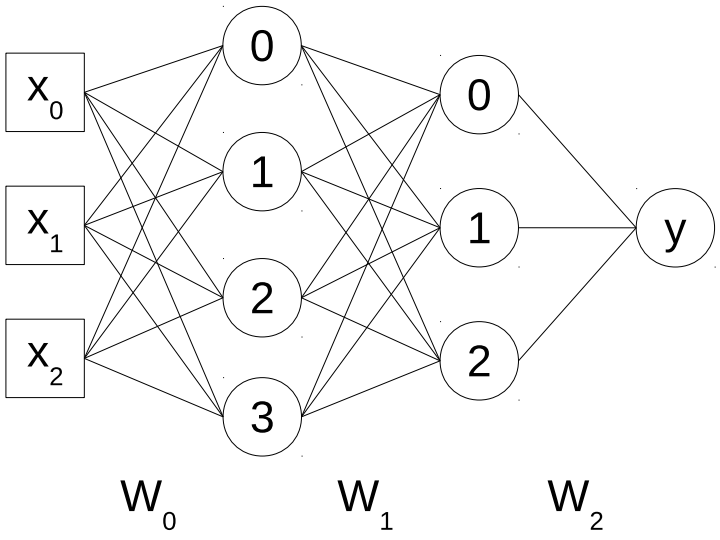


Figure 9.1: A traditional neural network.

classifications. From theory, we know neural networks are universal function approximators, so if there is a function mapping input feature vectors to class label outputs, we should be able to approximate it with a neural network of an appropriate size and with enough example training data. Of course, theory and practice are often wildly different.

As hinted above, mathematically, we represent the input to the network or the output of any network layer, as a vector. A matrix of weights represents the mapping from any layer to the next layer. Therefore, a traditional neural network's entire operation is implemented as a series of matrix/vector multiplications and additions. This makes the implementation particularly straightforward for array-processing libraries like NumPy.

The standard approach to learning the weights and biases, which works equally well for modern deep neural networks, is to use a first-order gradient descent algorithm to update the weights and biases based on propagation of the errors made by the network. Specifically, weights and biases are updated based on the gradient,

$$w_{i+1} \leftarrow w_i - \eta \frac{\partial L}{\partial w}$$

Here, L is a measure of the error made by the network, w is one of the weights of the network, and $\partial L/\partial w$ is how the loss changes for a change in w. The contribution of w to the gradient is multiplied by a scale factor, η, known as the learning rate. The learning rate controls the size of the update from iteration i to iteration $i+1$. Each iteration is a forward pass through the network of the training data to get a measure of the error and a backward pass using the chain rule for derivatives to find the $\partial L/\partial w$ for each w. The backward pass is known as *backpropagation*, and it's foundational to the success of deep learning ([25]).

However, that's not how we'll train our neural networks. Instead, we'll use a swarm search to learn what the weights and biases should be. This method works for small networks but does not scale to networks large enough to be more than only mildly interesting.[1] All the same, one of the early uses for PSO in the mid-1990s was to train neural networks.

Let's call the input vector $\boldsymbol{x}$, the weight matrix between the input and the first hidden layer $\boldsymbol{W}$, and the output of the first hidden layer $\boldsymbol{a}$. We'll also need a bias vector, one bias value for each node

[1]This is not entirely true. The new field of neuroevolution applies evolutionary algorithms to deep neural networks.

in the hidden layer. Let's call that $\boldsymbol{b}$. Finally, we need an activation function. For the moment, we don't care what it is; we'll just call it h and know that it accepts a vector as input and outputs a vector.

With these definitions, the entire operation of the network from the input feature vector to the output of the first hidden layer becomes,

$$\boldsymbol{a}_1 = h(\boldsymbol{W}_0\boldsymbol{x} + \boldsymbol{b}_0) = h\left(\begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \\ w_{30} & w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b3 \end{bmatrix}\right) = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

where $\boldsymbol{W}$, $\boldsymbol{x}$, and $\boldsymbol{b}$ are shown explicitly.

Consider the node marked as "0" in first hidden layer of Figure 9.1. The inputs to this node are each of the feature vector elements, (x_0, x_1, x_2), multiplied by the weights for this node, (w_{00}, w_{01}, w_{02}) and then summed along with the bias value for the node, b_0. The input to the activation function and the output is,

$$a_0 = h(x_0 w_{00} + x_1 w_{01} + x_2 w_{02} + b_0)$$

This is precisely what we get when multiplying the first row of $\boldsymbol{W}$ by $\boldsymbol{x}$ and adding the first element of $\boldsymbol{b}$. Likewise, we get the proper expressions for the remaining nodes of the first hidden layer by using the second through fourth rows of $\boldsymbol{W}$ and elements of $\boldsymbol{b}$.

To complete the flow through the network, the output of the first hidden layer, (a_0, a_1, a_2, a_3), becomes the input to the second hidden layer with three nodes,

$$\boldsymbol{a}_2 = h(\boldsymbol{W}_1\boldsymbol{a}_1 + \boldsymbol{b}_1) = h\left(\begin{bmatrix} w_{00} & w_{01} & w_{02} & w_{03} \\ w_{10} & w_{11} & w_{12} & w_{13} \\ w_{20} & w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}\right) = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$$

and, the final output value is,

$$a_3 = h(\boldsymbol{W}_2\boldsymbol{a}_2 + \boldsymbol{b}_2) = h\left(\begin{bmatrix} w_{00} & w_{01} & w_{02} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} + b_0\right)$$

with a_3 a scalar. We pass a_3 through a sigmoid function to determine the likelihood of $\boldsymbol{x}$ representing an instance of class 1. The sigmoid is,

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

and is bounded so $[-\infty, +\infty] \rightarrow [0, 1]$ with $\sigma(0) = 0.5$. The output of a sigmoid is always in the range $[0, 1]$, so it is natural to talk about it as a probability. To be pedantic, it's better to call the output a likelihood as most neural networks are not calibrated. A calibrated network is one where say 80% of the inputs with output 0.8 are actually class 1 and so on. However, the loose terminology is seldom an issue.

The series of expressions above tell us how to move data through a traditional neural network. The same steps may be expressed in matrix/vector form quite succinctly,

$$\begin{aligned} \boldsymbol{a}_1 &= h(\boldsymbol{W}_0\boldsymbol{x} + \boldsymbol{b}_0) \\ \boldsymbol{a}_2 &= h(\boldsymbol{W}_1\boldsymbol{a}_1 + \boldsymbol{b}_1) \\ a_3 &= \boldsymbol{W}_2\boldsymbol{a}_2 + b_2 \\ p &= \frac{1}{1 + e^{-a_3}} \end{aligned}$$

with p the probability of $\boldsymbol{x}$ representing an instance of class 1. We can think of the sigmoid as the activation function for the output layer.

We have yet to explicitly define the hidden layer activation function, h. We have several options, but the current favorite of the deep learning community is the *rectified linear unit* or *relu* (sometimes $ReLU$). The relu is,

$$\text{relu}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

and it's the activation function we'll use for our experiments below.

The equations above tell us how to pass a feature vector through the network to decide class membership. We'll use essentially this process when learning the weights and biases, the elements of the different $\boldsymbol{W}$ matrices and $\boldsymbol{b}$ vectors.

The simple network of Figure 9.1 has a total of 35 parameters to learn: 27 weights and 8 biases. If we want to use a swarm to find them, we need to search a 35-dimensional space. The actual network we'll experiment with is larger than this.

Our description of neural networks and training has been necessarily terse. We're only providing enough background to frame the problem we want to solve.[2]

Let's see how to morph the problem of learning the weights and biases for a particular classification task and architecture into a swarm optimization task.

9.2 The Setup

Before we can set up our swarm solution, we need to install a new Python library. The scikit-learn library provides a cornucopia of machine learning algorithms and tools, including support for traditional machine learning via the `MLPClassifier` class. We'll use this class to handle data processing for us and give us a performance baseline against which we can compare our swarm results.

To install scikit-learn on Ubuntu use,

```
$ sudo pip3 install scikit-learn
```

and to test it,

```
$ python3
>>> import sklearn
>>> sklearn.__version__
'0.20.3'
```

where your version should be at least the version shown.

We'll use the `MLPClassifier` class with our swarms by replacing the weights and biases it would use with those derived from the particle positions. That way, we need not write code to pass data through the model, and we can use the prediction methods to give us results.

We already know we'll use the particle position vector to represent the weights and biases we need for the network. Each particle position gives us a candidate set. We also need an objective function to tell us that the weights and biases a particle represents are good or bad. Here's where we use the training data. It already has labels, so we'll use the network's performance on the training set, using the current particle's position to set the weights and biases, to calculate a measure of how well the model is doing.

The steps we need are,

[2] To learn more about machine learning, and deep learning in particular, please see my books "Practical Deep Learning with Python: A Hands-On Introduction" (No Starch Press, January 2021) and "Math for Deep Learning" (No Starch Press, October 2021).

1. Initialize an instance of the `MLPClassifier` class with a selected number of layers and nodes per layer. We'll use the relu for the activation function.
2. For each particle position, replace the weights and biases of the `MLPClassifier` instance with the particle position components via a consistent mapping from component to weights and biases by layer.
3. Call the `predict` method on the model instance using the training data to generate output class label predictions.
4. Compare the model's predicted class to the actual known class to generate a measure of how well the model has done. This gives a metric we can use to compare the swarm positions.
5. Use the metric as the objective function value. The smaller the metric, the better the model is doing with that particular set of weights and biases.
6. Update the swarm positions according to the particular swarm algorithm in use and repeat for the desired number of iterations or until a set of weights or biases is found that gives the desired level of performance on the training data.

We'll clarify these steps as we walk through them; don't be concerned if they are fuzzy right now. We have another preliminary task before we dive into code. We need a problem to solve, a classifier to make. The classifier we'll make is designed to take a feature vector representing measurements derived from a histology slide and decide whether the feature vector represents a malignant or benign breast cancer case. The dataset is small and easily loaded by sklearn using the following code,

```
import numpy as np
from sklearn.datasets import load_breast_cancer

x,y = load_breast_cancer(True)
x = (x - x.mean(axis=0)) / x.std(axis=0)
i0 = np.where(y == 0)
i1 = np.where(y == 1)
yp = y[i1];  yn = y[i0]
xp = x[i1];  xn = x[i0]
pp = int(0.7*len(yp))
nn = int(0.7*len(yn))
xtrn = np.concatenate((xp[:pp,:], xn[:nn,:]))
ytrn = np.concatenate((yp[:pp], yn[:nn]))
xtst = np.concatenate((xp[pp:,:], xn[nn:,:]))
ytst = np.concatenate((yp[pp:], yn[nn:]))
idx = np.argsort(np.random.random(len(ytrn)))
xtrn = xtrn[idx]
ytrn = ytrn[idx]
idx = np.argsort(np.random.random(len(ytst)))
xtst = xtst[idx]
ytst = ytst[idx]

np.save("nn_xtrn.npy", xtrn)
np.save("nn_ytrn.npy", ytrn)
np.save("nn_xtst.npy", xtst)
np.save("nn_ytst.npy", ytst)
```

You'll find in the `download_dataset.py` script in the `nn` directory. The first time you run this code, sklearn will download the dataset it needs, subsequent runs will use the already downloaded data.

The code creates four output NumPy files. Two of the files are feature vectors, and two are class labels associated with those feature vectors. One set of files is training data; the data used to condition the model, especially when using backpropagation and gradient descent for training, as

we'll do to get our baseline model. The remaining set is test data. Neural networks are typically tested after they are fully trained with a set of data never used during training. This is the test set, and it gives us an unbiased measure, compared to the training data, of how well the model has learned. Of all the available breast cancer data, we'll use 70% of it for training and hold the remaining 30% back for testing. Going forward, we'll use the NumPy files.

The feature vectors are 30-dimensional; each input to our network is a vector of 30 elements representing the histology slide measurements. We don't know what the measurements represent, nor does any neural network. Instead, we'll use the measurements and the label as they are. We need to define our neural network architecture. We'll use a single hidden layer with 60 nodes, meaning our network has 30 inputs, 60 hidden nodes, and 1 output. This works out to 1921 weights and biases. Therefore, our problem size is fixed, we're working with particles in a 1921-dimensional space, far larger than any space we've used before. Are our swarm algorithms up to the task? We'll soon see.

Now we can write code. All of the code presented below is in the file `nn.py`. We start at the beginning of the main portion of the script. We'll skip the particular module imports and jump to parsing the command line,

```
npart = int(sys.argv[1])
niter = int(sys.argv[2])
ndim = 1921 # params in a 30 > 60 > 1 MLP
itype = sys.argv[3].upper()

xtrn = np.load("nn_xtrn.npy")
ytrn = np.load("nn_ytrn.npy")
xtst = np.load("nn_xtst.npy")
ytst = np.load("nn_ytst.npy")

MCC = []; M = []
TP = []; TN = []
FP = []; FN = []
T = []; SC = []

nnodes = 60
clf = MLPClassifier(hidden_layer_sizes=(nnodes,),
        solver="lbfgs", max_iter=3000, tol=0)
st = time.time()
clf.fit(xtrn, ytrn)
en = time.time()
mlp_prob = clf.predict_proba(xtst)
mlp_score = clf.score(xtst, ytst)
tn,fp,fn,tp,mcc = confmat(mlp_prob, ytst)
TP.append(tp); TN.append(tn); FP.append(fp); FN.append(fn);
T.append(en-st); SC.append(mlp_score)
MCC.append(mcc)
M.append("MLP")
```

The command line accepts a swarm size and a maximum number of iterations followed by the initializer type. For this experiment, we'll work with the random, sphere, and quasirandom initializers. Next, the train and test datasets are loaded into `xtrn` and `xtst` respectively.

As we train different models using the other algorithms, we'll track a series of metrics, so we set up empty lists for those next.

We need a baseline model. That model is the performance of a traditional neural network on this same dataset. Another name for a traditional neural network is a "multilayer perceptron" or MLP, which is how we're referring to it in code. We use the `MLPClassifier` class to create an instance of a neural network. We then train it using the training data by calling the `fit` method. The sklearn library handles all data, backpropagation, and gradient descent tasks for us. Training is this one line of code. The `predict_proba` method returns the output probability for the test data. This is the likelihood of a test feature vector being a member of class 1 (malignant). We also

use `score` to compute the accuracy. This is the fraction of correctly assigned class labels. The `confmat` function, which takes the model output probabilities and known test set labels (`ytst`), will be described shortly. Its goal is to generate metrics for evaluating the model's performance with its current set of weights and biases. Finally, we store the different metrics for this model.

Let's examine `confmat` to see what it's calculating for us,

```
def confmat(prob, ytst):
    p = np.argmax(prob, axis=1)
    tp=tn=fp=fn=0
    for i in range(len(ytst)):
        if (ytst[i]==1) and (p[i]==1):
            tp += 1
        if (ytst[i]==1) and (p[i]==0):
            fn += 1
        if (ytst[i]==0) and (p[i]==1):
            fp += 1
        if (ytst[i]==0) and (p[i]==0):
            tn += 1
    d = np.sqrt((tp+fp)*(tp+fn)*(tn+fp)*(tn+fn))
    if (d != 0):
        mcc = (tp*tn - fp*fn) / d
    else:
        mcc = 0.0
    return tn,fp,fn,tp,mcc
```

This function creates a 2x2 confusion matrix. The confusion matrix is a table showing counts for the number of times: a class 1 instance was called class 1 (true positive, TP), a class 0 instance was called class 0 (true negative, TN), a class 1 instance was called class 0 (false negative, FN), and a class 0 instance was called class 1 (false positive, FP). A perfect model has FN = FP = 0; it makes no mistakes.

With these primary tallies, we can calculate many possible metrics. The one we've chosen for ranking the final model performance ranking is Matthews Correlation Coefficient (MCC). This metric has a maximum value of 1.0 when the model is perfect. The `confmat` function returns TN, FP, FN, TP, and MCC.

At this point in `nn.py`, we have our datasets loaded and have built, trained, and tested our baseline MLP classifier. Let's define the swarm objective to see how we'll transform a particle position into a neural network. We know we need the training data as part of the objective, just like we needed the dataset for the curve fitting experiments. We also need a dummy instance of `MLPClassifier` we can use to update the weights and biases. Therefore, the setup for the objective function is

```
snn = MLPClassifier(hidden_layer_sizes=(nnodes,), max_iter=1)
snn.fit(xtrn,ytrn)
obj = SwarmObjective(snn,xtrn,ytrn)
```

Here, `snn` is our dummy instance of `MLPClassifier`. We need to train the instance by calling `fit` so structures internal to the object are in the proper state. We don't care about how well it did, we're going to overwrite the weights and biases anyway. With `snn` in a proper state, we define our objective function instance. Here's what the `SwarmObjective` class looks like,

```
class SwarmObjective:
    def __init__(self, snn, xtrn, ytrn):
        self.snn = snn
        self.xtrn = xtrn
        self.ytrn = ytrn
    def Evaluate(self, weights):
        self.snn.coefs_[0] = weights[:1800].reshape((30,60))
```

```
        self.snn.coefs_[1] = weights[1800:1860].reshape((60,1))
        self.snn.intercepts_[0] = weights[1860:1920]
        self.snn.intercepts_[1] = weights[1920]
        return 1.0 - self.snn.score(self.xtrn, self.ytrn)
```

The constructor accepts the dummy `MLPClassifier` instance (`snn`), the training feature vectors (`xtrn`), and associated labels (`ytrn`). The constructor stores these for use by `Evaluate`.

The `Evaluate` method accepts a particle position, which in this case we're calling `weights`. It's a vector of 1921 elements. Our goal is to copy the weights into the proper places in the dummy `MLPClassifier` instance and then calculate the training set's accuracy. To do this, we require some knowledge of the internals of the `MLPClassifier` class. Specifically, we need to know how it stores the weights and biases. A review of the relevant scikit-learn documentation tells us that weight matrices are stored in `coefs_` and biases are stored in `intercepts_`. The network has a single hidden layer, meaning we need one weight matrix between the input and the hidden layer and another between the hidden layer and the output layer. Also, the hidden layer and the output layer both have bias vectors. For the hidden layer, there are 60 elements in the bias vector because there are 60 nodes. For the output layer, there is a single, scalar value. The first weight matrix maps the 30-element input to a 60-element output, so it's a 30x60 matrix. The second weight matrix maps 60 elements to 1 and is a 60x1 matrix.

The mapping from the 1921-dimensional particle position to the weight matrices is arbitrary, but we need to be consistent. The code maps the first 1800 to the first weight matrix, the next 60 to the second weight matrix, the following 60 to the bias vector for the hidden layer, the last element to the bias for the output layer.

With the weight and biases set, a call to `score` using the training data evaluates the model returning the fraction of the training data that were correctly classified. Since we always minimize, and the maximum accuracy of the model is 1.0, we subtract this number from 1.0 to return an objective function value. As this dataset is reasonably balanced, meaning the number of benign and malignant cases is more or less equal, we'll use the score as our metric during optimization. If the dataset were highly imbalanced with one class was far more common than the other, we'd want another metric here, perhaps the MCC. The reason why is understood by considering a training set that is 99% class 0 and 1% class 1. If the overall accuracy is our metric, a model predicting class 0 for all inputs will be 99% accurate. Clearly, this is not good. Note, we'll use the accuracy to locate the best set of weights and biases, but use MCC to rank the final models against each other.

As before, the swarm is randomly initialized, but we need to be careful in this case. We have two different bounds we want to respect. The first has to do with the initial set of weights and biases, which we want bound to a small range, and the second has to do with the actual limit on weights and biases that we want to allow after searching for some period of time. We do this because it is known that neural networks initialized with large values do not learn well. This effect may be due to the use of backpropagation and gradient descent and not even apply to what we're attempting to do here, but without any prior knowledge of what the initialization values should be, we choose to keep them small. Feel free to experiment with other initialization schemes. In code, the next lines of the script are,

```
b = Bounds(-0.01*np.ones(ndim), 0.01*np.ones(ndim), enforce="resample")
if (itype == "SI"):
    i = SphereInitializer(npart, ndim, bounds=b)
elif (itype == "QI"):
    i = QuasirandomInitializer(npart, ndim, bounds=b)
else:
    i = RandomInitializer(npart, ndim, bounds=b)
b = Bounds(-10*np.ones(ndim), 10*np.ones(ndim), enforce="resample")
```

We initially set `b` to an instance of our Bounds class limited to the range $[-0.01, 0.01]$. This Bounds object is used to set up the desired swarm initializer (passed in on the command line). Here `ndim` is 1921. Once the initializer object is created, we want to allow the swarms to search over a larger

region. Ad hoc examination of several similar neural networks tells us that final learned weights and biases, for similar datasets, are seldom outside the range $[-10, 10]$, so these are the limits we'll use with the actual swarm objects. Therefore, we redefine `b` to be a Bounds object in that range, with resampling when bounds are exceeded.

Everything is now in place to train the network using the different swarm algorithms. We'll loop over the algorithm names and create the proper swarm object,

```
for alg in ["RO","PSO","DE","GWO","JAYA","GA"]:
    if (alg == "RO"):
        swarm = RO(obj=obj, npart=npart, ndim=ndim, max_iter=niter,
                   init=i, bounds=b)
    elif (alg == "PSO"):
        swarm = PSO(obj=obj, npart=npart, ndim=ndim, max_iter=niter,
                   init=i, bounds=b, inertia=LinearInertia())
    elif (alg == "DE"):
        swarm = DE(obj=obj, npart=npart, ndim=ndim, max_iter=niter,
                   init=i, bounds=b)
    elif (alg == "GWO"):
        swarm = GWO(obj=obj, npart=npart, ndim=ndim, max_iter=niter,
                   init=i, bounds=b)
    elif (alg == "JAYA"):
        swarm = Jaya(obj=obj, npart=npart, ndim=ndim, max_iter=niter,
                   init=i, bounds=b)
    elif (alg == "GA"):
        swarm = GA(obj=obj, npart=npart, ndim=ndim, max_iter=niter,
                   init=i, bounds=b)

    st = time.time()
    swarm.Optimize()
    en = time.time()

    SetWeights(obj.snn, swarm.gpos[-1])
    prob = obj.snn.predict_proba(xtst)
    score= obj.snn.score(xtst, ytst)

    tn,fp,fn,tp,mcc = confmat(prob, ytst)
    TP.append(tp); TN.append(tn); FP.append(fp);
    FN.append(fn); T.append(en-st); SC.append(score)
    MCC.append(mcc)
    M.append(alg)
```

Once we have the `swarm` object, we optimize while tracking the time it takes. Next, we call `SetWeights`, which we'll see below, to place the best set of weights in the dummy `MLPClassifier` object, `snn`, borrowing its reference from the `SwarmObjective` instance. Then, calls to `predict_proba` and `score` give us the output probabilities and the test set's accuracy. Finally, we calculate the same set of statistics we found initially for the MLP and move on to the next swarm algorithm.

The `SetWeights` method does essentially what we did inside of the objective function: it copies the weights from a particle position, here the swarm best, to the `MLPClassifier` instance,

```
def SetWeights(snn, weights):
    snn.coefs_[0] = weights[:1800].reshape((30,60))
    snn.coefs_[1] = weights[1800:1860].reshape((60,1))
    snn.intercepts_[0] = weights[1860:1920]
    snn.intercepts_[1] = weights[1920]
```

When every swarm algorithm has been tested, we rank the results by the Matthews Correlation Coefficient calculated on the test set and display them in decreasing order. Recall, the closer the MCC is to 1.0, the better the model is performing,

```
MCC = np.array(MCC)
idx = np.argsort(1.0-MCC)
MCC = MCC[idx]
M = np.array(M)[idx]
TP = np.array(TP)[idx]
TN = np.array(TN)[idx]
FP = np.array(FP)[idx]
FN = np.array(FN)[idx]
T = np.array(T)[idx]
SC = np.array(SC)[idx]

print("Ranked: (npart=%d, niter=%d)" % (npart, niter))
print("  MCC       Score       TP   FP   FN   TN     time")
for i in range(len(M)):
    print("%10.6f  %0.6f  %4d %4d %4d %4d %8.3f  %s" %
        (MCC[i],SC[i],TP[i],FP[i],FN[i],TN[i],T[i],M[i]))
```

The code is now in place. Let's run it for different combinations of swarm size, iterations, and swarm initialization to see how this approach to training a neural network compares to backpropagation and gradient descent.

9.3 The Results

A single run of `nn.py` looks like this at the command line,

```
> python3 -W ignore nn.py 30 1000 RI
```

We explicitly silence runtime warnings because the sklearn library raises them when the model hasn't converged according to the training heuristics. It's safe to ignore those warnings. The command uses a swarm of 30 particles for 1000 iterations and random initialization. The output produced is,

```
Ranked: (npart=30, niter=1000)
  MCC       Score       TP   FP   FN   TN     time
  0.987673  0.994186   107    0    1   64  126.168  DE
  0.929243  0.965116   102    0    6   64    0.176  MLP
  0.915988  0.959302   102    1    6   63   70.277  JAYA
  0.913252  0.959302   104    3    4   61   68.451  GA
  0.902805  0.953488   102    2    6   62   71.383  PSO
  0.891448  0.947674   101    2    7   62   76.840  GWO
  0.431914  0.738372    88   25   20   39   73.862  RO
```

telling us that for this run, DE produced the best performing model with an MCC of 0.987 and overall accuracy of 99.4%. The raw TP, FP, FN, TN counts are also given. Recall, the results are for the held-out test set. The model trained by DE made a single mistake; it called a single malignant case benign. We see from the reported training times that the swarm approach is not exceptionally fast, nor is it optimized in any way. The sklearn library is optimized for performance, as we'd expect from a widely-used piece of open-source software.

We see that DE beat the MLP, the model trained with traditional gradient descent. That model had 6 false negatives. Lastly, all the remaining swarm algorithms, except RO, performed reasonably well by not falling apart. RO, on the other hand, had a low MCC and overall accuracy.

We know that swarm optimization is stochastic because of the swarm initialization process, let alone randomness in many of the swarm update rules. And, we discussed above how traditional training of neural networks is also stochastic. Therefore, we shouldn't read too much into the results of a single run. The results might change, perhaps dramatically, if we were to run the same set of parameters a second time.

Let's set up a script to run each swarm size, iteration limit, and initialization type six times. Then, we'll develop code to summarize the multiple runs and thereby, hopefully, reach some consensus about how well the swarm algorithms perform on this task. The script is simply a list of repeated command lines that capture the output in files with names like `nn_30_1000_RI_run1.txt`. Running the script requires some patience, nearly two day's worth. When done, however, it has created output files for six runs each of the following combinations of parameters,

Size	*Iterations*	*Initializer*
30	1000	RI, QI, SI
30	3000	RI, QI, SI
100	1000	RI, QI, SI
100	3000	RI, QI, SI

for a total of 12 combinations. A bit of code,

```
def GetRanking(fname):
    lines = [i[:-1] for i in open(fname)]
    lines = lines[3:]
    ranks = []
    for line in lines:
        try:
            ranks.append(line.split()[-1])
        except:
            pass
    return ranks

ranks = {
    "MLP": [], "RO": [], "PSO": [],
    "DE": [], "GWO": [], "JAYA": [],
    "GA": [],
}

for i in range(1, len(sys.argv)):
    order = GetRanking(sys.argv[i])
    for i,label in enumerate(order):
        ranks[label].append(i)

for k in ["MLP","RO","PSO","DE","GWO","JAYA","GA"]:
    r = np.array(ranks[k])
    print("%4s: %0.3f +/- %0.3f, " % (k, r.mean(), r.std(ddof=1)),
          r.astype("uint16"))
```

parses the output files passed on the command line to produce a summary across the six runs.

Table 9.1 shows the rankings across the six runs for a swarm with 30 particles. On the left are the results for 1000 iterations and on the right for 3000 iterations. The set of six numbers on the right column of each subtable lists the per-run rankings for that algorithm.

What to make of these results? Two statements are easy to make: the traditional MLP is still the best performing model and the RO-trained model is always the worst. Another way to rank the results, for both 1000 and 3000 iterations together, is to tally the number of times each algorithm appears in the top three across iterations and initialization strategies. Doing this gives us an overall ranking of,

	MLP:	0.833 ± 1.169	[1 0 0 1 3 0]	MLP:	0.833 ± 0.983	[0 2 0 0 1 2]
	DE:	1.667 ± 1.862	[0 1 2 0 5 2]	GWO:	1.833 ± 1.722	[3 0 1 3 0 4]
	Jaya:	2.333 ± 1.633	[2 2 5 2 0 3]	PSO:	2.667 ± 1.506	[2 1 4 2 2 5]
RI:	PSO:	3.000 ± 1.414	[4 3 1 3 2 5]	DE:	3.000 ± 1.897	[4 5 2 1 5 1]
	GA:	3.500 ± 1.378	[3 5 4 4 1 4]	GA:	3.000 ± 2.098	[1 4 5 5 3 0]
	GWO:	3.667 ± 1.506	[5 4 3 5 4 1]	Jaya:	3.667 ± 0.816	[5 3 3 4 4 3]
	RO:	6.000 ± 0.000	[6 6 6 6 6 6]	RO:	6.000 ± 0.000	[6 6 6 6 6 6]
	MLP:	0.833 ± 1.169	[0 0 3 1 0 1]	MLP:	1.000 ± 0.632	[1 1 2 0 1 1]
	PSO:	2.167 ± 1.602	[2 1 2 4 4 0]	PSO:	1.833 ± 1.722	[0 5 1 1 2 2]
	DE:	2.333 ± 1.506	[1 2 5 2 1 3]	GWO:	2.500 ± 2.258	[2 0 3 5 0 5]
QI:	GWO:	2.500 ± 2.345	[4 3 0 0 2 6]	DE:	3.000 ± 1.549	[4 3 0 3 4 4]
	Jaya:	2.833 ± 1.329	[3 5 1 3 3 2]	GA:	3.167 ± 1.941	[5 4 5 2 3 0]
	GA:	4.500 ± 0.548	[5 4 4 5 5 4]	Jaya:	3.500 ± 1.049	[3 2 4 4 5 3]
	RO:	5.833 ± 0.408	[6 6 6 6 6 5]	RO:	6.000 ± 0.000	[6 6 6 6 6 6]
	MLP:	0.667 ± 0.516	[0 1 1 0 1 1]	MLP:	0.500 ± 0.548	[0 1 1 1 0 0]
	DE:	1.833 ± 0.753	[1 2 3 1 2 2]	Jaya:	1.667 ± 1.633	[2 0 0 3 4 1]
	GWO:	2.667 ± 1.751	[2 5 2 4 3 0]	GWO:	2.500 ± 1.643	[3 2 5 0 2 3]
SI:	PSO:	3.167 ± 1.722	[3 4 0 3 5 4]	PSO:	3.167 ± 1.472	[1 3 2 4 5 4]
	Jaya:	3.167 ± 1.835	[4 0 4 2 4 5]	DE:	3.167 ± 1.472	[4 4 3 2 1 5]
	GA:	3.500 ± 1.975	[5 3 5 5 0 3]	GA:	4.000 ± 1.265	[5 5 4 5 3 2]
	RO:	6.000 ± 0.000	[6 6 6 6 6 6]	RO:	6.000 ± 0.000	[6 6 6 6 6 6]

Table 9.1: Swarm algorithm rank (mean $\pm$ std) by initialization scheme for 30 particles and 1000 iterations (left) or 3000 iterations (right).

Algorithm	*Top-3 Count*
MLP	6
GWO	4
DE	3
PSO	3
Jaya	2
GA	0
RO	0

again confirming the MLP result already noted and giving a nod to GWO, DE, and PSO. However, a swarm of only 30 particles is relatively small. Repeating the experiment with a swarm of 100 particles might offer more insights.

Table 9.2 shows the rankings for a swarm with 100 particles across the same iteration limits and initializations as Table 9.1. If we again rank by the number of times each algorithm appears in the top three we get,

Algorithm	*Top-3 Count*
MLP	6
DE	4
Jaya	3
GWO	2
GA	2
PSO	1
RO	0

Again confirming the MLP, but also revealing a strong showing by DE. GWO dropped two positions relative to the 30-particle swarms while GA rose two positions. RO is still the worst performer.

	MLP:	0.667 ± 0.816	[0 1 0 0 1 2]	MLP:	0.667 ± 0.816	[0 1 0 1 0 2]
	DE:	1.500 ± 1.049	[2 0 1 3 2 1]	DE:	1.333 ± 1.506	[2 0 4 0 1 1]
	Jaya:	1.667 ± 1.633	[3 4 2 1 0 0]	GWO:	2.833 ± 1.169	[3 2 1 4 4 3]
RI:	PSO:	3.500 ± 0.837	[4 3 3 5 3 3]	Jaya:	2.833 ± 1.941	[5 3 2 5 2 0]
	GWO:	3.500 ± 1.761	[1 2 6 4 4 4]	PSO:	3.000 ± 1.414	[1 4 3 2 3 5]
	GA:	4.333 ± 1.211	[5 5 4 2 5 5]	GA:	4.333 ± 0.816	[4 5 5 3 5 4]
	RO:	5.833 ± 0.408	[6 6 5 6 6 6]	RO:	6.000 ± 0.000	[6 6 6 6 6 6]
	MLP:	0.500 ± 0.548	[0 0 1 1 0 1]	MLP:	1.667 ± 1.633	[0 4 0 2 3 1]
	PSO:	1.667 ± 1.506	[3 3 0 0 1 3]	DE:	2.000 ± 2.000	[1 0 5 1 1 4]
	GA:	2.500 ± 1.049	[1 2 3 3 4 2]	Jaya:	2.167 ± 0.753	[3 1 2 3 2 2]
QI:	DE:	2.667 ± 1.633	[2 4 4 4 2 0]	PSO:	2.333 ± 2.066	[4 2 3 0 5 0]
	Jaya:	3.667 ± 1.633	[4 5 2 2 3 6]	GA:	2.833 ± 1.169	[2 3 1 4 4 3]
	RO:	5.000 ± 0.632	[5 6 5 5 5 4]	GWO:	4.333 ± 2.251	[6 6 4 5 0 5]
	GWO:	5.000 ± 2.000	[6 1 6 6 6 5]	RO:	5.667 ± 0.516	[5 5 6 6 6 6]
	MLP:	0.833 ± 0.408	[0 1 1 1 1 1]	MLP:	1.000 ± 0.894	[2 1 2 0 0 1]
	GA:	1.333 ± 1.506	[3 0 2 3 0 0]	Jaya:	2.167 ± 1.722	[1 0 3 1 4 4]
	DE:	2.333 ± 1.633	[1 4 0 2 3 4]	GWO:	2.500 ± 1.049	[3 3 1 4 2 2]
SI:	PSO:	3.333 ± 1.211	[4 3 4 5 2 2]	GA:	2.833 ± 2.137	[0 5 4 2 1 5]
	Jaya:	3.333 ± 1.862	[5 5 3 0 4 3]	DE:	3.167 ± 1.722	[4 4 0 3 5 3]
	GWO:	3.833 ± 1.472	[2 2 5 4 5 5]	PSO:	3.333 ± 2.066	[5 2 5 5 3 0]
	RO:	6.000 ± 0.000	[6 6 6 6 6 6]	RO:	6.000 ± 0.000	[6 6 6 6 6 6]

Table 9.2: Swarm algorithm rank (mean ± std) by initialization scheme for 100 particles and 1000 iterations (left) or 3000 iterations (right).

If we rank the algorithms for the 100 particle/3000 iteration case by their mean MCC and initialization scheme we get the following,

Algorithm	*RI*	*SI*	*QI*
DE	$0.95547 \pm 0.01666, 0, 0$	$0.93353, 0.02950, 4, 0$	$0.94901 \pm 0.02231, 0, 0$
MLP	$0.95016 \pm 0.01039, 1, 0$	$0.94821, 0.00943, 0, 0$	$0.94253 \pm 0.01320, 2, 0$
Jaya	$0.93364 \pm 0.03090, 2, 0$	$0.93936, 0.01998, 1, 0$	$0.94467 \pm 0.01046, 1, 0$
GWO	$0.93198 \pm 0.01737, 3, 0$	$0.93362, 0.02113, 3, 0$	$0.62062 \pm 0.48133, 5, 2$
PSO	$0.92870 \pm 0.01319, 4, 0$	$0.92105, 0.03565, 5, 0$	$0.93461 \pm 0.03436, 4, 0$
GA	$0.90925 \pm 0.02435, 5, 0$	$0.93445, 0.02638, 2, 0$	$0.93831 \pm 0.00056, 3, 0$
RO	$0.80685 \pm 0.05445, 6, 0$	$0.10012, 0.27209, 6, 3$	$0.00000 \pm 0.00000, 6, 6$

where the table shows the mean MCC (± std), the algorithm's rank for that initialization scheme, and the number of times the search failed, meaning the MCC was zero or very close to zero.

Some general observations are in order. DE and Jaya tend to do well regardless of the initialization scheme. RO is entirely unsuited to this task as it always performs poorly and fails for SI and QI initialization schemes. PSO, GWO, and GA are mediocre performers, though GA's performance improves when using QI and SI. GA's rank for RI is 5, but it moves to 3 and then 2 for QI and SI.

As for the initialization schemes themselves, RI results in the best models followed by QI when the search doesn't fail. Some algorithms are more sensitive to the initialization method than others.

Many possible avenues remain to be explored when applying swarm techniques to neural networks. Please indulge your curiosity and explore them. For example, we initialized the swarms with small values since we know that works well when training models with gradient descent. Is that really necessary for the swarms? Also, we used the score, but other metrics could be used instead.

Finally, we spent no time optimizing the swarm algorithms themselves by adjusting their parameters. For example, we did not change F or CR for DE, let alone the "rand", "best", or "toggle"

options. Nor did we adjust the inertia parameter and schedule for PSO. The same is true for mutation and crossover probabilities for GA and the scale factor for RO. There is every reason to believe a more focused set of experiments seeking to optimize these values would lead to better results. Could a swarm be used to optimize swarm parameters?

In this chapter, we explored an optimization problem in a high-dimensional space, nearly 2000 dimensions. We saw that the swarms, with a few exceptions for specific algorithms and initialization schemes, were able to learn effectively and, at times, out-perform the standard MLP approach.

This chapter's experiments are a form of *neuroevolution* [26], a field of research that uses evolutionary algorithms to evolve neural networks. Neuroevolution is not restricted to searching for a fixed architecture's weights, as we did here, but goes beyond this to evolving the architecture itself. Neuroevolution has recently found a synergy with deep learning and models far larger than were thought possible to learn with an evolutionary algorithm have been trained. See [26] for a more comprehensive picture.

Speaking of pictures, let's continue now with a new application area: images.

Chapter 10

Images

In this chapter, we cast three common image manipulations, registration, segmentation, and enhancement, as swarm optimization problems. Registration involves aligning multiple images, perhaps acquired over time, so they overlap ideally. Segmentation seeks to split an image into meaningful parts. And, while the term "enhancement" means multiple things, it generally alludes to "making an image look nicer," which is how we'll use it here.

As with previous experiments, we set up the problem, discuss how to map it to a swarm optimization task, and then have fun with some experiments to see how well we do. As always, our overall goal is to build intuition as to how diverse tasks can be viewed as swarm optimizations.

10.1 Image Registration

Many medical imaging modalities collect a series of images over time. For example, a functional magnetic resonance imaging (fMRI) brain study images the brain over time to track changes in blood oxygenation corresponding to increased neuronal activity. Deoxyhemoglobin is paramagnetic, while oxyhemoglobin is diamagnetic. This difference is detectable in the signal received by an MRI scanner and useful in localizing the source of neuronal activity. The changes in the images over time are subtle and highly susceptible to patient motion. Therefore, we'd like to align the acquired images so the brain remains as stationary as possible image to image.

A second use case involves aligning images of the same subject acquired with different imaging modalities. For us this means aligning CT (computed tomography – x-rays) with PET (positron-emission tomography – gamma rays) so both the anatomy provided by CT and the physiology provided by PET are appropriately registered.

Our final use case is from microscopy. We'll use swarm techniques to align frames from videos taken with a geology microscope to produce a sharper image of the object.

The key to each of these is the ability to register the images to each other. Registration finds the best way to align the images to maximize some metric. In general, image registration falls into one of two approaches. Suppose the objects are not easily deformed, meaning they retain their shape over time or images. In that case, registration is termed "rigid registration" and consists of finding a rotation angle and translation vector, and possibly a scaling factor. If the objects do deform over time or images, we might need to use nonrigid registration to align and warp the images. We'll restrict ourselves to rigid registration in this chapter though there is no *a priori* reason why swarms cannot be applied to nonrigid registration.

10.1.1 The Problem

We can state the problem quite simply: we have a collection of misaligned images, and we need to align them via rotation, translation, and scaling. We assume the imaged object is rigid and does not change its shape from image to image, only its orientation and position. We'll use grayscale images from magnetic resonance imaging, a PET-CT study, and microscopy as our examples.

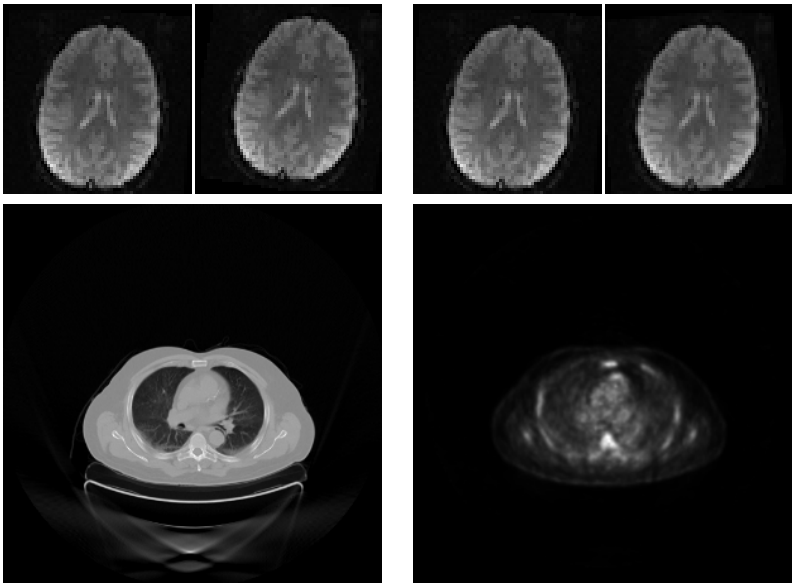


Figure 10.1: Top: two unaligned MRI images (left) and aligned versions after a translation and rotation (right). Bottom: a CT image (left) and paired PET image (right).

For example, Figure 10.1 shows two magnetic resonance images taken at times t_0 and $t_1, t_1 > t_0$ (top). Motion between the images has been exaggerated beyond what is typical. On the right, the images are aligned by applying a translation in x and y of 2.976 and -3.209 pixels followed by a counter-clockwise rotation of 5.195 degrees. Our goal is to align a series of images like these to the orientation of the first one.

The bottom of Figure 10.1 shows us a CT image and a paired PET image. CT displays anatomy, while PET shows physiology, the radioactive tracer's activity as taken up by different tissues. Typically, cancer shows a strong response to the tracer leading to a bright spot in the image.

10.1.2 The Setup

The problem is easy to state and straightforward to consider. We need to find three numbers: x translation, y translation, and rotation angle. We'll add scaling later. So, our search space is small, only three dimensions. The more interesting part of this experiment is the objective function. What does it mean for a pair of images to be aligned? How can we measure alignment?

If the images were identical, only shifted and rotated, we might minimize the error by subtracting them and adding the per-pixel residuals. Perfectly identical images that are perfectly aligned will, of course, have zero residuals. However, our images will not be perfectly identical. For example, the entire point of a functional brain scan is to observe small, but significant, differences in pixel intensities throughout the brain and correlate them with known external activity like listening to a sound or tapping fingers together. Further, breathing and even heartbeats cause a shift in the brain's position, so motion is inevitable.

Therefore, while we might consider minimizing an error metric, like the mean squared error between the images, we might want to think a little more before proceeding. Additionally, what if the images we desire to align are from entirely different modalities like our CT and PET images? Such images are not in any real sense "identical," so subtraction isn't likely to be helpful.

Images are not random noise; they contain structure. In terms of information theory, images have information, so we might expect their entropy is something less than that of pure randomness. When information is measured in bits, a random 8-bit grayscale image should have an overall entropy near eight as there are eight bits in a byte. Bearing this in mind, could we do something with the information content of the images and build a metric indicating how well they are aligned?

The entropy alone won't suffice. It's entirely possible to have two completely different images with the same entropy. We need a way to measure information between the two images, a measure telling us something about how much we know about image B given image A. As it happens, there is a metric like that: mutual information.

Mutual information is a measure of how much information we have about one thing, given information about another. For images, the mutual information is maximized when the images are best aligned. The particular mutual information we'll work with is known as *normalized mutual information* (NMI). To find the NMI between two images, we need the histogram of the individual images and the pair's joint histogram. The histograms are proxies for the individual (marginal) probabilities and the joint probability of the image pixel values. We'll start with the joint probability as we get the individual probabilities by summing along the axes of the 2D joint probability distribution.

To find the joint histogram of two images, x and y, we need to count the number of times each pair of image intensities appears. We are using grayscale images with intensities in the range $[0, 255]$. Therefore, we could make a joint histogram by creating a 256x256 array and then running through each pixel of the image incrementing the (i, j) bin by one every time image x has an intensity of i and image y has an intensity of j for the current pixel. The histogram of image x is found by summing across the rows of the joint histogram. Likewise, the histogram of image y is found by adding across the columns. Dividing by their overall sum turns the histograms into probability distribution estimates.

In practice, using 256 bins for the joint histogram produces a sparse array. There are many combinations of intensities from image x and image y that do not appear together, so the count is zero. Good alignment can be found using a smaller number of bins, like twenty.

So far, our NMI calculation has found the joint histogram between the two images, converted it to a joint probability distribution, and then found the two marginal probabilities to give us the individual image probability distributions. What do we do with all these distributions?

The answer is to calculate some entropies and combine them to get the NMI, the value we'll use as an objective, something we seek to maximize as that implies well-aligned images.

Let's define some quantities, the probability distributions we'll find from the image histograms. Specifically, p_x is the intensity probability distribution for image x. Likewise, p_y is the probability distribution for image y and p_{xy} the joint probability distribution for both images. With these, we can calculate the normalized mutual information,

$$h_x = -\sum_i^N p_x^i \log_2 p_x^i \tag{10.1}$$

$$h_y = -\sum_i^N p_y^i \log_2 p_y^i \tag{10.2}$$

$$h_{xy} = -\sum_i^{2N} p_{xy}^i \log_2 p_{xy}^i \tag{10.3}$$

$$\text{NMI} = (h_x + h_y)/h_{xy} \tag{10.4}$$

The NMI has a maximum value of 2, so the closer we are to this, the better aligned the images. On the right of Figure 10.1, the aligned magnetic resonance images have an NMI of 1.568, for example. The beauty of using NMI comes from the fact that the modality of the images need not be the same; mutual information still captures the alignment because knowledge of one image implies knowledge about the other.

The NMI is straightforward to calculate in code using NumPy. Our implementation is

```
def NMI(a,b):
    h = np.histogram2d(a.ravel(),b.ravel(), bins=20)[0]
    pxy = h / h.sum()
    px = pxy.sum(axis=0)
```

```
    py = pxy.sum(axis=1)
    hx = -(px*np.log2(px+1e-9)).sum()
    hy = -(py*np.log2(py+1e-9)).sum()
    hxy= -(pxy*np.log2(pxy+1e-9)).sum()
    return (hx+hy) / hxy
```

where we first calculate the joint histogram using twenty bins. The `np.histogram2d` function does this for us. Once we have `h`, a 20x20 matrix, we get the joint probability by dividing each element by the overall sum. This gives us `pxy`. The marginal probabilities come next by simple summing of `pxy` along the two axes.

With the distributions in place, we calculate the necessary entropies adding a tiny amount to avoid zeros in the log. The negative sum calculates `hx`, `hy`, and `hxy`, all scalars. Finally, the NMI is returned as the sum of the individual entropies divided by the joint entropy.

To set up the problem as a swarm search, we need a reference image, the image to which we'll align the others, and an image to be aligned. Each particle in the swarm represents a rotation angle, an x direction shift, and a y direction shift. The objective function applies the shifts in x and y, then the rotation, to generate a candidate image. The objective function returns the normalized mutual information between the candidate image and the reference image. Since the NMI is maximized when aligned, the objective function returns the negative as the framework always minimizes.

The bounds are straightforward, we can use the `Bounds` class as it is. Therefore, we need only define the objective function class and use the framework components. The code is in the file `rigid.py`. The `Objective` class is,

```
from scipy.ndimage import rotate, shift
class Objective:
    def __init__(self, dname, sname):
        self.dst = np.array(Image.open(dname).convert("L"))
        self.src = np.array(Image.open(sname).convert("L"))
        self.x, self.y = self.dst.shape
    def Evaluate(self, p):
        angle, xshift, yshift = p
        img = rotate(shift(self.src, (xshift,yshift)), angle)
                .astype("uint8")
        x,y = img.shape
        img = img[(x//2-self.x//2):(x//2+self.x//2),
                  (y//2-self.y//2):(y//2+self.y//2)]
        if (img.shape[0] != self.dst.shape[0]) or
           (img.shape[1] != self.dst.shape[1]):
            return 1e9
        return -NMI(self.dst, img)
```

When initialized, we pass in the names of the reference image (`dname`) and the image to be aligned (`sname`). The images are read, converted to grayscale, and stored as NumPy arrays along with the size of the images, here always assumed to be the same.

To evaluate a candidate rotation and translation, we first extract the `angle`, `xshift`, and `yshift`. Then we use SciPy functions, `shift` and `rotate`, to create the candidate image. These functions adapt the size of the output image to ensure it is completely contained. Therefore, we extract the central region of `img` to compare with the reference. If the shift and rotation result in an image that cannot be the same size as the original, we immediately return a very high fitness value; otherwise, we return the negative of the NMI between the candidate and reference.

The bulk of `rigid.py` is in the `main` function. It's format is quite familiar. At the beginning, we parse the command line and gather the pathnames of the images we want to align,

```
def main():
    sdir = sys.argv[1]
    outdir = sys.argv[2]
    npart = int(sys.argv[3])
```

```
ndim = 3 # (angle, x shift, y shift)
niter = int(sys.argv[4])
alg = sys.argv[5].upper()
itype = sys.argv[6].upper()

simgs = [os.path.abspath(sdir+"/"+i) for i in
            os.listdir(sdir) if (i.find(".png") != -1)]
simgs.sort()

os.system("rm -rf %s; mkdir %s" % (outdir,outdir))
os.system("mkdir %s/frames" % outdir)

x,y = np.array(Image.open(simgs[0]).convert("L")).shape
b = Bounds([-180,-x//2,-y//2], [180,x//2,y//2], enforce="resample")

results = []
im = Image.open(simgs[0]).convert("L")
im.save(outdir+"/frames/"+os.path.basename(simgs[0]))
```

The command line holds the usual suspects. We'll see an example call below. We use `simgs` to store the pathnames of the images to be aligned. After creating an output directory, we load the first image, which we'll always use as the reference image, and get its dimensions.

The dimensions let us create the Bounds object. Angles are expressed in degrees and bounded to $[-180, 180]$. Image shifts are bounded to one-half the dimensionality of the image, either to the left or right for x or top and bottom for y. Finally, edge transgressions during the search are handled by resampling along the offending dimension. Before continuing, we set up a list to hold registration results and copy the reference image to the output directory.

The rest of `main` runs a loop over the images aligning each one to the reference with a unique swarm search. Output is stored to be dumped in the output directory as a record of the search process. In code,

```
s  = "\nRegistration results:\n"
print(s, flush=True, end="")

for k in range(len(simgs)):
    if (k == 0):
        continue
```

to start the loop over images and skip the first as that's the reference image.

Next, a standard swarm search using the framework takes place,

```
        if (itype == "QI"):
            i = QuasirandomInitializer(npart, ndim, bounds=b)
        elif (itype == "SI"):
            i = SphereInitializer(npart, ndim, bounds=b)
        else:
            i = RandomInitializer(npart, ndim, bounds=b)
        obj = Objective(simgs[0], simgs[k])
        if (alg == "PSO"):
            swarm = PSO(obj=obj, npart=npart, ndim=ndim, init=i, bounds=b,
                        max_iter=niter, inertia=LinearInertia())
        elif (alg == "DE"):
            swarm = DE(obj=obj, npart=npart, ndim=ndim, init=i, bounds=b,
                        max_iter=niter)
        elif (alg == "RO"):
            swarm = RO(obj=obj, npart=npart, ndim=ndim, init=i, bounds=b,
                        max_iter=niter)
        elif (alg == "GWO"):
```

```
            swarm = GWO(obj=obj, npart=npart, ndim=ndim, init=i, bounds=b,
                        max_iter=niter)
        elif (alg == "JAYA"):
            swarm = Jaya(obj=obj, npart=npart, ndim=ndim, init=i, bounds=b,
                        max_iter=niter)
        elif (alg == "GA"):
            swarm = GA(obj=obj, npart=npart, ndim=ndim, init=i, bounds=b,
                        max_iter=niter)
        st = time.time()
        swarm.Optimize()
        en = time.time()
        res = swarm.Results()
        results.append(res)
        aligned = ApplyRegistration(simgs[0], simgs[k], res["gpos"][-1])
        Image.fromarray(aligned).save(outdir+"/frames/"+
            os.path.basename(simgs[k]))
        angle, xshift, yshift = res["gpos"][-1]
        t = "  %3d: (NMI=%0.6f)  theta=%10.5f, x=%10.5f, y=%10.5f\n" %
                (k, -res["gbest"][-1], angle, xshift, yshift)
        print(t, flush=True, end="")
        s += t
```

The proper swarm is constructed using a custom `Objective` class instance. The `Optimize` call performs the search and `Results` returns the conclusion reached. The current image is aligned according to the search results (`aligned`) and written to the `frames` directory of the output. Summary information is displayed.

Finally, the summary and list of search results is dumped to the output directory,

```
    s += "\n"
    print()
    pickle.dump(results, open(outdir+"/results.pkl","wb"))
    with open(outdir+"/README.txt","w") as f:
        f.write(s)
```

The `ApplyRegistration` function referred to above is,

```
def ApplyRegistration(dname, sname, p):
    dst = np.array(Image.open(dname).convert("L"))
    X,Y = dst.shape
    src = np.array(Image.open(sname).convert("L"))
    angle, xshift, yshift = p
    img = rotate(shift(src, (xshift,yshift)), angle).astype("uint8")
    x,y = img.shape
    img = img[(x//2-X//2):(x//2+X//2),(y//2-Y//2):(y//2+Y//2)]
    return img
```

and it applies a rotation and translation to an image (`sname`) to align it with a reference image (`dname`).

The output directory contains `frames` to hold the aligned images, `results.pkl` to keep the list of search results, one per aligned image, and, finally, `README.txt`, a copy of the text output to the console during the search.

Let's take `rigid.py` for a test drive.

10.1.3 The Results

We need some misaligned images against which to test our code. A functional brain imaging dataset is a good choice for us. The necessary dataset is on the book website, see `fmri_rtk.npy`. The script

extract_frames.py creates a frames directory after applying additional random translations and rotations. We do this to make the alignment task more of a challenge and to make misalignments more obvious.

The images themselves are 64x64 pixel grayscale, echo-planar magnetic resonance images. The subject was yours truly, so there is no issue about using the data for any purpose. Echo-planar images are rather noisy. The experiment was conducted over twenty years ago using a customized Bruker 3 Tesla scanner that was notoriously tricky to set up correctly, hence the ghosting in the images. Nonetheless, the images are ideal for our purposes.

The functional brain images are a time series to be aligned. A slight variation of rigid.py, rigid_pairs.py, aligns pairs of images. We won't show the code changes; they are small, but we need a paired dataset, so we'll use frames from a publicly available PET-CT acquisition. The images are aligned already, so we'll intentionally shift and rotate the PET frames and leave the CT frames fixed. The images are in the medical data directory. This dataset demonstrates the power of using mutual information as our objective function since the images are from different modalities and are, visually, quite different from each other.

After our experiments with three-parameter rigid registration, we'll add a fourth, scaling, and experiment with a collection of geology microscope images extracted from a video taken at the eyepiece using a handheld cell phone. The images are in the bryozoa, crane_fly, and coin directories. We'll use the rigid_scale.py script for these experiments. Again, the differences between rigid.py and rigid_scale.py are small, so we won't list them here.

Let's start with three-parameter rigid registration.

Three-Parameter Registration

To run a three-parameter registration at the command line we need something like this,

```
> python3 rigid.py fmri_rtk/frames aligned 30 1000 RO RI
```

Here, fmri_rtk/frames points to the location of the MRI images we want to align. The output summary and frames go in the aligned directory. As always, we specify the number of particles, number of iterations, algorithm, and initialization strategy.

Our experiments use similar command lines to test each algorithm under three conditions: 30 particles/100 iterations, 100 particles/100 iterations, and 30 particles/1000 iterations. The best way to see the results is to use an image viewer to scroll through the output frames, those in the frames directory. There are numeric measures we can look at as well.

The output generated by rigid.py looks like this,

```
Registration results:
    1: (NMI=1.566116) theta=  5.19247, x= -3.33804, y= -8.01754
    2: (NMI=1.607348) theta=  7.21931, x=  0.51426, y= -2.00828
    3: (NMI=1.582311) theta= 10.99823, x= -2.35665, y= -3.28493
    4: (NMI=1.611861) theta=  7.96963, x=  1.44706, y= -1.94081
    5: (NMI=1.680435) theta= -0.00403, x=  4.04877, y= -1.97663
    6: (NMI=1.660613) theta=  0.05624, x=  1.07595, y= -8.00458
    7: (NMI=1.587619) theta=  8.96489, x= -0.14424, y= -3.35200
    8: (NMI=1.568797) theta=  9.85593, x= -3.57678, y= -6.91617
```

The first eight frames are shown. The best NMI for each frame, aligning it to the first frame in the directory, is given. Higher is better. The final set of parameters found is shown next: theta, the rotation angle, and x and y shifts. The SciPy shift and rotate routines work with non-integer values, so the subpixel shifts are meaningful, at least to a digit or two.

The functional MRI dataset was randomly shifted and rotated. Rotations were restricted to ± 10 degrees, so any theta value in the output beyond that range indicates a search failure. When scrolling through output frames, you'll immediately detect such failures visually by a sudden jump in the image sequence.

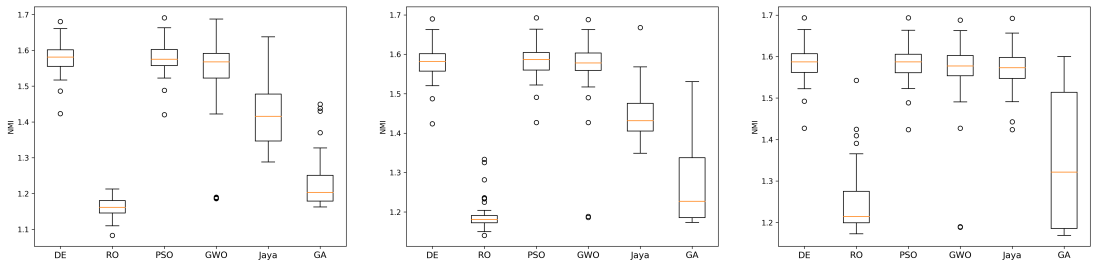


Figure 10.2: NMI over all frames by algorithm type and swarm parameters for the fMRI frames. Left: 30 particles, 100 iterations. Middle: 100 particles, 100 iterations. Right: 30 particles, 1000 iterations.

Let's run each algorithm against the magnetic resonance images. A simple script sets up repeated calls to `rigid.py` and the output directories capture the results. We'll not look at the images directly here, they are too small to show meaningful changes in print, but we'll consider the distribution of NMI values by algorithm type and swarm/iteration limits.

Figure 10.2 shows box plots for each algorithm and swarm configuration. On the left are the results for a swarm of 30 particles and 100 iterations. The clear winners are DE and PSO, with the highest average NMI and a relatively narrow IQR. This is true for a small swarm and few iterations on up. On the other hand, RO is a consistently poor performer at this task. Likewise, GA performs poorly with a wide range of results. However, to be fair to GA, there are only three parameters here, not a lot for evolution via mutation and crossover to work with.

GWO and Jaya give mixed results. Let's start with GWO. A first glance at Figure 10.2 might lead one to believe GWO is a solid performer, and, in some ways, it is. However, notice the clump of outliers at low NMI. There are times when GWO fails spectacularly; it isn't consistent like DE and PSO are.

Jaya's results are interesting. For small swarms and fewer iterations, Jaya doesn't deliver strong results. When flipping through the registered images, a lot of motion is evident. As the number of swarm iterations increases, Jaya's results improve, but it isn't until 1000 iterations that Jaya starts to look like DE and PSO. For this task, Jaya converges slowly. This is a definite negative as the entire point of image registration is to align the images as precisely as possible, which might take Jaya an asymptotically long time to accomplish.

Let's take a look at aligning the paired medical images. For that we need command lines like this,

```
> python3 rigid_pairs.py ct pt med 20 100 RO RI
```

with `ct` pointing to the CT frames, `pt` pointing to the paired but shifted PET frames, and `med` the output directory. The remainder of the command line matches `rigid.py`.

The output directory contains the source frames (`frames0`), the paired frames after alignment (`frames1`), and an additional directory called `merged`. Since we're working with grayscale images, we can overlay the CT and PET by setting the red channel of an RGB image to the CT and the green channel to the PET. This is what ends up in the `merged` directory. The merged images will be more yellow when the CT and PET overlap well and show red or green alone if the alignment is poor. I encourage you to flip through the merged images for various runs of the alignment task.

Let's run three sets of experiments on the paired images. First, we'll use 20 particles and 100 iterations. Next, 100 particles and 100 iterations. Finally, 20 particles and 1000 iterations. Our approach will be quite similar to what we did above for the time series of images.

Figure 10.3 shows how we did for the three sets of experiments in the same way we showed the earlier results in Figure 10.2. A similar story presents itself. DE, PSO, and, this time, GWO, do well overall for the 20 particle, 100 iteration case, though GWO fails once (the outlier). RO and GA again perform poorly. However, when the search is allowed to continue, the 20 particle, 1000

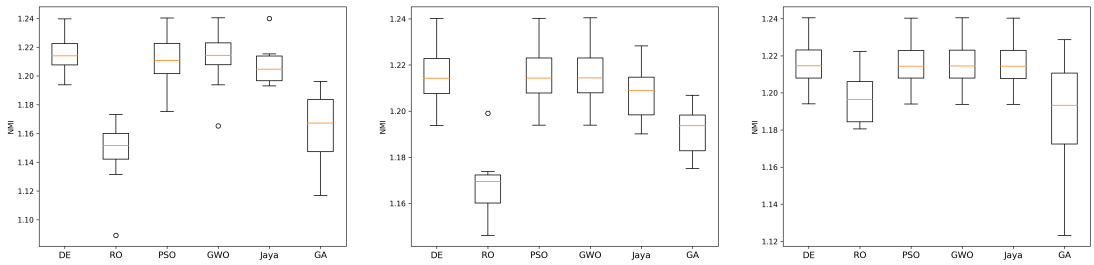


Figure 10.3: NMI over all frames by algorithm type and swarm parameters for the PET-CT frames. Left: 20 particles, 100 iterations. Middle: 100 particles, 100 iterations. Right: 20 particles, 1000 iterations.

iteration case, even RO does better, but our money is better put on DE, PSO, Jaya, and GWO.

Swarms can locate useful rigid registration parameters, but, as we've seen repeatedly, not all swarm algorithms are equally efficacious. The old standards, DE and PSO, are again good options. Jaya works well, too, but takes longer to converge. The sporadic failings of GWO make one hesitate to use it, especially when the system is automated and high reliability is required, though for a one-off task that can be run again manually it might be just fine.

Let's continue with rigid registration, but this time expand our parameter space by adding a scaling factor to help account for out of plane camera motion.

Four-Parameter Registration

The experiments of Section 10.1.3 searched for three parameters: rotation, x translation, and y translation. The images we're working with in this section were acquired by holding a cell phone while recording video in front of the eyepiece of a geology microscope with a magnification of 20x. As might be expected, a lot of motion occurred, even when trying to be as still as possible. A three-parameter registration can compensate for translation and rotation in the plane of the image, but not for motion towards or away from the eyepiece. To account for that possibility, we'll introduce an additional search parameter: scaling of the image. We'll scale the image by multiplying the size by a constant factor and interpolating.

The code we need is in the file `rigid_scale.py`. It is nearly identical to the original `rigid.py` script, but the objective function is adjusted,

```
class Objective:
    def __init__(self, dname, sname):
        self.dst = np.array(Image.open(dname).convert("L"))
        self.src = np.array(Image.open(sname).convert("L"))
        self.x, self.y = self.dst.shape
    def Evaluate(self, p):
        angle, xshift, yshift, scale = p
        im = Image.fromarray(self.src)
        a,b = im.size
        img = np.array(im.resize((int(a*scale),int(b*scale)),
                      resample=Image.BICUBIC))
        img = rotate(shift(self.src, (xshift,yshift)),
                angle).astype("uint8")
        x,y = img.shape
        img = img[(x//2-self.x//2):(x//2+self.x//2),
                  (y//2-self.y//2):(y//2+self.y//2)]
        if (img.shape[0] != self.dst.shape[0]) or
           (img.shape[1] != self.dst.shape[1]):
            return 1e9
        return -NMI(self.dst, img)
```

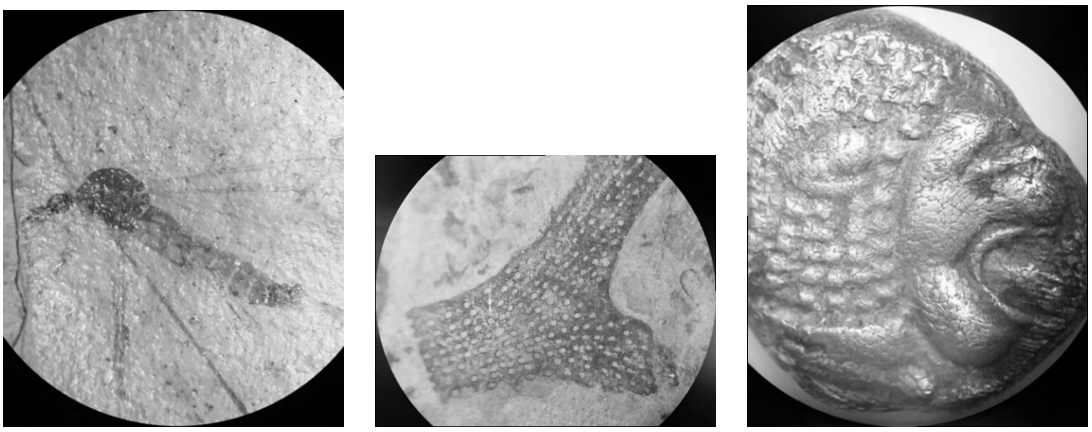


Figure 10.4: Example microscope images: crane fly (left), bryozoa (middle), and coin (right).

There is now a fourth parameter in the particle position, `scale`. To use `scale`, we extract the image dimensions, and use Pillow's `resize` method with bicubic interpolation to change the image size after multiplying by the scale factor. As before, we extract the central region to produce a candidate image the same size as the reference frame. The `ApplyRegistration` function is similarly updated.

We need to set limits on the scale factor. To do this, we add it to the Bounds object,

```
b = Bounds([-180,-x//2,-y//2,0.7], [180,x//2,y//2,1.3], enforce="resample")
```

where we limit the scale factor to between 70% and 130% of the original image size.

As with `rigid.py`, we register the sequence of frames to the first frame. The goal here, however, isn't to generate a time series, but to align the frames so we can add them together to produce a sharper, less noisy, image. To add the frames is straightforward,

```
simgs = [os.path.abspath(sys.argv[1]+"/"+i) for i
            in os.listdir(sys.argv[1])]
oname = sys.argv[2]
img = np.array(Image.open(simgs[0]))
x,y = img.shape
oimg = np.zeros((x,y))

for s in simgs:
    img = np.array(Image.open(s))
    oimg += img

oimg = oimg / oimg.max()
Image.fromarray((255.0*oimg).astype("uint8")).save(oname)
```

where a directory's worth of frames are loaded, added, and scaled $[0, 1]$ to generate a single frame for output (see `add.py`).

Figure 10.4 shows example images from each of the three sets of frames: `crane_fly`, `bryozoa`, and `coin`. The crane fly is a late Eocene to early Oligocene epoch fossil from Florissant, Colorado. Bryozoa are small, colonial animals similar to corals. This specimen is from the middle Devonian Milwaukee Formation in southeastern Wisconsin. Finally, the coin is a silver diobol from the ancient Greek city of Miletos, circa 500 BCE. It features a lion's head facing right. Our hope with these frames is to generate an aligned and stacked version sharper than the originals.

The command line for `rigid_scale.py` is identical to that of `rigid.py`. Based on the results above, we'll restrict our runs to just DE and PSO. When a run is complete, we'll use `add.py` to produce an aligned frame and then characterize the frame's sharpness to see whether the swarm alignment helped or not.

A typical registration run produces output like so,

```
> python3 rigid_scale.py data/bryozoa/ bryozoa_30_200_PSO 30 200 PSO RI

Registration results:
 1: (NMI=1.601956) theta= 0.04450, x=-0.04652, y=-0.01747, scale= 0.94108
 2: (NMI=1.525078) theta= 0.06125, x=-0.11371, y= 0.53900, scale= 0.90403
 3: (NMI=1.467275) theta= 0.03272, x=-0.53992, y= 0.10252, scale= 1.06752
 4: (NMI=1.503909) theta= 0.03595, x=-0.39840, y= 0.35789, scale= 0.85060
 5: (NMI=1.493586) theta= 0.03350, x=-0.55124, y=-0.06489, scale= 1.07348
 6: (NMI=1.475772) theta= 0.04018, x=-0.88871, y= 0.05542, scale= 1.12517
 7: (NMI=1.441288) theta= 0.06283, x=-0.80059, y= 0.39092, scale= 1.02170
 8: (NMI=1.399838) theta= 0.02687, x=-0.52736, y=-0.25481, scale= 0.98239
 9: (NMI=1.352890) theta= 0.04636, x=-0.36493, y=-0.79724, scale= 0.82886
10: (NMI=1.309046) theta= 0.02626, x= 0.14694, y=-0.02422, scale= 1.04672
11: (NMI=1.276236) theta= 0.07970, x=-0.31550, y=-0.40125, scale= 0.99238
```

which we use to produce the final, aligned image,

```
> python3 add.py bryozoa_30_200_PSO bryozoa_30_200_PSO.png
```

Likewise, using

```
> python3 add.py data/gray/bryozoa bryozoa_unaligned.png
```

sums the original, unaligned frames. Repeating the above for the crane fly and coin frames gives us three sets of final images, DE, PSO, and unaligned, from each of the three sources.

Look at the images carefully, you'll notice differences, particularly with the crane fly frames. However, the differences are subtle enough they won't show up in this book. We'd like some sort of numeric measure of the image's sharpness.

Perhaps the most straightforward approach to measuring image sharpness is to compute the mean per pixel sharpness where sharpness is defined as the magnitude of the gradient. To get the gradient per pixel, we use NumPy's `np.gradient` function, which returns gradient images in the x and y directions. The mean of the norm of these components over the image is the value we seek.

For example, Figure 10.5 shows the x (left) and y (right) gradient values for the DE-aligned coin image. The mean per pixel magnitude of these images is our sharpness value,

```
y,x = np.gradient(coin)
sharpness = np.sqrt(x*x+y*y).mean()
```

where `coin` is the NumPy array version of the image. See the file `sharpness.py`. Here's the output per image,

```
Crane fly:
    DE        :  94.6379
    PSO       :  94.6234
    unaligned:  94.3780
Bryozoa:
    DE        :  97.8336
    PSO       :  97.8299
    unaligned:  97.4194
Coin:
    DE        :  98.5367
    PSO       :  98.5257
```

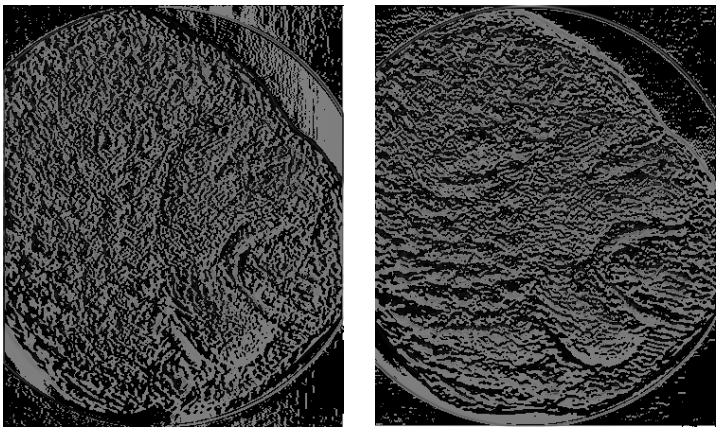


Figure 10.5: Gradients for the DE-aligned coin image: x direction (left), and y direction (right).

```
unaligned:  98.0557
```

showing that in each case, alignment with DE or PSO leads to an improvement over merely stacking raw frames together; the mean sharpness goes up in each case compared to the unaligned images. Also note, the DE-aligned image is consistently slightly sharper than the PSO-aligned image, though the difference is very small and likely not meaningful, certainly not easily visible.

Swarm techniques can help us align a time series, pairs of images from different modalities, or frames for stacking to produce an improved overall image. This is by no means all the utility swarms offer in the area of image processing. Let's change focus now, pun intended, and consider how swarm techniques might help image segmentation.

10.2 Image Segmentation

The segmentation of images into regions is a standard image processing operation. There exist many approaches to image segmentation, some are extremely sophisticated including the application of semantic segmentation with deep neural networks. We'll be more old-school here. We'll segment grayscale images by fitting the image histogram with the sum of N Gaussians and then place each pixel in the image into one of the N bins using the center of the Gaussian as a label. This approach splits the image along intensity lines without concern for spatial relationships in the image. As such, it is a minimalist approach, but will place pixels with similar intensities into the same output bin.

10.2.1 The Problem

Our problem is to learn how to separate an image into regions that have some desired relationship. Here, the association is imposed by the image histogram and how best to approximate it with the sum of a set of Gaussians. The number of Gaussians in the approximation determines the number of labels used for the image. The label is an integer assigning each pixel to a bin. We'll use the intensity value associated with the peaks of the Gaussians.

Figure 10.6 shows a standard test image segmented into 2, 4, or 6 groups by fitting that many Gaussians to the image histogram. The bottom row shows the Gaussian fit superimposed over the original grayscale histogram. The swarm's goal is to find these best-fit Gaussians for the chosen number of groups.

To be specific, for N desired groups, the swarm's goal is to find the parameters to fit N Gaussians,

$$f(x) = c_0 e^{-(x-c_1)^2/c_2^2}$$

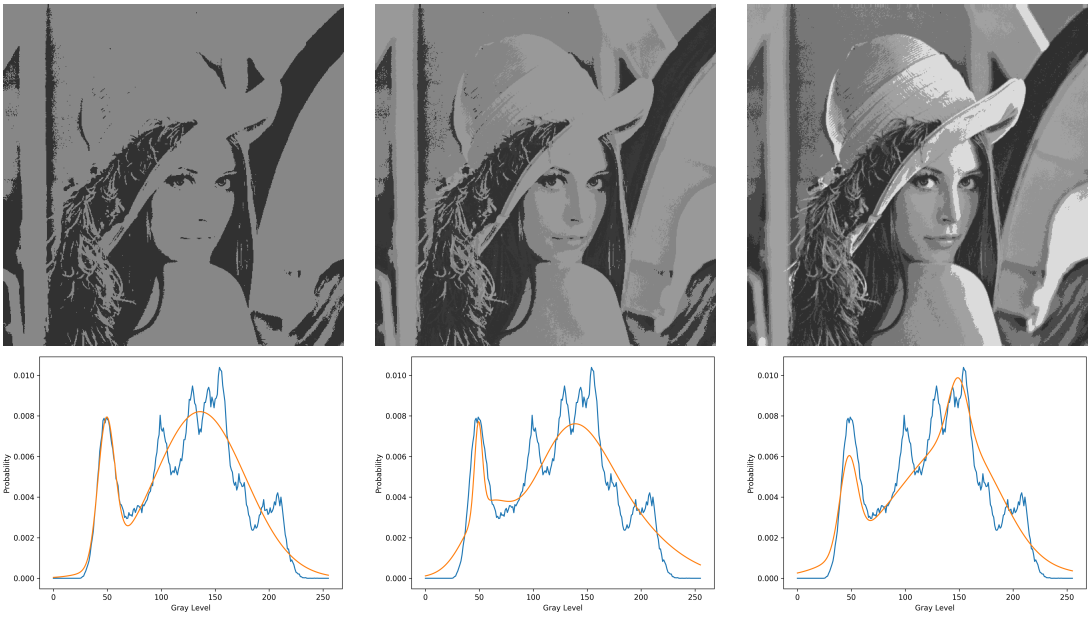


Figure 10.6: Top: The standard Lena test image segmented to 2, 4, and 6 groups. Bottom: The original image histogram and the best fit set of Gaussians (2, 4, or 6).

each with three parameters, c_0, c_1, and c_2. Therefore, if we desire six groups, we need to find $3 \times 6 = 18$ parameters. The height of a Gaussian is controlled by c_0, the width by c_2, and the center position by c_1 with x a grayscale intensity value, $x \in [0, 255]$. The c_1 value will be the assigned label.

10.2.2 The Setup

The problem statement makes it straightforward to map parameters to particle positions: if N groups are desired, we need to find $3N$ parameters, so a particle position is a $3N$-dimensional vector. Further, the parameter values are continuous, so we do not need a custom `Validate` method on a custom `Bounds` object.

What about allowed parameter ranges? Suppose we work with normalized histograms meaning we divide the input image histogram by the sum to create a probability distribution. In that case, the maximum value any bin in the histogram can have is 1.0. Therefore, the c_0 parameter is limited to $[0, 1]$. As c_1 positions the center of the Gaussian, we limit it to $[0, 255]$, the limit being the number of intensities in an 8-bit grayscale image. Finally, we arbitrarily pick $[0, 200]$ as the range for c_2 as 200 is more than half the width of the histogram. Intuitively, as the number of desired groups goes up, we expect the best fit set of Gaussians to be increasingly narrow.

What's our objective function? In a sense, we're curve fitting, but instead of a single function, we have a set of N functions, the number of desired groups, and it's their sum that needs to be as close a fit to the original image histogram as possible. We'll use the sum of the squared error as our metric, which is similar to the curve fitting experiment in Section 8.3. For curve fitting, we used the mean squared error instead. The difference between the two objective functions is minor, and, as we'll see, the sum works nicely in this case.

The code we'll work with is in `segment.py`. The layout is very much old hat. Let's start with the objective function class, the only custom portion of the script,

```
class Objective:
    def __init__(self, img):
        x,y = img.shape
        self.h = np.bincount(img.reshape(x*y), minlength=256)
```

```
        self.h = self.h / self.h.sum()
        self.h[0] = self.h[255] = 0
        self.fcount = 0
    def Evaluate(self, p):
        self.fcount += 1
        n = len(p) // 3
        c = p.reshape((n,3))
        y = np.zeros(256)
        for x in range(256):
            t = 0.0
            for i in range(n):
                t += c[i,0] * np.exp(-(x-c[i,1])**2/c[i,2]**2)
            y[x] = t
        y = y / y.sum()
        return np.sqrt(((self.h-y)**2).sum())
```

The constructor accepts the image to be segmented (`img`), grabs the size, and uses `np.bincount` to get the 256-bin grayscale histogram, which is immediately normalized and stored (`h`). Additionally, we set the count for intensities zero and 255 to zero. Doing this compensates for images with large background regions that are zero (or 255, if inverted). We don't want to waste time trying to fit empty regions that contain no helpful information about the image itself. Finally, we initialize the counter, `fcount`, so we can track how often the objective function is called.

The `Evaluate` method implements a fit and returns the squared error (really square root, but the effect is the same). The particle position (`p`) is reshaped into n sets of three parameters (`c`) corresponding to the three parameters of each of the n Gaussians. Next, we fill `y`, the vector representing the fit to the image histogram, by generating the output of each Gaussian and summing it with the others. After normalizing `y`, we return the error between the fit and the actual image histogram in `h`. The closer this error is to zero, the better the sum of Gaussians is doing at approximating the histogram.

The swarm seeks to find a best set of Gaussians that when summed approximate the image histogram. However, this does not actually segment the image, it only sets up the boundaries to use for segmenting. The function `SegmentImage` takes a best particle position and uses it to produce a segmented version of the image where each pixel is assigned the intensity of the closest Gaussian center. In code,

```
def SegmentedImage(src, p):
    x,y = src.shape
    n = len(p) // 3
    c = p.reshape((n,3))
    labels = np.zeros(x*y, dtype="uint8")
    seg = np.zeros(x*y, dtype="uint8")
    k = 0
    for i in range(x):
        for j in range(y):
            d = np.abs(src[i,j] - c[:,1])
            l = np.argmin(d)
            seg[k] = int(c[l,1])
            labels[k] = l
            k += 1
    return seg.reshape(x,y), labels
```

The particle position, `p`, is split into triplets, the parameters for the `n` Gaussians. We then create an empty segmented image, `seg`, the same size as the input image, `src`, but a flat vector. Next, we loop over each pixel in the input image and locate the nearest Gaussian using the c_1 parameter as the peak location. We set the current output pixel of the segmented image to the selected c_1 value. Recall, c_1 is in the range $[0, 255]$, so we use it as the intensity of the output pixel. We also keep the specific Gaussian index in `labels`. We'll see why shortly.

When all pixels have been placed into a bin, meaning `seg` is updated with the Gaussian peak location as the intensity value and `label` holds the index of the Gaussian selected, we reshape `seg` to match the input image and return it along with the list of associated indices (`labels`).

The bulk of `segment.py`, as far as the `main` function is concerned, is nearly identical to many of our other experiments. We parse the command line which looks like this,

```
> python3 segment.py lena.png 6 30 100 DE RI lena_30_100_DE
```

to segment the file `lena.png` into six groups using DE with a swarm of 30 particles for 100 iterations and random initialization. The output directory is `lena_30_100_DE`.

As before, we select the desired initializer and create the desired swarm object. To set up bounds and the objective function we use,

```
b = Bounds([0,0,0]*nclusters, [1.0,255,200]*nclusters, enforce="resample")
obj = Objective(img)
```

to configure `nclusters` worth of Gaussian parameters in the ranges discussed above and to create the `Objective` instance passing in the NumPy array version of the input image.

Running the segmentation requires a call to `Optimize` followed by results. The summarization code is

```
res = swarm.Results()
pickle.dump(res, open(outdir+"/results.pkl","wb"))

seg,labels = SegmentedImage(img, res["gpos"][-1])
Image.fromarray(seg).save(outdir+"/segmented.png")
Image.fromarray(img).save(outdir+"/original.png")
```

to get and store the results in the output directory, followed by the segmented and original images.

Next, we plot the original image histogram and the fit found. This is where the plots in Figure 10.6 came from,

```
y = PlotFit(res["gpos"][-1])
plt.plot(obj.h)
plt.plot(y)
plt.xlabel("Gray Level")
plt.ylabel("Probability")
plt.savefig(outdir+"/histogram_plot.png", dpi=300)
```

with `PlotFit` defined as

```
def PlotFit(p):
    n = len(p) // 3
    c = p.reshape((n,3))
    y = np.zeros(256)
    for x in range(256):
        t = 0.0
        for i in range(n):
            t += c[i,0] * np.exp(-(x-c[i,1])**2/c[i,2]**2)
        y[x] = t
    y = y / y.sum()
    return y
```

making it virtually identical to the `Evaluate` method of the `Objective` class.

We'd like a numeric way to compare the output of different swarm searches. Of course, we can examine the segmented images to see how they look subjectively, but it would be nice to have some measure we could compare. Fortunately, what we are doing is clustering in one dimension as

we are partitioning the histogram with the nclusters-worth of Gaussians. Indeed, the standard algorithm for this approach is k-means clustering, as we'll see below.

The metric we need is called the *silhouette score*; we can interpret it as a clustering quality metric where good clusterings imply silhouette scores near one. The sklearn toolkit provides the function we need in the sklearn.metrics module. To use it, we need the original image grayscale values and the labels output by the call to SegmentedImage above. The pair tells us which Gaussian, which cluster, each grayscale value was assigned to. This lets the silhouette score do its magic. The function's actual operation is given in the sklearn documentation; we need only know that higher silhouette scores are better and that overlapping clusters return a score near zero while wrong clustering leads to a negative score. The code we need is

```
t = img.reshape((img.shape[0]*img.shape[1],1))
idx = np.argsort(np.random.random(t.shape[0]))
idx = idx[:int(0.25*len(idx))]
score = silhouette_score(t[idx], labels[idx], metric="euclidean")
```

where t is the flattened image. We select a random fraction of the pixels, 25%, to make the silhouette score calculation faster, and pass them, and their labels, to silhouette_score.

Finally, we end the script by displaying the results and storing them in the output directory,

```
s  = "\nSegmentation results:\n\n"
s += "Optimization minimum squared error %0.6f (time = %0.3f)\n" %
         (res["gbest"][-1], en-st)
s += "(%d best updates, %d function evaluations)\n\n" %
         (len(res["gbest"]), obj.fcount)
s += "Silhouette score = %0.6f\n\n" % score
s += "Cluster centers:\n"
s += np.array2string(res["gpos"][-1].reshape((nclusters,3))) + "\n\n"
print(s)
with open(outdir+"/README.txt","w") as f:
    f.write(s)
```

A walk on your own through segment.py from top to bottom will ensure that you follow the sequence of steps. Now, let's try it on some images.

10.2.3 The Results

We'll work with some standard test images, those of Figure 10.7. All of these are 256x256 pixel grayscale images. We'll run each one through segment.py for all combinations of algorithms and 4, 6, and 8 clusters. Finally, we mentioned above that the gold standard for this type of segmentation is to use k-means, so we'll compare it with k-means. To do that, we'll use the code in kmeans.py. To save space, we won't walk through the k-means script, but it makes use of the KMeans class in the sklearn.cluster module and produces output similar to segment.py including the segmented image and the silhouette score.

To apply the algorithms to each test image we use segment_test_images.py,

```
import os
f = [i for i in os.listdir("test_images")]
for alg in ["RO","DE","PSO","GWO","JAYA","GA"]:
    for n in [4,6,8]:
        for t in f:
            cmd = "python3 segment.py test_images/%s %d 20 2000 %s RI
                    segmentations/%s_20_2000_%s_%d" % \
                    (t,n,alg,t[:-4],alg,n)
            print(cmd, flush=True)
            os.system(cmd)
```

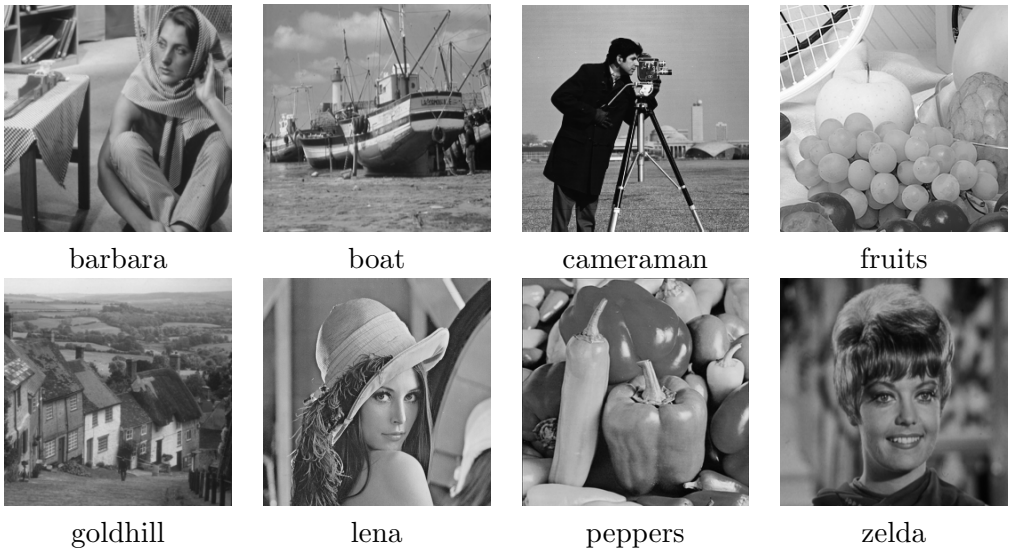


Figure 10.7: The grayscale test images.

which is a straightforward application of `segment.py` to each of the test images with results in the `segmentation` directory. The auxiliary scripts, `merge_test_images.py` and `score_test_images.py`, create merged result images by algorithm and output the silhouette scores, respectively.

Let's use swarms of 20 particles for 2000 iterations each and compare the silhouette scores with the k-means results. Do use the merge script to generate per test image output and examine it in conjunction with the k-means output. It's clear that k-means is doing better overall, which isn't surprising given the known utility of the technique.

Consider Table 10.1, it shows the silhouette score for each segmentation of the test images. Each cell of the table is sorted by rank. The most obvious conclusion is that k-means is the gold standard as it delivers segmentations leading to the highest silhouette scores with only two exceptions: barbara 4 and cameraman 4. In those cases, Jaya outperformed k-means, and in cameraman 4, RO outperformed both (or did it? See below.) Additionally, in many cases, the gap between the k-means score and the best swarm score is rather high.

Do any swarm approaches show more promise than the others? If we rank the swarms across all images and number of clusters by how often they appear in the top two positions, excluding k-means, we get

Jaya	14
GA	10
RO	8
GWO	7
DE	6
PSO	2

suggesting that Jaya is working well at this task. The genetic algorithm is also a contender in this case.

Both k-means and the swarm search partition the histogram into sections. The k-means algorithm learns specific breakpoints while the swarm learns breakpoints as the c_1 parameter of the Gaussians. For example, the cameraman 4 search is one where both Jaya and RO outperformed k-means. The center locations found for each algorithm were

k-means	18	95	136	171
Jaya	13	71	96	161
RO	13	32	128	167

	4		6		8	
goldhill	k-means:	0.5625	k-means:	0.5434	k-means:	0.5328
	DE:	0.4923	GA:	0.5007	GA:	0.4941
	GWO:	0.4863	JAYA:	0.4778	JAYA:	0.4939
	JAYA:	0.4526	DE:	0.4369	RO:	0.4588
	RO:	0.4317	GWO:	0.4032	GWO:	0.4397
	PSO:	0.4277	PSO:	0.3683	DE:	0.3885
	GA:	0.3670	RO:	0.3628	PSO:	0.3300
lena	k-means:	0.5814	k-means:	0.5793	k-means:	0.5725
	RO:	0.5661	GWO:	0.5167	JAYA:	0.5243
	JAYA:	0.4997	RO:	0.5141	GA:	0.4716
	PSO:	0.4722	GA:	0.5413	RO:	0.4131
	GWO:	0.4453	JAYA:	0.5109	DE:	0.3931
	GA:	0.4270	PSO:	0.3750	PSO:	0.2851
	DE:	0.3828	DE:	0.3434	GWO:	0.2837
barbara	JAYA:	0.5897	k-means:	0.5650	k-means:	0.5461
	k-means:	0.5834	RO:	0.5393	JAYA:	0.5269
	RO:	0.5296	GWO:	0.5270	RO:	0.5120
	DE:	0.5052	JAYA:	0.5076	GA:	0.5055
	GWO:	0.5024	GA:	0.4660	DE:	0.4655
	PSO:	0.4666	DE:	0.3864	GWO:	0.4592
	GA:	0.4402	PSO:	0.3391	PSO:	0.3783
peppers	k-means:	0.5860	k-means:	0.5620	k-means:	0.5538
	GA:	0.5513	RO:	0.5288	GWO:	0.4676
	JAYA:	0.5383	JAYA:	0.4770	DE:	0.4635
	RO:	0.5344	PSO:	0.4703	JAYA:	0.4530
	GWO:	0.5354	DE:	0.4538	GA:	0.4457
	PSO:	0.5201	GA:	0.4020	PSO:	0.4447
	DE:	0.4365	GWO:	0.3882	RO:	0.3201
boat	k-means:	0.5828	k-means:	0.5422	k-means:	0.5279
	DE:	0.4914	GWO:	0.4378	GA:	0.4624
	RO:	0.4440	GA:	0.4065	GWO:	0.3895
	GWO:	0.4009	DE:	0.3407	PSO:	0.3317
	GA:	0.1712	RO:	0.2664	JAYA:	0.2777
	JAYA:	0.1056	JAYA:	0.2512	RO:	0.2142
	PSO:	-0.0882	PSO:	0.0470	DE:	0.1311
zelda	k-means:	0.5520	k-means:	0.5443	k-means:	0.5266
	JAYA:	0.5512	JAYA:	0.5284	JAYA:	0.5250
	GA:	0.5454	PSO:	0.5212	GA:	0.4826
	DE:	0.5360	GWO:	0.4938	GWO:	0.4800
	GWO:	0.5299	GA:	0.4810	RO:	0.4790
	RO:	0.5282	DE:	0.4797	DE:	0.4060
	PSO:	0.5073	RO:	0.4563	PSO:	0.3989
cameraman	RO:	0.6438	k-means:	0.6014	k-means:	0.6163
	JAYA:	0.6264	DE:	0.5193	GWO:	0.5204
	k-means:	0.6231	RO:	0.5141	DE:	0.4965
	PSO:	0.6098	GA:	0.5121	JAYA:	0.4366
	GWO:	0.5931	GWO:	0.4746	PSO:	0.3997
	DE:	0.5298	JAYA:	0.4673	RO:	0.3797
	GA:	0.4579	PSO:	0.4060	GA:	0.3653
fruits	k-means:	0.5509	k-means:	0.5471	k-means:	0.5398
	GA:	0.5455	JAYA:	0.5062	JAYA:	0.5155
	DE:	0.5319	PSO:	0.4359	GA:	0.5115
	JAYA:	0.5212	GA:	0.3997	RO:	0.4431
	GWO:	0.4115	RO:	0.3857	PSO:	0.3941
	RO:	0.3404	DE:	0.3564	DE:	0.3497
	PSO:	0.2847	GWO:	0.2993	GWO:	0.3447

Table 10.1: Silhouette scores by algorithm type, number of clusters, and ranking.

where the cluster centers (grayscale values) have been rounded to the nearest integer. The image is segmented by replacing every pixel value with the center value for the nearest cluster. So, if the pixel value is 16 and the k-means breakpoints are being used, the 16 is replaced by 18. Similarly, a pixel with intensity 141 would be replaced by 136.

The corresponding segmented images are

k-means

Jaya

RO

and it would be difficult to argue that the swarms have found a better segmentation even though the silhouette scores are higher. The k-means segmentation preserves more image detail. For example, the buildings in the background are easier to interpret in the k-means image compared to the Jaya or RO segmentations.

Our goal was to see how to we might apply swarm optimization to image segmentation. We were able to do so successfully, if not at a state-of-the-art level. Let's move on from image segmentation to image enhancement.

10.3 Image Enhancement

The final experiment of this chapter involves what we'll call "image enhancement". Our goal is to take a grayscale image and make it "look better" where "look better" means improving some metric associated with human perception.

Our approach is quite similar to one that has appeared, repeatedly, in the literature. This approach is a good example of the all-too-common tendency to take a paper, make a slight alteration to the technique, and publish it as a new paper. A cursory perusal of the literature produced, as an example, the following papers, all of which implement the same optimization using the same objective function to enhance images: [27], [28], [29], [30], [31], [32], [33], and [34]. The only functional difference between the papers is the particular optimization technique used: PSO, Firefly, Cuckoo search, DE, PSO, Cuckoo, Cuckoo, and DE, respectively, and often with only minor tweaks to the algorithm.

Of course, our implementation is yet another in this illustrious line of research, but we use the excuse of pedagogy and make no claims to novelty nor applicability. For us, this is merely an exercise.

Disclaimers aside, what, exactly, is it we aim to do? We want to learn how to apply a local image enhancement function to an input grayscale image to make the image look nicer. The previous sentence implies a couple of things. First, we have a local image enhancement function, and we do. Second, that we have some way of measuring what "looks nicer" might mean. Again, we do.

10.3.1 The Problem

The function we seek to optimize is,

$$g_{ij} \leftarrow \frac{kG}{\sigma + b}(g_{ij} - c\mu) + \mu^a \tag{10.5}$$

where g_{ij} is a pixel of the original image, G is the original image mean, μ is the mean pixel intensity of a 3x3 window around the current pixel, (i, j), and σ is the standard deviation of the 3x3 window. The parameters we need to learn are a, b, c, and k.

Our objective function is

$$F = \log(\log(I)) \left(\frac{\text{edgels}}{rc} \right) h \tag{10.6}$$

where r and c are the image dimensions, rows and columns, I is the sum of the pixel intensities of an edge-detected version of the input image, edgels is the number of edges above a threshold, here 20, and h is the entropy of the image,

$$h = -\sum_i p_i \log_2 p_i$$

where p_i is the probability of pixel intensity i derived from the grayscale histogram with 64 bins.

We'll be more explicit about these terms when we look at the code. For now, we claim that F measures "image niceness" and the larger F is, the better the image will look to a human observer.

10.3.2 The Setup

Our setup is straightforward. We just introduced our transfer function, the function whose parameters we need to learn, and the objective function we'll use to decide how well we're doing. We need an `Objective` class that implements the transfer function and F, and we need to decide on limits for the parameters: a, b, c, and k. The parameters are continuous, so our standard `Bounds` class is sufficient.

First, the `Objective` class,

```
class Objective:
    def __init__(self, img):
        self.img = img.copy()
        self.fcount = 0
    def F(self, dst):
        r,c = dst.shape
        Is = Image.fromarray(dst).filter(ImageFilter.FIND_EDGES)
        Is = np.array(Is)
        edgels = len(np.where(Is.ravel() > 20)[0])
        h = np.histogram(dst, bins=64)[0]
        p = h / h.sum()
        i = np.where(p != 0)[0]
        ent = -(p[i]*np.log2(p[i])).sum()
        F = np.log(np.log(Is.sum()))*(edgels/(r*c))*ent
        return F
    def Evaluate(self, p):
        self.fcount += 1
        a,b,c,k = p
        dst = ApplyEnhancement(self.img, a,b,c,k)
        return -self.F(dst)
```

The constructor stores a copy of the image to enhance and sets the function call counter to zero. The `Evaluate` method counts the objective function call, parses the particle position to extract the a, b, c, and k parameters, and applies them with a call to `ApplyEnhancement`, which we'll see in a bit. The enhanced image is returned in `dst` and passed to `F` to calculate the F score. As higher F is better, we return the negation.

The `F` method implements Equation 10.6. First, we use Pillow to apply an edge detector to the enhanced image to get `Is`. From `Is` we find `edgels` by asking how many pixels of the edge image are above 20. This gives us two of the three parts of Equation 10.6. The last part is the entropy.

We get that from the normalized histogram (p) ignoring empty bins where p is zero. This gives us ent. Finally, we return F by direct application of Equation 10.6.

The Objective class makes use of ApplyEnhancement to generate the candidate image,

```
from skimage.exposure import rescale_intensity
def ApplyEnhancement(g, a,b,c,k):
    def val(a,b,r,c):
        if (a<0) or (a>=r) or (b<0) or (b>=c):
            return False
        return True
    def valid3(g,i,j):
        r,c = g.shape
        v = []
        if val(i-1,j-1,r,c):  v.append(g[i-1,j-1])
        if val(i-1,j,r,c):    v.append(g[i-1,j])
        if val(i-1,j+1,r,c):  v.append(g[i-1,j+1])
        if val(i,j-1,r,c):    v.append(g[i,j-1])
        if val(i,j,r,c):      v.append(g[i,j])
        if val(i,j+1,r,c):    v.append(g[i,j+1])
        if val(i+1,j-1,r,c):  v.append(g[i+1,j-1])
        if val(i+1,j,r,c):    v.append(g[i+1,j])
        if val(i+1,j+1,r,c):  v.append(g[i+1,j+1])
        return np.array(v)
    def mean(g,i,j):
        return valid3(g,i,j).mean()
    def sigma(g,i,j):
        return valid3(g,i,j).std(ddof=1)

    rows,cols = g.shape
    dst = np.zeros((rows,cols))
    for i in range(rows):
        for j in range(cols):
            m,s = mean(g,i,j), sigma(g,i,j)
            dst[i,j] = ((k*g.mean())/(s+b))*(g[i,j]-c*m)+m**a
    return rescale_intensity(dst, out_range=(0,255)).astype("uint8")
```

ApplyEnhancement accepts the image (g) and the parameters represented by the current particle's position and returns the enhanced image. There are four embedded functions: val, valid3, mean, and sigma. The mean and sigma functions return the mean and standard deviation of the valid pixels in the 3x3 region with the current pixel, (i, j), in the center. The valid3 and val functions determine which pixels in the region are valid, meaning they exist. An alternative would have been to zero pad the input image. Here, we consider only existing image pixels.

The main part of ApplyEnhancement examines each image pixel in turn and, after determining the local mean and standard deviation, applies Equation 10.5 to determine the new pixel value in the output image, dst. Finally, we rescale dst to bring it into byte range and return it. Notice the inclusion of rescale_intensity from skimage.

If skimage is not already installed, the following should install it on most Linux distributions,

```
> sudo pip3 install scikit-image
```

The main function in enhance.py follows our usual format. We parse the command line (we'll see an example below) and then open and scale the image,

```
orig = np.array(Image.open(src).convert("L"))
img = orig / 256.0
```

Where we are working with images in $[0,1]$ and not $[0,255]$. When we create the final enhanced version, of course, we scale back to byte range.

Next, we set up the boundaries for the search,

```
b = Bounds([0.0,1.0,0.0,0.5], [1.5,22,1.0,1.5], enforce="resample")
```

where we restrict the search to $a \in [0,1.5]$, $b \in [1,22]$, $c \in [0,1]$, and $k \in [0.5,1.5]$ with resampling on boundary violations.

Next, we create the initializer based on the command line, set up the instance of the objective function passing in the scaled image (`img`), and start the search, this time explicitly showing the best F score on each iteration,

```
k = 0
swarm.Initialize()
while (not swarm.Done()):
    swarm.Step()
    res = swarm.Results()
    t = "    %5d: gbest = %0.8f" % (k,res["gbest"][-1])
    print(t, flush=True)
    s += t+"\n"
    k += 1
```

Output is stored in `s` and dumped to the output directory in the `README.txt` file when the search is complete,

```
res = swarm.Results()
pickle.dump(res, open(outdir+"/results.pkl","wb"))
s += "\nSearch results: %s, %d particles, %d iterations, %s\n\n"
        % (alg, npart, niter, itype)
s += "Optimization minimum %0.8f (time = %0.3f)\n"
        % (res["gbest"][-1], en-st)
s += "(%d best updates, %d function evaluations)\n\n"
        % (len(res["gbest"]), obj.fcount)
print(s)
with open(outdir+"/README.txt","w") as f:
    f.write(s)
```

Finally, we end the script by dumping the original and enhanced images to the output directory,

```
a,b,c,k = res["gpos"][-1]
dst = ApplyEnhancement(img, a,b,c,k)
Image.fromarray(dst).save(outdir+"/enhanced.png")
Image.fromarray(orig).save(outdir+"/original.png")
```

Let's run the script on each of our test images, those of Figure 10.7, and see how we fare by algorithm type.

10.3.3 The Results

A single run of `enhance.py` uses a command line like

```
> python3 enhance.py image.png 10 50 DE RI results
```

to enhance `image.png` and create an output directory, `results`, containing

```
enhanced.png
original.png
README.txt
```

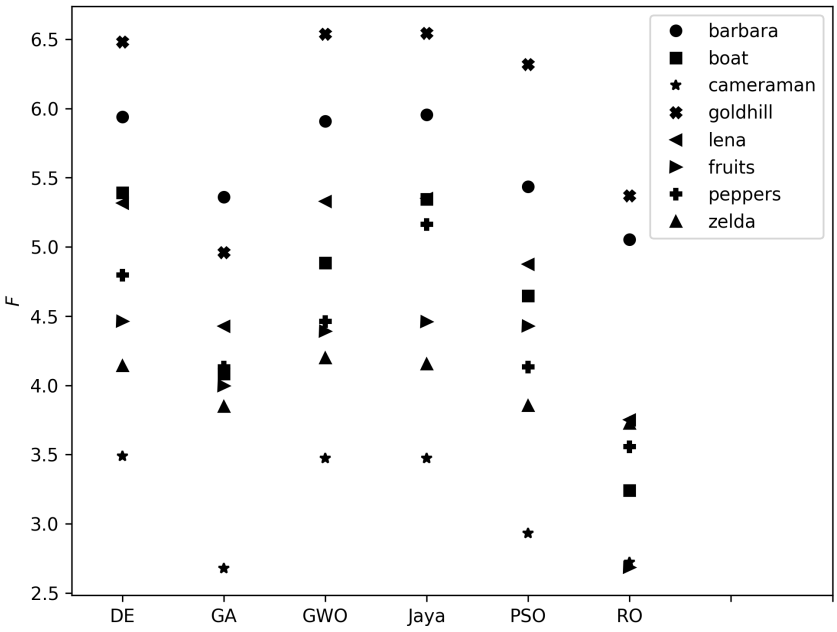


Figure 10.8: Image enhancement F scores by algorithm type and test image.

`results.pkl`

---

the enhanced image, original image, Python pickle of the swarm results, and the search text dumped to the console (`README.txt`).

The `process_images.py` script generates command lines to process each test image with each algorithm using a small swarm of ten particles for fifty iterations. We use random initialization of the swarm (`RI`), but the reader is encouraged to experiment with quasirandom (`QI`) and sphere (`SI`) initializations as well.

Figure 10.8 plots the best F value found by each algorithm for each test image. Recall, higher F scores imply a better enhancement. A first glance at Figure 10.8 marks GA and RO as the weaker performers.

There are definite ranges to the typical F values found. For example, the goldhill image produces, for almost all algorithms, the highest F values and cameraman the lowest. Doubtless, the content of the image is responsible for the ranges. The critical part for us to consider is the relative differences in F values found by algorithm. In most cases, Jaya, DE, and GWO produce the highest F values with PSO between them and GA and RO bringing up the rear.

What do the enhanced images look like? In the end, that's what matters if we're seeking to use swarm approaches to enhance grayscale images. Figure 10.9 shows the original and per algorithm enhanced version for the goldhill and boat test images.

The difference between the algorithm outputs is striking. And it's clear that for both images, the enhancement has, in many cases, improved the appearance of the image. This is particularly true for the DE, GWO, and Jaya images, a trend that holds for the other test images as well.

Let's rank the algorithms across all test images tallying how often the swarm algorithm appears in the top two positions by F score. This gives,

Jaya	7
DE	5
GWO	4
PSO	0
GA	0
RO	0

Figure 10.9: (Left) Original goldhill and boat images. (Right) Enhanced versions. Clockwise from upper left: DE, GA, GWO, RO, PSO, and Jaya.

showing us a decisive split between algorithms leading to nice enhancements (Jaya, GWO, DE) and algorithms that fail (PSO, GA, RO), though PSO consistently leads to a middle-of-the-road level of performance. Jaya is clearly victorious at this task.

However, are we being fair to GA? We know, unlike the other algorithms except perhaps RO, that GA takes longer to converge, longer to explore the space it's searching. And, we have only four parameters, not a large genome for GA to manipulate. What if instead of a short search favoring rapidly-converging algorithms, we took a longer view and let GA run for many more iterations and gave it a larger population to work with?

If we process each test image with GA using swarms of 20 particles instead of 10 and 500 iterations instead of 50, we do see considerable improvement in the final F value and a corresponding perceptual improvement in the enhanced image. For example, comparing the first set of GA F scores with the second set and Jaya gives,

	GA 10/50	*GA 20/500*	*Jaya 10/50*
barbara	5.3602	5.6461	5.9559
boat	4.0826	5.0791	5.3452
cameraman	2.6779	3.3829	3.4716
fruits	3.9964	4.3807	4.4596
goldhill	4.9597	6.4791	6.5423
lena	4.4269	5.2559	5.3502
peppers	4.1345	4.7982	5.1636
zelda	3.8476	4.0930	4.1558

proving that allowing GA to spend more time evolving and exploring has led to an improvement in the final F score, sometimes a significant improvement. However, none of the GA scores exceed those found by Jaya.

This chapter introduced three different experiments with images. In each case, we showed that swarms could accomplish the desired goal, but perhaps less efficiently than other, known techniques like using k-means to segment images. The image enhancement experiment is where the swarms were able to shine. Visually, the output of the enhancements was often quite pleasing and genuinely improved on the original image.

Our next chapter shifts from what we can see to what we can hear. Instead of manipulating pixel intensity values, we'll focus on music itself and see how swarms might be used as an aid to composers, or, at the very least, to implement some fun experiments and show us yet another example of how widely applicable swarm optimization can be.

Chapter 11

Music

Music is perhaps the most pleasant of human inventions. We live in a world of music with the twentieth century alone witnessing the development of more new musical genres than all of previous history.

One aspect of music garnering increasing attention is computer-generated music. Much of the current research focuses, rightly so, on the power of deep neural networks. However, it is possible to consider swarm-based approaches as well, hence the experiments of this chapter.

Using swarms to develop music isn't a new idea. A Google Scholar search for "swarm-based music generation" returned over 800 hits. So, we aren't claiming anything novel here, just one more example of the impressive range of topics to which swarm optimization is applicable.

Specifically, we'll first set the stage in terms of the additional software we need installed on our Ubuntu system to work with music files, including MIDI files. See Section 11.1. We'll also develop Bach and Irish slip jig music files, files we need for the experiments of Section 11.3.

We start in Section 11.2 by asking if a swarm can learn a single melody, if it can converge from a random state to the melody we give it. Spoiler alert: it can. Next, we build on this basic result by developing an objective function where the swarm attempts to learn to merge two separate melodies, thereby producing a novel melody with aspects of both.

Section 11.3 presents experiments in learning a melody in the same vein as a group of existing melodies. Here is where we'll use the Bach and slip jig melodies of Section 11.1. Can a swarm learn a new melody that sounds like Bach? We'll find out if it can, and if so, how well.

For the ultimate set of experiments, those of Section 11.4, we attempt to learn a pleasant melody from scratch. To do this, we'll develop a multi-component objective function. For example, one component will drive the swarm towards notes that are part of the selected musical mode (think major or minor scale). We won't win any awards for computer-generated music. Still, we'll have some fun with the idea – we know the concept of "pleasant melody" is one that might be quite difficult, if not impossible, to quantify. However, we won't let a simple impossibility stop us from trying.

11.1 Setting the Stage

Throughout the book, we've focused on basic Python code, so while we've mentioned expecting an Ubuntu-style Linux system as our operating environment, we've really only relied, primarily, on access to the NumPy library, a library well-supported on Macintosh and Windows platforms. In this chapter, however, we need some specific Linux tools to handle MIDI files, a music file format that works with music hardware like keyboards, and tools to change MIDI files into musical scores. We'll install the Ubuntu versions here and hope that enterprising readers who use other platforms will find the same or equivalent tools. I suspect Macintosh users will have no trouble doing this. Windows users might consider setting up an Ubuntu 18.04 or later virtual machine instead.

11.1.1 Tools

We need to install the following tools for use from the command line and Python. The Ubuntu installation commands are

Package	*Installation*
`midiutil`	`sudo pip3 install midiutil`
`wildmidi`	`sudo apt-get install wildmidi`
`musescore`	`sudo apt-get install musescore`
`mftext`, `abc2mid`	`sudo apt-get install abcmidi`

The commands above work for Ubuntu 18.04. If using Ubuntu 20.04 or later, replace `musescore` with `musescore3` during install and in the source code later in the chapter.

We'll use the `midiutil` Python library to turn swarm-generated melodies, NumPy vectors, into proper MIDI files on disk. MIDI stands for "Musical Instrument Digital Interface" and is the standard used by musical devices. We'll use MIDI files as output and, when building melodies, as input. We'll describe the process as needed while we go through the experiments.

If we generate MIDI files, which use a `.mid` extension, we need some way of playing the files. Standard audio programs won't work; the MIDI file doesn't contain audio data. Instead, it includes instructions to musical instruments on how to produce audio output, be it a single melody line, like the files we'll work with, or an entire orchestra playing a full score.

To play a MIDI file from the command line, we need a program to translate MIDI instructions to audio – this is precisely what `wildmidi` does. For example, if we had an output file, say `melody_DE.mid`, we could listen to the file with,

```
> wildmidi melody_DE.mid
```

There are options at the keyboard we can use when the MIDI file is playing, but our generated files are only a few seconds long, so fast-forward and rewind really aren't helpful.

`Musescore` is a powerful, open-source tool for scoring music. We'll use it from the command line in silent mode to turn a MIDI file into a PNG image representing the score. We'll be able to see as well as hear the output of our experiments. If you are a composer, you might enjoy the capabilities the GUI interface to `musescore` provides.

We need two additional command-line tools to help us construct the NumPy melody files used in Section 11.3. These are `mftext` and `abc2midi`, both of which are in the `abcmidi` package. We'll describe the tools when we need them.

11.1.2 Building Melodies

Our goal in this chapter is to generate melodies. However, to do that, we'll sometimes need existing melodies. For us, a melody is a NumPy vector, a set of pairs where each pair is a MIDI note number, think note on a piano keyboard, and a duration, a fraction of a quarter note.

The website associated with this book has included several MIDI (`.mid`) and NumPy (`.npy`) files for you. The NumPy files, `mary.npy`, `happy.npy`, and `ode.npy`, were created by hand to represent the familiar tunes "Mary Had A Little Lamb," "Happy Birthday to You,"[1] and Beethoven's "Ode to Joy". We'll use these in Section 11.2.

The `bach` directory contains ten MIDI files and associated NumPy files; the opening melodies for ten of J. S. Bach's chorales. The MIDI files were created by composer Paul Kneusel (see `www.paulkneusel.com`) and are in the public domain. Special thanks are due to Paul for his considerable help with the objective function of Section 11.4. We'll use the Bach pieces in Section 11.3.

The Bach pieces are one collection of similar melodies. A second collection is a set of Irish slip jigs from the Nottingham Music Database (NMD). Because of licensing requirements, you'll need to build the NumPy versions of these pieces yourself using the directions below.

[1] On June 28, 2016, "Happy Birthday to You" was legally declared to be in the public domain in the United States.

First, download the slip jigs in abc format from

```
http://abc.sourceforge.net/NMD/nmd/slip.txt
```

ABC is a text music notation format. See the main NMD page at

```
http://abc.sourceforge.net/NMD/
```

for a description of the database and links to information about the ABC format.

The `slip.txt` file combines 11 slip jigs in one file. The separation between the pieces is easy to discern. With a text editor, separate the one file into 11 `.abc` files with the following names matching the order in `slip.txt`

```
brandy.abc
dublin_streets.abc
gingerhog.abc
stout.abc
kid_mountain.abc
lamppost.abc
racehorse.abc
rocky_road.abc
coverley.abc
slip_jig.abc
staggering.abc
```

Then, convert the ABC files to MIDI files individually with

```
> abc2midi brandy.abc
```

ignoring any messages from `abc2midi`. When done, you'll have a set of `.mid` files with the same names as the `.abc` files plus a number after the base name corresponding to the position in the original `slip.txt` file.

Play the MIDI versions with `wildmidi`. You'll hear the entire piece, including multiple instruments. For our purposes, we need only the first line, or channel in MIDI-speak. This is where the `mftext` utility comes into play, though we need not use it manually. Instead, copy `midi_dump.py`, included on the book website, into the directory where the `.mid` files reside and run it,

```
> python3 midi_dump.py *.mid
```

to produce the final `.npy` files used in Section 11.3.

Our tools and melody files are in place. Let's run some experiments.

11.2 Learning and Merging Melodies

A swarm can learn a simple melody. Let's see how in this section. Along the way, we'll introduce functions for dealing with music and music files. This chapter's experiments are more complex than any others in the book, so we'll start at the beginning and build from there.

The code we'll develop and use in this section is in the `melody_merge.py` file. Several functions in this file will be used in the other experiments, so we'll describe them here.

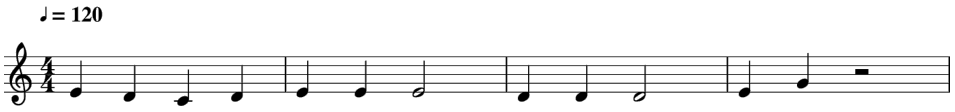
Our task is easy to state: we have a melody, and we want to initialize a swarm and get it to learn the melody. Of course, there is no practical utility in this; we already have the melody; view the experiment as a stepping stone to using swarms to generate novel melodies. We'll see one way to do that in this section as well.

We have three melody files to work with, `mary.npy`, `happy.npy`, and `ode.npy`. All three files are NumPy vectors interpreted as (note, duration) pairs. For example, here's `mary.npy`,

```
64,1; 62,1; 60,1; 62,1; 64,1; 64,1; 64,2; 62,1; 62,1; 62,2; 64,1; 67,1
```

where the vector has been separated into pairs of numbers. The first number is the MIDI note number. Each semitone is represented in a MIDI file as an integer. We'll work with notes from 35 through 85 though we'll interpret note 35 as a rest.

The second number is the duration of the note measured in quarter notes. The melody consists of quarter notes (duration 1) and a half note (duration 2). The corresponding score is



From the swarm's perspective, its goal is to learn the vector, the pattern of notes and durations.

Let's walk through `melody_merge.py` in detail. First, we import the usual modules, including a new one

```
from midiutil import MIDIFile
```

This gives us the ability to write MIDI files to disk from Python. We installed `midiutil` in Section 11.1.

We intend to represent notes as pairs: MIDI note number and number of quarter notes. Therefore, we'll constrain the MIDI note numbers to $[35, 85]$ and the durations to $[0.25, 2]$ to allow notes as short as a 1/16-th note and as long as a half note. Additionally, we need to make the MIDI note number an integer and round the duration to the nearest 0.25. The latter requirement makes the notes even 1/16-th, 1/8-th, quarter, and half notes. In code, then, we need a custom `MusicBounds` class,

```
class MusicBounds(Bounds):
    def __init__(self, lower, upper):
        super().__init__(lower, upper, enforce="resample")
    def Validate(self, p):
        i = 0
        while (i < p.shape[0]):
            note, duration = p[i:(i+2)]
            p[i] = int(note)
            p[i+1] = np.round(4*duration)/4
            i += 2
        return p
```

The only method to implement is `Validate`. In `Validate`, we loop over the notes and durations in the position vector, `p`, making the note an integer and rounding the duration to the nearest 0.25.

Next, we need an objective function. We'll minimize the squared error between a candidate position vector and the melody vector. However, we'll add a twist, the ability to balance between the squared error of two different melodies. If we give `melody_merge.py` a single melody, the swarm will attempt to learn it by minimizing the squared error. If given *two* melodies, the swarm will balance between minimizing the error to both. Additionally, we add `alpha` to shift the balance between the first and second melodies. If `alpha` is zero, the swarm will focus exclusively on the second melody. If `alpha` is one, the focus is on the first melody.

Let's look at the code,

```
class MusicObjective:
    def __init__(self, template1, template2=None, alpha=0.5):
        if (template2 is None):
            template2 = template1
            alpha = 1.0
```

```
        if (len(template1) < len(template2)):
            template2 = template2[:len(template1)]
        if (len(template2) < len(template1)):
            template1 = template1[:len(template2)]
        self.template1 = template1
        self.template2 = template2
        self.alpha = alpha
    def Evaluate(self, p):
        d1 = ((p - self.template1)**2).sum()
        d2 = ((p - self.template2)**2).sum()
        return self.alpha*d1 + (1.0-self.alpha)*d2
```

Contrary to many of our other objective functions, `MusicObjective` spends more lines on the constructor than `Evaluate`. The constructor expects two templates, two melody vectors loaded from NumPy files. The second template is optional. Additionally, a default of 0.5 is set for `alpha` to focus on both melodies equally.

The rest of the constructor decides what to do if only one melody is given, namely, set the second to the first and make `alpha` one. As we'll see, this eliminates one of the terms in the objective function value. Next, if the melodies are of different lengths, the longer is cut to the length of the shorter.

The `Evaluate` method calculates two squared errors, first between the particle position in `p` and the first melody vector (`d1`) and then with the second (`d2`). The sum of the two squared errors, weighted by `alpha`, is then returned.

Before discussing the main part of `melody_merge.py`, we need to define a few helper functions: `StoreMelody`, `DisplayMelody`, and `PlayMelody`. We'll use these functions, with slight variation, in all the experiments of this chapter. The first writes a melody to disk as a MIDI file. The second displays a melody at the console, and the third is a wrapper to play a melody in code. The last two are self-evident, so we'll only present the first one here,

```
def StoreMelody(p, fname):
    tempo = 120
    volume = 100
    m = MIDIFile(1)
    m.addTempo(0, 0, tempo)
    m.addProgramChange(0, 0, 0, 0)  # acoustic piano
    i = 0
    t = 0.0
    while (i < len(p)):
        note, duration = p[i:(i+2)]
        i += 2
        if (note == 35):
            m.addNote(0, 0, 21, t, duration, 0) # rest
        else:
            m.addNote(0, 0, int(note), t, duration, volume)
        t += duration
    with open(fname, "wb") as f:
        m.writeFile(f)
```

The essence of `StoreMelody` is the creation of a `MIDIFile` object, `m`, and the addition of notes with their duration, the `while` loop. We fix the tempo and volume and set the MIDI output for an acoustic piano sound. Other instruments are possible, but the piano sounds the best for the simple melodies we'll be working with. With all notes added, we write the MIDI file to the given output filename, `fname`. Notice the check for MIDI note 35. If present, the volume is set to zero to play the note as a rest.

The main function in `melody_merge.py` interprets the command line, sets up the bounds and objective function, creates the swarm object, and calls `Optimize`. The results are put in the output

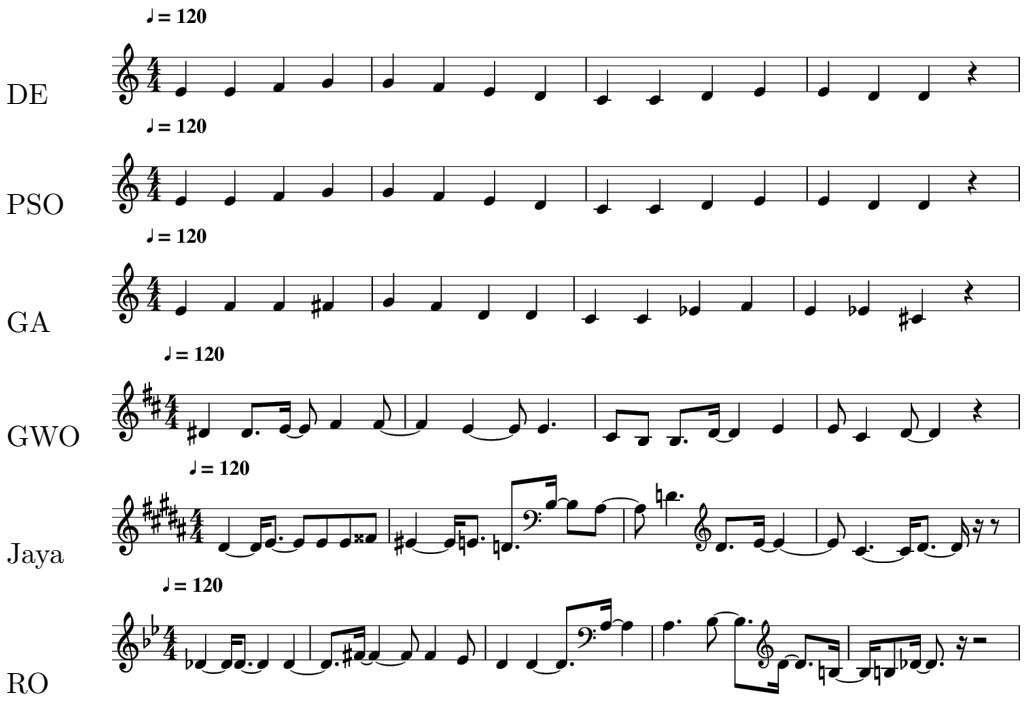


Figure 11.1: Learning "Ode to Joy" by algorithm using 20 particles, 1500 iterations.

directory, including `results.pkl`, the swarm results, and `melody.mid`, the MIDI version of the melody.

To test the code, we'll use command lines like this,

```
> python3 melody_merge.py ode.npy none 0.5 results 20 1500 DE
```

to learn "Ode to Joy" using DE with 20 particles and 1500 iterations. The alpha value of 0.5 is ignored because there is no second melody.

The search runs quickly, about five seconds. In this case, the swarm learns the melody correctly. Repeating the search for each algorithm and generating scores from the MIDI files using `musescore`, we'll see how later in the chapter, produces Figure 11.1.

The figure is organized, from top to bottom, by how well the algorithm learned the melody. Both DE and PSO converged on the exact melody while GA was close. GWO deviated still more with Jaya and RO off in the weeds.

How much longer does the swarm need to search for GA, GWO, Jaya, and RO to converge? For GA, moving to 4500 iterations did the trick. However, for Jaya, GWO, and RO, even 40,000 iterations were not enough for the swarm to find the proper melody. It's possible the strict discretization of the durations threw Jaya and GWO, but it's not certain that is the cause. GWO did get close, though the timing of notes was rather odd.

Of course, suppose we are interested in swarm techniques as an aid to composition. In that case, we don't want to converge precisely to the source melody but instead learn something similar, a variation. We'll do just that in Section 11.3. For now, we've demonstrated that swarms can learn melodies. Let's see what happens when we force the swarm to learn something between two melodies.

To learn something between two melodies, we need to add a second melody to the command line,

```
> python3 melody_merge.py mary.npy ode.npy 0.5 results 20 10000 PSO
```

This command attempts to find a melody balanced between "Mary Had A Little Lamb" and "Ode to Joy". Note, we increase the iteration count to 10,000 and use PSO. Let's run this for different alpha values: 0.1, 0.5, and 0.9. The first should favor "Ode to Joy" while the last should favor "Mary Had A Little Lamb." Figure 11.2 shows us what results.

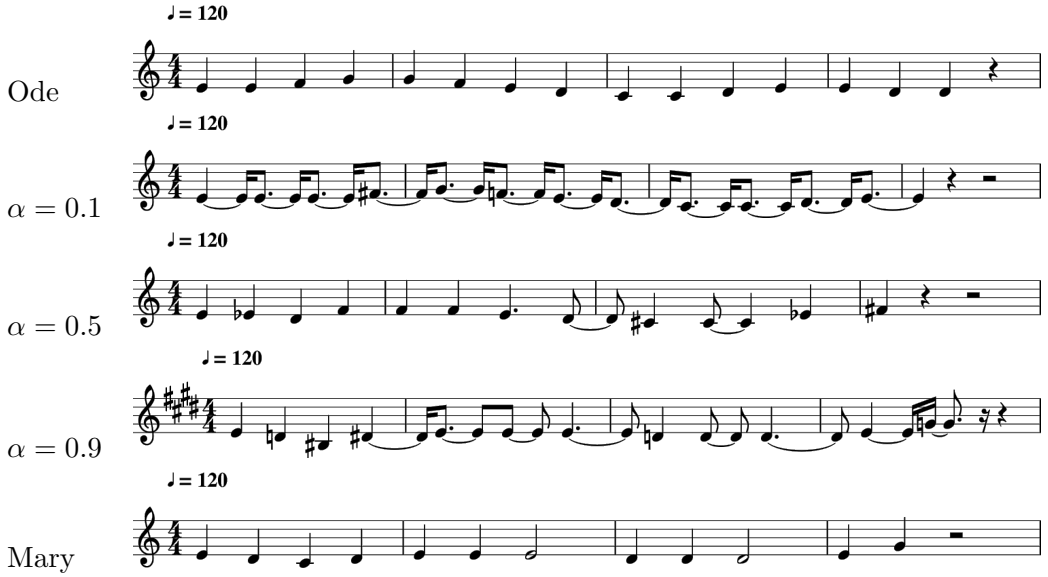


Figure 11.2: Gradation in melody between "Ode to Joy" and "Mary Had A Little Lamb."

When $\alpha = 0.9$ we expect the learned melody to be close to "Mary Had A Little Lamb", and, though somewhat strange in appearance, it is, but the key is wrong. Similarly, when $\alpha = 0.1$, we expect something close to "Ode to Joy", which, again, while strange looking is by sound close. At $\alpha = 0.5$ we hope something equidistant from both. An argument could be made that the melody is just that.

If you are like me and not used to reading musical scores, please listen to the MIDI files; they are included on the book's website. Or, rerun the code yourself to experiment with parameters and alpha values, which is even more helpful at building intuition.

Our metric is squared error. Therefore, we expect the Euclidean distance between the swarm best vector and the melody vectors from `mary.npy` and `ode.npy` to reflect this fact. Indeed, the distances between the swarm best and the melodies are

Melody	$\alpha = 0.1$	$\alpha = 0.5$	$\alpha = 0.9$
Mary	8.310	4.528	1.090
Ode	1.436	5.050	8.842

These track exactly as we expect. When $\alpha = 0.1$, there is little emphasis placed on "Mary Had A Little Lamb" and almost all on "Ode to Joy" and we see this as the distance to the first is greater. For $\alpha = 0.9$, the situation is reversed, again clearly seen in the distance. Finally, for $\alpha = 0.5$, the distance is roughly the same; the swarm has converged to a position in the search space nearly halfway between the two melodies.

Both PSO and DE learn the same final melody when $\alpha = 0.5$ and the inputs are "Mary Had A Little Lamb" and "Happy Birthday." Multiple runs produce virtually identical output regardless of the random initializer's configuration. For five runs of DE, we get

62,0.75 61,0.75 61,1.00 61,1.00 **65**,1.00 64,1.50 62,1.25 61,0.75 62,1.00 61,1.50 66,1.00 66,1.00
62,0.75 61,0.75 61,1.00 61,1.00 64,1.00 64,1.50 62,1.25 61,0.75 62,1.00 61,1.50 **65**,1.00 66,1.00

62,**1.00** 61,0.75 61,1.00 61,1.00 64,1.00 64,1.50 62,1.25 61,0.75 62,1.00 61,1.50 66,1.00 66,1.00
62,0.75 61,0.75 61,1.00 61,1.00 64,1.00 64,1.50 62,**1.50** 61,0.75 62,1.00 61,1.50 66,1.00 66,1.00
62,0.75 61,0.75 61,1.00 61,1.00 64,1.00 64,1.50 62,1.25 61,0.75 62,1.00 61,1.50 **65**,1.00 66,1.00

These results are virtually identical except for single changes in note number or duration from run to run shown in **bold**. This indicates that the search space, discretized in note number and duration, has a specific minimum for $\alpha = 0.5$, and both DE and PSO can find it repeatedly.

11.3 Learning Similar Melodies

In Section 11.2, we explored merging two melodies by minimizing the joint Euclidean distance between the melody vectors. In this section, we'll continue in a similar vein, but instead of minimizing the distance jointly between two melodies, we'll measure the mean squared error between the current swarm particle position and a randomly selected melody from a set of similar melodies.

The code we'll work with is in `melody_match.py`. Here's where we use the slip jig and Bach chorale melodies from Section 11.1. Let's walk through the main parts of the code focusing on boundaries (`MusicBounds`) and the objective function (`MusicObjective`), which differ from the versions used by `melody_merge.py` in Section 11.2.

We start with `MusicBounds`,

```
class MusicBounds(Bounds):
    def __init__(self, lower, upper):
        super().__init__(lower, upper, enforce="resample")
    def Validate(self, p):
        i = 0
        while (i < p.shape[0]):
            note, duration = p[i:(i+2)]
            p[i] = int(note)
            p[i+1] = int(duration*10)/10
            i += 2
        return p
```

The purpose here is to discretize the MIDI note numbers, as before. We discretize the durations as well, but instead of forcing to the nearest 0.25, we round to the nearest tenth. Doing this more closely matches the durations found in the slip jig and Bach melodies.

The most interesting part of `melody_match.py` is the objective function. We supply the objective object with a database of existing melodies, a list of note, duration pairs. Then, on each call to `Evaluate`, the mean squared error between the candidate particle position and a randomly selected member of the set of melodies is returned. In code,

```
class MusicObjective:
    def __init__(self, db):
        self.fcount = 0
        self.db = db
    def Evaluate(self, p):
        self.fcount += 1
        n = np.random.randint(0, len(self.db))
        b = self.db[n][:len(p)].copy()
        b[1::2] = b[1::2]*30
        a = p.copy()
        a[1::2] = a[1::2]*30
        return np.sqrt(((b-a)**2).sum()/len(a))
```

A random melody is selected from the database (`db`, `n`) and the mean squared error between it and the particle position (`p`) is returned. Notice that the durations are multiplied by 30 before calculating the MSE. Doing this makes the durations roughly equal to the MIDI note numbers. Without this

scale factor, the MSE is dominated by note differences with duration differences counting for only a tiny portion of the error sum.

When `melody_match.py` starts, it loads the similar melody database by calling `LoadDatabase` which looks for all NumPy files in the given directory,

```
def LoadDatabase(dbdir):
    dnames = [dbdir+"/"+i for i in os.listdir(dbdir)
              if i.find(".npy") != -1]
    dnames.sort()
    db = []
    for name in dnames:
        db.append(np.load(name))
    return db
```

The remainder of `melody_match.py` is quite similar to `melody_merge.py`. The main function parses the command line,

```
> python3 melody_match.py 20 results 25 130000 PSO RI bach
```

to run a search generating 20 notes with a swarm of 25 particles and 130,000 iterations using randomly initialized PSO. The source melodies are the `.npy` files in the `bach` directory.

To get a feel for how the swarms perform, run `experiment_match.py` which is essentially

```
for alg in ["RO","DE","PSO","GWO","JAYA","GA"]:
    for run in range(5):
        cmd = "python3 melody_match.py 20 results/match/jigs/%s_run%d 25
               130000 %s RI NMD" % (alg.lower(),run,alg)
        os.system(cmd)

for alg in ["RO","DE","PSO","GWO","JAYA","GA"]:
    for run in range(5):
        cmd = "python3 melody_match.py 20 results/match/bach/%s_run%d 25
               130000 %s RI bach" % (alg.lower(),run,alg)
        os.system(cmd)
```

It runs each algorithm five times on the slip jigs and Bach chorales.

When complete, play each of the melodies using `wildmidi`. Some general observations from my run of the code for the slip jigs are:

- RO produced random melodies with some interesting sequences.
- Jaya melodies were less random sounding than RO, but still not coherent.
- GWO was much like Jaya with quick bursts of notes indicating clusters of short durations.
- GA produced some short but nice sequences without quick bursts of notes like GWO.

Of particular interest is what happened with PSO and DE. For these algorithms, the swarm converged on a single melody, one of the 11 slip jigs, namely "Dublin Streets." DE learned the melody almost note for note while PSO produced melodies strongly influenced by "Dublin Streets." Why "Dublin Streets" when all 11 melodies were equally likely to be selected for each call to `Evaluate`? The exact cause is unclear, but we know from experience DE, and to a lesser degree PSO, tend to converge quickly. DE's propensity to fall into local minima might help explain things as well. There might be a strong basin of attraction around this particular melody that DE quickly finds, abandoning exploration for exploitation.

Do we see, or rather, hear the same effect with the Bach chorales? Listening to the generated melodies leads us to much the same set of comments for RO, Jaya, GWO, and GA as above. Likewise, PSO and DE are more interesting.

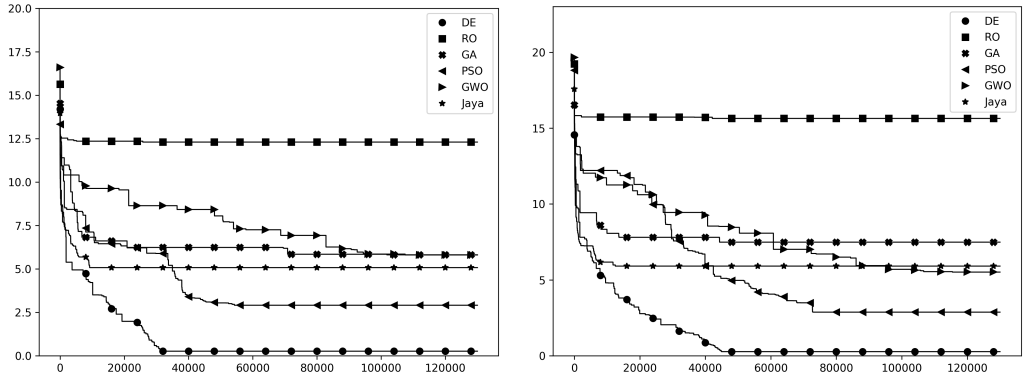


Figure 11.3: Evolution of the swarm best objective function value for slip jigs (left) and Bach chorales (right).

Unlike the slip jigs, the chorales appear to have multiple "local minima." For PSO, the runs were split between melodies much like `Bach_Chorale_5` and `Bach_Chorale_2`. For DE, results were split between `Bach_Chorale_5` and `Bach_Chorale_4`.

Figure 11.3 shows the evolution of the swarm best for the slip jigs (left) and Bach chorales (right). In both cases, RO failed to improve, and DE converged rapidly to almost no error, consistent with collapsing to a specific melody. The algorithms performed much the same way for both sets of melodies. PSO did second best but didn't reach the same low MSE as DE. GWO, Jaya, and GA all end up in much the same place as well.

What does all of this mean? In terms of learning, DE performs well, followed by PSO. We've seen similar results before. In terms of using swarms to learn interesting variations, DE is probably not the best option. PSO, or perhaps one of the others if run for a longer period of time, might produce pleasing results, an experiment for the reader.

11.4 Learning Melodies from Scratch

The experiments of Section 11.2 and Section 11.3 attempted to generate a new melody by using existing melodies as references. In this section, we dispense with pre-existing melodies and instead seek to learn a pleasant, or at least reasonable, melody from scratch.

To do this, we maintain most of the framework we've used in this chapter and focus on a novel objective function, one that can point us towards how pleasant a melody is.

It should be uncontroversial to state that what makes a melody pleasant is subjective. To rein in the subjectivity, we seek a multi-part objective function that measures characteristics of a melody generally believed to make it more pleasant sounding, at least for Western music.

The first rule of thumb for melodies is the range: they should not cover more than 18 steps (semitones). A semitone is a half-step, from one white key on a piano to the next key to the right, black or white. So, we need a term to penalize the melody if its range is too great.

Melodies using notes that are part of the desired musical mode are also more likely to sound pleasant. The first note in a melody is the root, and the remaining notes should be those used by a specific musical mode starting with that root. These seven modes, central to the tradition of Western music, set the overall tone. For us, modes are a set of rules, a sequence of intervals from the root to the next root one octave higher. We need a term that measures how far a given melody is from the mode we've selected.

If the swarm is entirely free to select note durations, we might end up with awkward ones. We'd like to favor quarter, half, and whole notes, though not with equal weighting. We need a term to characterize the notes durations and how similar they are to the desired quarter, half, and whole notes.

Mode	*Intervals*	*Characteristics*
Ionian (Major)	W W H W W W H	bright, positive, strong, simple
Aeolian (Minor)	W H W W H W W	sad
Dorian	W H W W W H W	light, cool, jazzy
Lydian	W W W H W W H	bright, airy, sharp
Mixolydian	W W H W W H W	blues, celtic
Phrygian	H W W W H W W	dark, depressing
Locrian	H W W H W W W	darker still, "evil"

Table 11.1: The modes and their characteristics.

Finally, a melody is played one note at a time. The difference between note i and note $i+1$ is the interval. As we're using MIDI notation, integer values assigned to each semitone, we'll consider the integer difference between two notes. Not all intervals sound good. It's generally agreed that thirds and fifths sound pleasant. Thirds are three or four semitones apart (minor or major thirds) while fifths are seven apart. Additionally, the intervals should match notes in the desired mode. We need a term that tells us about the intervals in a melody.

All in all, we will develop a four-part objective function. The sum of the four parts is what we seek to minimize with the hope that a small value implies a pleasant-sounding melody in the desired mode.

For reference, Table 11.1 presents the modes giving their Greek names, intervals from the root, and adjectives describing what melodies in that mode often sound like. The intervals tell us how to step from the root to the notes of the mode with `W` a whole step (two semitones) and `H` a half step (one semitone). For example, if the root is MIDI note 60, middle C, and the mode is Ionian with a sequence of `W W H W W W H`, the notes of the scale are,

C	w	**D**	w	**E**	h	**F**	w	**G**	w	**A**	w	**B**	h	**C**
60	→	62	→	64	→	65	→	67	→	69	→	71	→	72

which is nothing more than the standard C major scale. Our objective function will look for notes in this scale, if the root is C and the mode is Ionian.

11.4.1 The Code

The code for this experiment is in `melody_maker.py`. As before, we use a custom Bounds class,

```
class MusicBounds(Bounds):
    def __init__(self, lower, upper):
        super().__init__(lower, upper, enforce="resample")
    def Validate(self, p):
        i = 0
        while (i < p.shape[0]):
            note, duration = p[i:(i+2)]
            p[i] = int(note)
            p[i+1] = np.floor(duration)
            i += 2
        return p
```

where we ensure integer MIDI note numbers and truncated durations. We create the bounds object with

```
note_lo = 35
note_hi = 84
dur_lo  = 1
dur_hi  = 5
```

```
lower = [note_lo, dur_lo] * ndim
upper = [note_hi, dur_hi] * ndim
b = MusicBounds(lower, upper)
```

to limit durations to $[1, 5]$. When generating the MIDI output files, we'll multiply all durations by a global value, `M = 0.5`, meaning durations are limited to eighth to whole notes.

Let's define the `MusicObjective` class piece by piece. The constructor is straightforward,

```
class MusicObjective:
    def __init__(self, note_lo, note_hi, mode="major"):
        self.mode = mode
        self.fcount = 0
        self.lo = note_lo
        self.hi = note_hi + 1
```

where we set note limits, the objective function call counter, and the desired musical mode.

Next, the `Evaluate` method,

```
    def Evaluate(self, p):
        self.fcount += 1
        score = self.Distance(p[::2], self.mode)
        dur = self.Durations(p)
        R = self.CheckRange(p)
        v,t = self.Intervals(p[::2], self.mode)
        return R + v + t + score + dur
```

which counts the objective function call and computes the four elements we outlined above: melody distance from the desired mode (`Distance`), the note durations (`Durations`), the melody range (`CheckRange`), and finally, the note intervals (`Intervals`). We'll detail each of these below. The objective function value returned by `Evaluate` is the sum of the values returned by each of these functions.

Let's start with `CheckRange`,

```
def CheckRange(self, p):
    notes = p[::2]
    lo = notes.min()
    hi = notes.max()
    return 0 if (hi-lo) <= 18 else 1
```

to get the high and low MIDI note numbers and ask if that range is less than or greater than 18. If it is greater, return one; otherwise, zero. The binary return value is a strong signal to the swarm that the range should be within 18 semitones.

Next, we look at the durations of the notes,

```
def Durations(self, p):
    d = p[1::2].astype("int32")
    dp = np.bincount(d, minlength=8)
    b = dp / dp.sum()
    a = np.array([0,0,100,0,60,0,20,0])
    a = a / a.sum()
    return np.sqrt(((a-b)**2).sum())
```

This function gets the note durations (`d`). Recall, these are integers in $[1, 5]$. The histogram is created (`dp`) and normalized to probabilities (`b`). What should we do with this distribution? We want to compare it to the distribution of durations we'd like to see. Above, we stated we want primarily quarter, half, and whole notes with a strong emphasis on quarter, then half, and last of all, whole. Consider the vector, `a`. It has as many elements as `b` and is normalized to be a probability distribution, like `b`.

For the range we set up, and the multiplier on durations of `M = 0.5`, quarter notes are element two, half notes are element four, and whole notes element six. These elements of `a` are, arbitrarily, set to 100, 60, and 20, meaning we'd like to see the ratio between quarter, half, and whole notes be 100:60:20 → 5:3:1. A natural measure between `a` and `b` is the squared error, so we return it as our metric. Another choice for metric might have been the KL-divergence to see, in a probabilistic sense, how much `b` is like `a`. A desire for clarity in the presentation leaves the computation of `a` where it is. It would be faster to define `a` once in the `MusicObjective` constructor and only refer to it in `Durations`.

We have two parts of the objective function yet to define, `Distance` and `Intervals`. However, both of these depend upon the selected musical mode. Specifically, we need to know which notes in the set of allowed MIDI notes are notes that would be seen in the given mode for the given root of the melody, the first note of the melody. To find those, we need a helper function, `ModeNotes`.

`ModeNotes` is rather long, so we'll work with it in pieces. The first part is,

```
def ModeNotes(self, notes, mode):
    modes = {
        "ionian":     [2,2,1,2,2,2,1],
        "dorian":     [2,1,2,2,2,1,2],
        "phrygian":   [1,2,2,2,1,2,2],
        "lydian":     [2,2,2,1,2,2,1],
        "mixolydian": [2,2,1,2,2,1,2],
        "aeolian":    [2,1,2,2,1,2,2],
        "locrian":    [1,2,2,1,2,2,2],
        "major":      [2,2,1,2,2,2,1],
        "minor":      [2,1,2,2,1,2,2],
    }
    m = modes[mode.lower()]
    A = np.zeros(self.hi-self.lo+1)
    for i in range(notes.shape[0]):
        A[int(notes[i]-self.lo)] = 1
```

The goal of `ModeNotes` is to return a binary vector representing MIDI notes from 36 through 85. If the corresponding element of the vector is one, that note is part of the mode with the first note of the melody in `notes` as the root. If the note shouldn't be in the mode, its entry is zero.

This part of the function defines the steps between notes by mode. We saw above that a whole step adds 2 to the MIDI note number while a half step adds 1. Therefore, the proper sequence of intervals for each mode is defined first. The sequence is indexed by the actual `mode` and assigned to `m` to be used later in the function.

We intend to generate a binary vector of the notes in the mode. We also want a binary vector of the notes actually in the melody. That's what we set up in `A`. We loop over notes and set those in the melody to one, leaving the rest as zero.

The next part of `ModeNotes` builds `B`, a vector like `A`, but marking the notes that are part of the mode,

```
    B = np.zeros(self.hi-self.lo+1)
    note = int(notes[0])
    while (note <= self.hi):
        if (note <= self.hi):
            B[note-self.lo] = 1
        note += m[0]
        if (note <= self.hi):
            B[note-self.lo] = 1
        note += m[1]
        if (note <= self.hi):
            B[note-self.lo] = 1
        note += m[2]
        if (note <= self.hi):
```

```
            B[note-self.lo] = 1
        note += m[3]
        if (note <= self.hi):
            B[note-self.lo] = 1
        note += m[4]
        if (note <= self.hi):
            B[note-self.lo] = 1
        note += m[5]
        if (note <= self.hi):
            B[note-self.lo] = 1
        note += m[6]
        if (note <= self.hi):
            B[note-self.lo] = 1
```

The root is `note`, the first note of the melody. Next comes a loop from this MIDI note number to the maximum note number. For each possible MIDI note number, we ask if we are in range. If we are, we set the element of `B`. Note the use of `m` to add the proper increment to the note to move to the next note of the mode.

Finally, we repeat this process from the root down to the lowest MIDI note number,

```
    note = int(notes[0])
    while (note >= self.lo):
        if (note >= self.lo):
            B[note-self.lo] = 1
        note -= m[6]
        if (note >= self.lo):
            B[note-self.lo] = 1
        note -= m[5]
        if (note >= self.lo):
            B[note-self.lo] = 1
        note -= m[4]
        if (note >= self.lo):
            B[note-self.lo] = 1
        note -= m[3]
        if (note >= self.lo):
            B[note-self.lo] = 1
        note -= m[2]
        if (note >= self.lo):
            B[note-self.lo] = 1
        note -= m[1]
        if (note >= self.lo):
            B[note-self.lo] = 1
        note -= m[0]
        if (note >= self.lo):
            B[note-self.lo] = 1
    return A,B
```

where we use `m` in reverse order as the sequence of increments it contains goes from the root to one octave higher. When the loop is complete, we return `A` and `B`. The `Distance` and `Intervals` functions use these vectors in different ways.

The `Distance` function computes the Hamming distance between the two binary vectors returned by `ModeNotes`. In code,

```
def Distance(self, notes, mode):
    A,B = self.ModeNotes(notes, mode)
    lo = int(notes.min() - self.lo)
    hi = int(notes.max() - self.lo)
    a = A[lo:(hi+2)]
```

```
    b = B[lo:(hi+2)]
    score = (np.logical_xor(a,b)*1).sum()
    score /= len(a)
    return score
```

where we limit `A` and `B` to just the range of MIDI note numbers actually used by the melody.

The Hamming distance between two binary vectors (or integers) is the number of corresponding bits that are different. For example,

```
A         : 1 0 0 1 1 0 1 0 1 0 1 1
B         : 1 1 0 1 0 0 1 0 1 0 1 0
different:   1     1             1
```

meaning the Hamming distance between `A` and `B` is three. Note, the Hamming distance is the number of one bits in the exclusive-OR of `A` and `B`. The `Distance` function scales the Hamming distance by the length of the vectors to turn it into a fraction where identical vectors return zero, and completely different vectors return one.

The `Intervals` function looks at the intervals, the difference between note i and note $i+1$ counting the number that are valid for the mode and whether the interval is a third (minor or major), or a fifth. In code,

```
def Intervals(self, notes, mode):
    _,B = self.ModeNotes(notes, mode)
    valid = minor = major = fifth = 0
    for i in range(len(notes)-1):
        x = int(notes[i]-self.lo)
        y = int(notes[i+1]-self.lo)
        if (B[x] == 1) and (B[y] == 1):
            valid += 1
            if (abs(x-y) == 3):
                minor += 1
            if (abs(x-y) == 4):
                major += 1
            if (abs(x-y) == 7):
                fifth += 1
    w = (3*minor + 3*major + fifth) / 7
    return 1.0 - np.array([valid,w])/len(notes)
```

Note the call to `ModeNotes` uses only the `B` vector, the list of notes in the mode for the current root. We then loop over the actual notes of the melody and calculate the corresponding index into `B` for the current note (`x`) and the following note (`y`).

If the notes are part of the mode, `valid` is incremented. If the interval is 3, it's a `minor` interval. If 4, it's a `major` interval. Finally, if a fifth, we count it as well. We combine the three counts, `minor`, `major`, and `fifth` using a weighted sum (`w`) to make thirds count three times as much as fifths.

There are two return values. The first is the fraction of melody notes that are valid mode notes. The second is the fraction of notes that are thirds or fifths. As we want to minimize, the return value is one minus these values.

Our extensive objective function is now complete. The rest of `melody_maker.py` follows our usual structure: parse the command line; set up the bounds, objective function, and initializer; create the swarm object; and call `Optimize` to do the search.

When done, search results are stored in the output directory along with a MIDI file of the final melody and the score. Let's put `melody_maker.py` to work and see what evolves.

11.4.2 The Experiments

We'll run `melody_maker.py` via a script so we can generate sample melodies in all modes with all algorithms. The file `experiments_maker.py` does just that,

```
n = 0
for alg in ["RO","DE","PSO","GWO","JAYA","GA"]:
    for mode in ["ionian","dorian","phrygian","lydian","mixolydian",
                 "aeolian","locrian"]:
        cmd = "python3 melody_maker.py 20 results/maker/%s/%s 20
                 800000 %s RI %s" % (alg,mode,alg,mode)
        os.system(cmd)
        print(cmd)
        n += 2
print("%d melodies generated (time = %0.3f seconds)" % (n, time.time()-st))
```

A double loop over algorithms and modes creates what we need. Notice, we run for 800,000 iterations. If you are impatient, a smaller number can be used. I used 800,000 because I wanted to see what a well-searched space would lead to and to see if the swarm collapsed in the end or maintained some level of diversity.

Start `experiments_maker.py` and then go do something else. I recommend a weekend getaway. When done, you can listen to the results with something like

```
> wildmidi results/maker/DE/lydian/melody_DE.mid
```

My run produced melodies that fit the desired musical mode and sounded nice in terms of intervals and note durations. Naturally, our objective function is only covering the basics, it doesn't know about repetition or sudden jumps that sound nice, etc., so we should not set our expectations too high.

Unexpectedly, the different swarm algorithms led to melodies that played for more or less time on average, regardless of the mode. For instance, the DE melodies averaged about 9 seconds long, while the PSO melodies were 8 seconds. GA averaged 9 seconds and GWO was the shortest, averaging only 6 to 7 seconds. RO averaged 11 seconds, and Jaya 13 to 14 seconds, twice the length of GWO. Yet, all melodies were only twenty *notes* long.

In terms of adjectives applicable to the results, DE and PSO were steady and pleasant sounding, according to mode. Jaya had interesting, sometimes erratic, and definitely slower melodies. GWO melodies were fast and wandering. GA produced airy, ranging, pleasant, and flowing melodies. Finally, RO melodies were erratic and jumpy.

Subjective evaluation of the melodies is perhaps the best thing for us since we are exploring how swarm optimization might help in the composition of new music. Still, we are optimizing an objective function, so examination of swarm convergence is fair game.

Figure 11.4 shows the swarm best as a function of iteration for runs producing Ionian (left) and Aeolian (right) mode melodies. The results are consistent across modes and follow what we've seen in earlier chapters. RO learns slowly, if at all. DE and PSO converge quickly, with Jaya and GWO converging less rapidly. GA ends up in much the same place between the two modes and does not match Jaya or GWO, let alone PSO and DE, in terms of achieving a low objective function value.

We've focused on the swarm best, but, especially in a subjective setting like music, we should expand our appreciation and listen to the evolution of the swarm best and the final swarm itself.

To listen to the evolution of the melody, we might use a sequence of instructions like

```
>>> import numpy as np; import pickle; from melody_maker import *
>>> p = pickle.load(open("results/maker/DE/dorian/melody_DE.pkl","rb"))
>>> len(p["gpos"])
49
>>> PlayMelody(p["gpos"][0])
>>> p["gbest"][0]
3.366777528150162
```

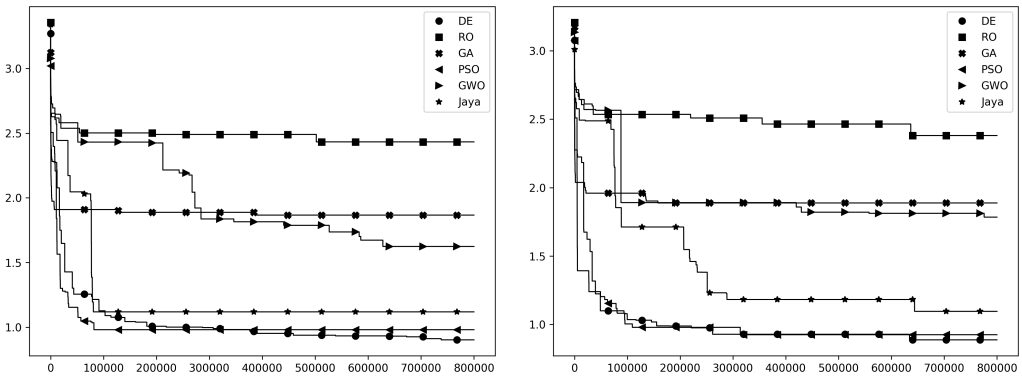


Figure 11.4: Convergence of the swarms by algorithm type for Ionian mode (left) and Aeolian mode (right).

```
>>> PlayMelody(p["gpos"][25])
>>> p["gbest"][25]
2.238948531956374
>>> PlayMelody(p["gpos"][-1])
>>> p["gbest"][-1]
0.9236193905898076
```

to load the search results and play different versions of the swarm best melody from the first (index 0) to the last (index -1). The objective function values decrease from 3.37 to 0.92, and the melodies improve accordingly.

Continuing with the session immediately above, we can also play the swarm as it existed when the search concluded,

```
>>> PlayMelody(p["pos"][0])
>>> PlayMelody(p["pos"][1])
>>> PlayMelody(p["pos"][2])
>>> p["vpos"][0], p["vpos"][1], p["vpos"][2]
(1.0328695638122378, 0.9593336763040933, 1.0042981352408091)
```

As well as show the objective function value for each swarm particle position played.

The interactive Python session above raises a question: what did the swarm do for so many iterations? Did it continue to explore, or did it collapse onto itself? The `analysis_maker.py` script examines the generated melodies and produces several outputs. Two are the plots shown in Figure 11.4. In addition, the script dumps a measure of swarm diversity for each algorithm and mode. Here, we define swarm diversity as the mean of the standard deviation of each element of a particle position across the swarm,

```
>>> pos = ... 20x40 NumPy array ...
>>> s = pos.std(ddof=1, axis=0)
>>> diversity = s.mean()
```

for `pos` a set of swarm positions, say 20 particles and 40 dimensions, `s` a vector of 40 elements, and `diversity` the mean of `s`. The higher the mean of `s` is, the more diverse the swarm. By algorithm we get,

RO	6.4985
DE	2.8550
Jaya	2.4262
GA	0.2321
GWO	0.2111
PSO	0.0580

meaning RO maintains the greatest swarm diversity, even after 800,000 iterations, while three of the algorithms show little remaining diversity. PSO, in particular, has virtually none left.

Diversity isn't enough, however. RO has high diversity, but the melodies it produces are not particularly pleasing. DE and Jaya, on the other hand, made reasonable melodies, so high diversity among the particles of the swarm is a benefit on top of the subjective quality of the results. Note, diversity is measured across all modes; see `analysis_maker.py` for implementation details.

If we scale the per element standard deviations by the mean value of each element across the swarm, we can plot the resulting vector to see if each element of a particle position is equally diverse or if some elements have collapsed across the swarm. The result is Figure 11.5. Note, the code to generate this plot is also in `analysis_maker.py`.

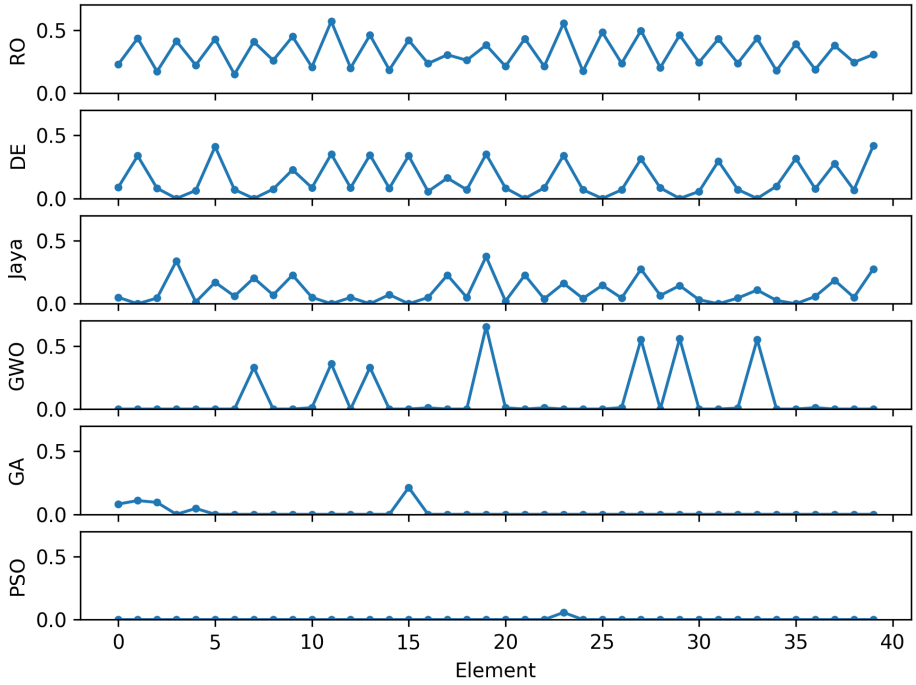


Figure 11.5: Per element swarm diversity for the Ionian mode, by algorithm.

Figure 11.5 mirrors the table above, showing the diversity across the swarms for Ionian mode. RO has the highest diversity, evenly split across the elements of the swarm positions. Recall, swarm positions are pairs of MIDI note number $[35, 85]$ and durations $[1, 5)$. The oscillation is a consequence of scaling by the mean value of each position element.

DE and Jaya also maintain diverse swarms, though less so than RO. The effect on GWO, and especially GA and PSO, is dramatic. GWO has virtually zero diversity across most elements of the position vectors. GA has even less, and PSO is, practically speaking, completely homogeneous. If we play the final swarm melodies of PSO, we immediately hear that they are almost entirely the same.

This chapter explored music as generated by swarm algorithms. We demonstrated that swarms could learn simple melodies and create melodies similar to those of a collection of related melodies, though with some surprising results. Finally, we generated melodies from scratch, melodies that fit

the desired musical mode, and produced pleasant, if uninspiring, results. As Paul Kneusel noted, the Bach-like melodies sounded like a beginning student making small rhythm errors.

The experiments gave us a new way to appreciate our swarms. Instead of graphs and tables, in this chapter, we *hear* the swarm evolve. The adjectives we used to describe the generated melodies are loosely applicable to the behavior of the swarm algorithms themselves. DE and PSO are "steady" in that they are consistent performers. Jaya is sometimes "erratic" and often "slow" to converge. GWO does sometimes "wander" – it generally performs well but can sometimes go wide of the mark. GA is "airy" and "ranging", a consequence of mixing the elements of the particle positions. Finally, RO is definitely "erratic", like GWO sometimes is; it's unclear whether it will perform well or not. Of course, all such comparisons break down at some point, but on the whole, the adjectives describing the melodies the algorithms generate fit the algorithms themselves.

It is entirely conceivable more could be done to enhance the `melody_maker.py` objective function. The current implementation pays attention to mode, intervals, and range, but does not include any attempt to insert some other, more subjective, rule of thumb related to what makes a "good" melody. Perhaps something encouraging repetition of a short theme? Enhancements are left as an exercise for the reader. During my experiments, I ran across a few melodies I liked, so I saved them for possible future use.

Let's continue our exploration of the wide range of tasks amenable to swarm optimization by leaving music behind and focusing instead on the placement of assets.

Chapter 12

Cell Towers and Circles

Many engineering and logistics problems boil down to placing assets to maximize the coverage of some area. The assets might be police stations, fire departments, camera traps for conservation efforts, or cell phone towers.

In this chapter, we'll run experiments to place simulated cell towers on a map to maximize the area covered while avoiding masked areas where no towers are allowed. The individual cell towers may cover different sized areas. The masked regions represent roads or parking lots, etc.

The idea of placing things an ideal distance apart can be generalized. For example, maximizing the minimal distance between a set of points on the unit square corresponds to packing a set of uniform circles in a square. The basic idea is similar to the cell phone tower example. Therefore, we'll also experiment with placing points on the unit square to see if swarms can find the known ideal arrangements, or at least get close.

12.1 Cell Towers

Per our usual process, we'll set up the simulation (Section 12.1.1), detail the code (Section 12.1.2), and run various scenarios to see how we do all in an effort to increase our intuition about swarm algorithms and their behavior (Section 12.1.3).

12.1.1 The Setup

Our world is a black and white map. We'll use a grayscale image with 0 intensity for allowed regions and maximum intensity (255) to mark banned areas. Some example maps are included, relatively small to make our many searches run faster, but the code works with maps of any size.

Next, we need to simulate a cell tower. From the map's perspective, the cell tower is a disc with a center, (x, y), and a radius, r. The radius is fixed; it represents the strength of the tower, the area it can cover. The location, of course, is what we are hoping the swarms will find for us. We'll list towers in a text file with each line a floating-point number, $[0, 1]$, interpreted as the tower's coverage region as a fraction of some maximum allowed radius.

Therefore, our goal, for a given map, is to find a set of (x, y) pairs placing the N cell towers to maximally cover the map while avoiding masked regions.

The search bounds are fairly obvious: we're in pixel space, so the (x, y) locations are bounded to $[0, M]$ where M is the length of one of the map's sides, assuming a square map. Additionally, to save on extraneous swarm updates, we'll discretize the positions to correspond to actual pixels in the map image.

What are we optimizing? For a particular arrangement of towers, we can, literally, set map pixel values to something other than zero, and when all towers are set, count the number of zero pixels remaining. This number divided by the total number of pixels gives the fraction of the map not covered by a tower. It's this value we'll minimize.

12.1.2 The Code

The simulation code is in `cell.py`. We load the usual modules at the top. We then need to define custom Bounds and Objective classes. The Bounds class is

```
class CellBounds(Bounds):
    def __init__(self, lower, upper, enforce):
        super().__init__(lower, upper, enforce)
    def Validate(self, p):
        return np.floor(p)
```

`CellBounds` need only implement `Validate` to discretize a set of tower positions. The argument, `p`, is a $2N$-element vector of (x, y) pairs for N towers. Discretization is straightforward: the floor operation ensures the elements are integer and in range.

The `Objective` class is more interesting as it needs to accomplish two goals. First, it needs to check if any of the proposed tower centers are in the map's masked areas. If they are, the tower configuration is invalid, and a suitable objective function value is returned. The second goal is creation of a coverage map. This map is initially empty and then filled in, tower by tower, by setting pixels in the area covered by the tower to a nonzero value. As stated above, the fraction of zero pixels remaining gives us a measure of how well the area is covered. For this task, we know what perfection is– there are no zero pixels, though such a condition might be impossible if the sum of the area covered by all the towers is less than the area of the map.

Let's walk through the `Objective` class code:

```
class Objective:
    def __init__(self, image, towers, radius):
        self.image = image.copy()
        self.R, self.C = image.shape
        self.radii = (towers*radius).astype("int32")
        self.fcount = 0
    def Collisions(self, xy):
        n = 0
        for i in range(xy.shape[0]):
            x,y = xy[i]
            if (self.image[x,y] != 0):
                n += 1
        return n
    def Evaluate(self, p):
        self.fcount += 1
        n = p.shape[0]//2
        xy = p.astype("uint32").reshape((n,2))
        if (self.Collisions(xy) != 0):
            return 1.0
        empty = np.zeros((self.R, self.C))
        cover = CoverageMap(empty, xy, self.radii)
        zeros = len(np.where(cover == 0)[0])
        uncovered = zeros / (self.R*self.C)
        return uncovered
```

The `Objective` class implements three methods. Two are familiar: the constructor and `Evaluate`, which is called by our framework. The third, `Collisions`, checks to see if the proposed tower centers are allowed. If even one tower is centered in a masked region, the entire configuration is rejected.

The constructor accepts the map (`image`), a 2D NumPy array scaled $[0, 1]$, a vector of tower radii as fractions of some maximum allowed radius (`towers`), and the maximum radius (`radius`). We'll clarify what we mean by "maximum allowed radius."

Consider the `Evaluate` method. The $2N$-dimensional particle position (`p`) is converted to `xy`,

a NumPy array of N rows, one for each tower, and two columns, the x and y coordinates of the candidate tower center.

If `Collisions` returns nonzero, at least one tower center was in a map region that was not zero. This invalidates the tower arrangement proposed by `p`, so we return 1.0 to indicate that none of the map is covered.

Now that we know the arrangement of towers is allowed, we set up an empty map (`empty`) the same size as the given map. The function `CoverageMap` accepts a map, a set of tower centers, and the corresponding vector of tower radii. It returns a new 2D array where each pixel covered by a tower is set to a nonzero value. Therefore, the number of zero elements in the `cover` array represents locations not covered by at least one tower. `Evaluate` returns this number divided by the total number of elements (pixels) in the map.

Let's look in detail at `CoverageMap` as it requires a bit of explanation. First, the code,

```
def CoverageMap(image, xy, radii):
    im = image.copy()
    R,C = im.shape
    for k in range(len(radii)):
        x,y = xy[k]
        for i in range(x-radii[k],x+radii[k]):
            for j in range(y-radii[k],y+radii[k]):
                if ((i-x)**2 + (j-y)**2) <= (radii[k]*radii[k]):
                    if i < 0 or j < 0:
                        continue
                    if i >= R or j >= C:
                        continue
                    im[i,j] += 0.5*(k+1)/len(radii)
    imax = im.max()
    for k in range(len(radii)):
        x,y = xy[k]
        im[x,y] = 1.4*imax
    return im
```

`CoverageMap` accepts an `image`, either an empty array the size of the map or the map itself when generating the final output, `xy`, the set of tower locations, and `radii`, the radius of each tower, the area it covers. The return value is a new array representing the area covered by the towers.

For example, Figure 12.1 presents a map on the left and the same map on the right with the best set of tower locations found by GA after 400 iterations of a swarm with 20 particles. The map on the left marks no-tower regions that might represent a road and some buildings. The image on the right is the output of `CoverageMap` when given the map on the left as input.

`CoverageMap` creates a copy of the input `image` and keeps the size, `R` and `C`, handy. It then loops over each tower to get the center location (`x`, `y`) and the radius (`radii[k]`).

To fill in the points inside the disc representing the tower, we loop over a square region with side length equal to the diameter asking if each point in the square is also within the disc,

$$(i-x)^2 + (j-y)^2 \leq r^2$$

for current pixel (i, j), tower center at pixel (x, y), and radius, r. If the point is inside the disc, we set it to a nonzero value,

$$\text{im}_{ij} \leftarrow \text{im}_{ij} + \frac{k+1}{2N}$$

with k the index of the current tower (zero-based) and N the number of towers. Adding in a per tower intensity marks each tower differently while also showing us how they are overlapping.

Finally, `CoverageMap` loops over the tower centers one last time to set the actual center points to be the brightest pixels in the output. Doing this marks the tower centers so we can verify they

Figure 12.1: (Left) A map marking roads and buildings. (Right) A set of towers covering the map.

are not inside a masked region. To convert `im` to an actual image file, we divide by the maximum value and multiply by 255.

The core of the cell tower application lies in `CoverageMap`. It is the most task-specific portion of the code. The remainder of `cell.py` drives the search based on user-supplied arguments.

To run `cell.py`, use

```
> python3 cell.py <map> <towers> <npart> <niter> <alg> RI|QI|SI <outdir> [frames
    ]
```

where `<map>` is a map image, see the `maps` directory. The next three arguments, `<npart>`, `<niter>`, and `<alg>`, supply the number of particles, iterations, and algorithm type as we've used before. Likewise, the next two arguments are the initialization type and the name of the output directory.

The final, optional argument is the literal word "frames". In reality, any argument value here will do. If the argument is present, the output directory will contain a `frames` directory showing the cell tower placement for each iteration of the swarm, the right side of Figure 12.1. You can use this set of images to visually track the swarm's evolution in terms of the current best tower arrangement.

The `main` function parses the command line. In particular, we load and process the input map,

```
map_image = 0.9*np.array(Image.open(sys.argv[1]).convert("L"))/255
```

converting to grayscale and scaling by 255. The additional factor of 0.9 sets the maximum intensity of the masked regions. This is done to make the center points of the towers more prominent in the output coverage images.

Next, we load the text file of tower radii, one per line, $[0, 1]$,

```
towers = np.array([float(i[:-1]) for i in open(sys.argv[2])])
```

We'll calculate a maximum radius later and use it to set the per tower radii.

Next, we create an instance of `CellBounds` and set it to the limits of the input map,

```
x,y = map_image.shape
lower = [0,0]*len(towers)
upper = [x,y]*len(towers)
b = CellBounds(lower, upper, enforce="resample")
```

```
ndim = 2*len(towers)
```

The maximum allowed radius, in pixels, is next. We chose to limit any tower to half the largest dimensions of the input map,

```
w = x if (x>y) else y
radius = w//2
```

From here, we create the desired initializer and swarm objects. We've seen these statements before. The only new option here is adding `tol=1e-9` to the constructors since once we've covered all pixels in the map, if possible with the given set of towers, the search is complete.

The objective function is created as well,

```
obj = Objective(map_image, towers, radius)
```

passing in the scaled map, the set of towers, and the maximum radius.

Everything is now in place. We could call `Optimize` and then look at the search results. Instead, we'll iterate step-by-step to track the evolution of the swarm,

```
k = 0
swarm.Initialize()
while (not swarm.Done()):
    swarm.Step()
    res = swarm.Results()
    t = "    %5d: gbest = %0.8f" % (k,res["gbest"][-1])
    print(t, flush=True)
    s += t+"\n"
    k += 1
    if (frames):
        p = res["gpos"][-1]
        n = p.shape[0]//2
        xy = p.astype("uint32").reshape((n,2))
        radii = (towers*radius).astype("int32")
        cover = CoverageMap(map_image, xy, radii)
        img = Image.fromarray((255*cover/cover.max()).astype("uint8"))
        img.save(outdir+"/frames/"+("frame_%05d.png" % k))
```

The calls to `Initialize`, `Done`, `Step`, and `Results` we've seen before. While not done with iterations or perfection, perform a step of the swarm, get the current best arrangement of cell towers, and output the fraction of the map still not covered. Everything printed at the console is stored in `s` to be dumped in the output directory.

If frames are output, a call to `CoverageMap` using the current best tower arrangement and the actual map showing masked regions gives us the desired image. Notice the scaling of the image to $[0, 255]$ for output in the `frames` directory. Regardless of whether frames are output or not, at the end of the search, we dump the final tower configuration image to the output directory as `coverage.png`. The final output image is passed through a square root operation to enhance the contrast of the tower regions.

12.1.3 The Experiments

With everything in place, we are now ready to run the simulation to see which algorithms perform best on this task.

Let's first define a collection of towers. We'll use four different sets to which we give the uncreative names of `towers`, `towers1`, `towers2`, and `towers3`.

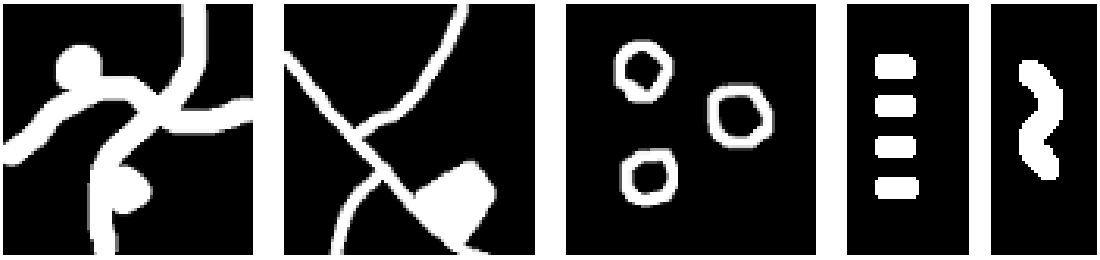
Table 12.2 details the tower sets, including the maximum possible area they could cover if they were not to overlap. Note, we do not enforce any limitation on the overlap. Two towers can be adjacent to each other. Indeed, the simulation code allows towers to be placed on top of each other.

Filename	*Number of Towers*	*Maximum Area*	*Fraction*
`towers`	6	0.715	$0.1, \ldots, 0.6$
`towers1`	15	1.186	0.3 (15x)
`towers2`	30	2.121	0.3 (30x)
`towers3`	6	1.100	0.1,0.6,1.0

Table 12.1: The tower sets used in the experiments.

The number of towers ranges from a low of six to a maximum of 30 with radii fractions from 0.1 to 1.0. Note, `towers` and `towers3` imply a maximum coverage area of less than 1.0, meaning the strength and number of towers are insufficient to cover a map completely.

Our goal is the placement of towers on a map, so we need some maps. We'll start with the empty map, one with no masked regions. Additionally, we'll define four more maps, those of Figure 12.2. We intend to run many experiments, so the maps are rather small, 80x80 pixels, but that's still large enough for our purposes. The map in Figure 12.1 is larger, 512x512 pixels.



`map_01` `map_02` `map_03` `map_04`

Figure 12.2: The maps used for the experiments.

Maps 1 and 2 represent roads and parking lots. Map 4 adds in a few buildings and a pond. Map 3 is a flight of fancy. We'll use all of the maps in the experiments below. Recall, if making your own maps, make available background pixels intensity zero and masked regions intensity 255. The larger the map, the longer the code takes to run. As always, we choose clarity in the code over performance. I expect any system based on the ideas in this book will employ proper software engineering practices.

A single run of `cell.py`,

```
> python3 cell.py maps/map_04.png towers3 20 300 PSO RI example frames
```

using `map_04` and `towers3` produced output in the `example` directory. The `frames` argument created per iteration coverage maps as well.

The `example` directory contains the following files:

```
coverage.png
frames
map.png
README.txt
results.pkl
```

The first is the final cover image showing the towers superimposed on the map. Next is the directory of frames. The original map image is in `map.png` and the dictionary returned by `swarm.Results()` is in `results.pkl` to allow for detailed analysis of the search, if desired.

All text dumped to the console is captured in `README.txt`. The tail end of this file, for this particular search, is,

```
      297: gbest = 0.13140625
      298: gbest = 0.13140625
      299: gbest = 0.13140625

Search results: PSO, 20 particles, 300 iterations, RI

Optimization minimum 0.13140625 (time = 205.279)
(20 best updates, 6020 function evaluations)
```

showing the current swarm best coverage for the final three iterations of the swarm and some general statistics on the search itself. This search ended with just over 13% of the map not covered by a tower. From Table 12.2, we see that `towers3` has a maximum total area of 1.1, but this does not mean it is possible to completely cover the map since the towers represent discs, not arbitrary shapes.

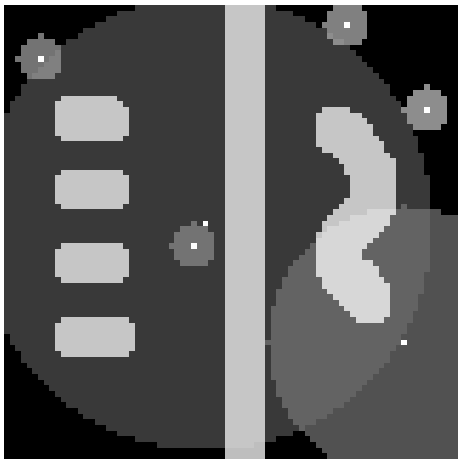


Figure 12.3: The final tower arrangement for the example PSO search.

Figure 12.3 shows the final arrangement of towers. The image has been enhanced to show the tower bounds more clearly on the page. The one large tower is near the center and likely would have been moved to the center if the road did not block it. Notice that none of the six tower center locations are over a masked region of the image. Also, notice that the swarm did not optimize as well as it might. The large tower completely overshadows one of the small towers. The swarm could have moved this tower to the lower left or upper right corner to cover more of the map.

Let's write a script to generate multiple runs of each algorithm for each map and two tower sets: `towers` and `towers1`. The script is in the file `experiments.py`,

```
os.system("rm -rf results; mkdir results")
n = 0
for alg in ["RO","DE","PSO","JAYA","GWO","GA"]:
    for tower in ["towers","towers1"]:
        for m in range(5):
            for r in range(8):
                cmd = "python3 cell.py maps/map_%02d.png %s
                       20 300 %s RI results/%s_map_%02d_%s_run%d" %  \
                       (m, tower, alg, alg.lower(), m, tower, r)
                os.system(cmd)
                n += 1
```

The script loops over algorithms, tower sets, maps (`m`), and runs (eight each) to fill the `results` directory with 480 outputs. The script does take some time to run.

When `experiments.py` finishes running, we can interpret the output with `analysis.py`. The script loops over the output directories,

```
def Results(alg):
    res = np.zeros((2,5,8))
    k = 0
    for tower in ["towers","towers1"]:
        for m in range(5):
            for r in range(8):
                fname = "results/%s_map_%02d_%s_run%d/README.txt" %
                        (alg.lower(), m, tower, r)
                lines = [i[:-1] for i in open(fname)]
                f = float(lines[-3].split()[-4])
                res[k,m,r] = f
        k += 1
    return res
```

for each algorithm (`alg`). The return value is a NumPy array with the final fraction of the map not covered for each of the eight trials. It's the mean over trials, maps, and tower sets we are after.

So, how did the algorithms do? The output of `analysis.py` is shown in condensed form in Table 12.2. Let's start with `map_0`, the one with no masked regions. The total area of `towers` is 0.715, and it is possible to arrange the towers so they do not overlap at all. Therefore, the best we can hope to do in that case is $1 - 0.715 = 0.285$. None of the algorithms meet this minimum value, though PSO gets close with one run producing a result of 0.297. If we stick with the `towers` column and look down across the maps, we see, overall, there is no clear winner or loser in terms of algorithms; all do approximately equally well.

Now consider the `towers1` column of Table 12.2. For `map_0` we see that the 15 identical towers are, as we might expect, better able to cover the region than the six larger towers on the left. Here the winner is GA with a run producing a minimum of 0.093; only 9.3% of the area was not covered by a tower.

Now, look at `map_1`, especially the upper limits, meaning the worse case outputs. Four of the algorithms, RO, PSO, GWO, and GA, all had at least one run that failed to find an allowed arrangement of towers. Recall, the objective function returns 1.0, no area covered, when a tower center lands on a masked region. This failure also happened for GA and `map_2`.

Figure 12.4 shows the final uncovered area per algorithm and run for `map_1` and the `towers1` set of towers. There were eight runs for each algorithm. The plot offsets the runs slightly to prevent the symbols from overlapping completely.

As noted, several algorithms failed, but not just once, several times each. When the algorithms did not fail, they all performed about the same with one exception: GA. For GA, either the search failed, or it produced the best coverage of all. A glance back at the `towers1` minimums of Table 12.2 confirms GA as the best algorithm of all regardless of the map. It seems that GA either performs well or fails spectacularly.

Figure 12.5 presents the final coverage for the `towers1` set and GA. These are the best configurations found; the minimum noted in Table 12.2. We see that GA, and to a lesser degree the other algorithms, can find reasonably good solutions while respecting the constraints. However, as we've also seen previously in this book, the swarm algorithms' stochastic nature requires vigilance and, sometimes, persistence to achieve the goal.

Let's move to a new set of experiments for a related problem, that of packing circles in a square.

12.2 Packing Circles

Section 12.1 solved a practical problem, albeit in a simplified manner. In this section, we attempt to solve the related problem of packing circles in a square. In reality, we'll work with an equivalent formulation, that of placing points in the unit square to maximize the minimal distance between them. These points become the centers of identical circles packing a square as tightly as possible.

	Towers	Towers1
Map 0:		
RO	0.333 +/- 0.003 (0.319, 0.341)	0.139 +/- 0.005 (0.119, 0.167)
DE	0.319 +/- 0.002 (0.312, 0.329)	0.183 +/- 0.003 (0.171, 0.192)
PSO	0.302 +/- 0.002 (0.297, 0.310)	0.192 +/- 0.006 (0.161, 0.211)
GWO	0.314 +/- 0.006 (0.301, 0.351)	0.147 +/- 0.017 (0.094, 0.210)
JAYA	0.321 +/- 0.004 (0.309, 0.338)	0.175 +/- 0.002 (0.165, 0.185)
GA	0.323 +/- 0.004 (0.311, 0.346)	0.115 +/- 0.004 (0.093, 0.131)
Map 1:		
RO	0.410 +/- 0.011 (0.349, 0.451)	0.570 +/- 0.126 (0.284, 1.000)
DE	0.339 +/- 0.004 (0.324, 0.354)	0.258 +/- 0.010 (0.214, 0.293)
PSO	0.321 +/- 0.005 (0.298, 0.339)	0.386 +/- 0.088 (0.278, 1.000)
GWO	0.324 +/- 0.010 (0.305, 0.392)	0.577 +/- 0.124 (0.279, 1.000)
JAYA	0.369 +/- 0.005 (0.349, 0.384)	0.283 +/- 0.007 (0.260, 0.306)
GA	0.339 +/- 0.005 (0.318, 0.358)	0.574 +/- 0.161 (0.122, 1.000)
Map 2:		
RO	0.380 +/- 0.010 (0.350, 0.440)	0.261 +/- 0.023 (0.185, 0.394)
DE	0.338 +/- 0.004 (0.321, 0.354)	0.212 +/- 0.006 (0.186, 0.233)
PSO	0.316 +/- 0.005 (0.304, 0.344)	0.245 +/- 0.005 (0.228, 0.270)
GWO	0.322 +/- 0.007 (0.308, 0.371)	0.252 +/- 0.018 (0.162, 0.308)
JAYA	0.340 +/- 0.007 (0.313, 0.366)	0.242 +/- 0.006 (0.217, 0.271)
GA	0.340 +/- 0.009 (0.306, 0.393)	0.251 +/- 0.107 (0.099, 1.000)
Map 3:		
RO	0.365 +/- 0.006 (0.337, 0.386)	0.177 +/- 0.010 (0.127, 0.205)
DE	0.331 +/- 0.002 (0.323, 0.343)	0.199 +/- 0.003 (0.187, 0.207)
PSO	0.310 +/- 0.002 (0.302, 0.322)	0.228 +/- 0.005 (0.206, 0.243)
GWO	0.316 +/- 0.003 (0.308, 0.330)	0.210 +/- 0.017 (0.149, 0.259)
JAYA	0.343 +/- 0.003 (0.332, 0.358)	0.210 +/- 0.007 (0.186, 0.238)
GA	0.333 +/- 0.004 (0.312, 0.348)	0.133 +/- 0.005 (0.100, 0.146)
Map 4:		
RO	0.367 +/- 0.009 (0.323, 0.407)	0.257 +/- 0.017 (0.197, 0.333)
DE	0.335 +/- 0.002 (0.328, 0.347)	0.208 +/- 0.004 (0.185, 0.225)
PSO	0.310 +/- 0.003 (0.297, 0.323)	0.235 +/- 0.008 (0.186, 0.260)
GWO	0.327 +/- 0.009 (0.300, 0.375)	0.248 +/- 0.013 (0.182, 0.291)
JAYA	0.343 +/- 0.004 (0.325, 0.363)	0.219 +/- 0.006 (0.198, 0.245)
GA	0.330 +/- 0.007 (0.303, 0.366)	0.136 +/- 0.007 (0.110, 0.164)

Table 12.2: Results for the towers and towers1 sets showing mean ± SE along with minimum and maximum uncovered area.

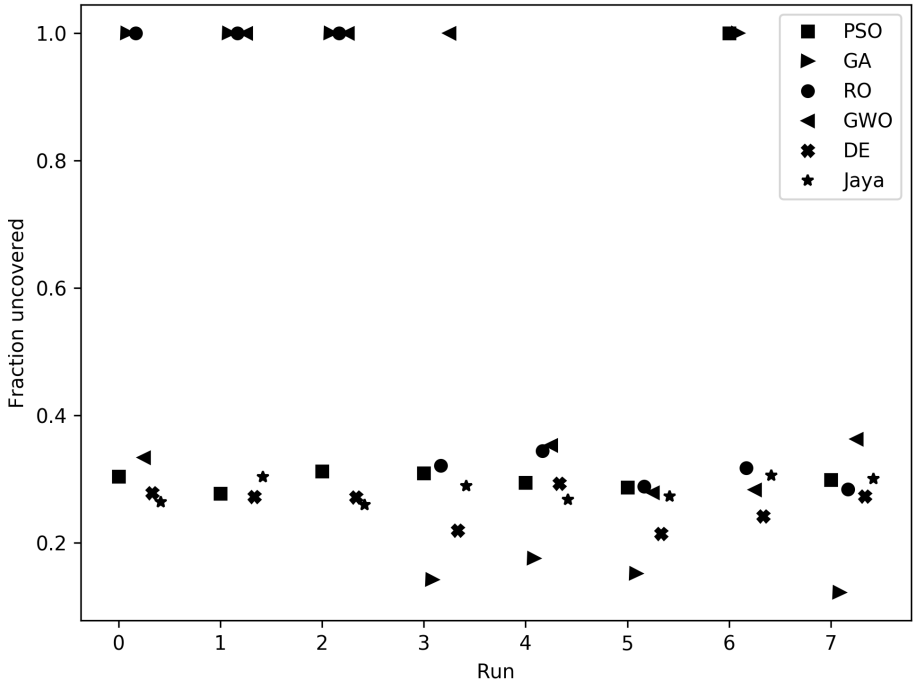


Figure 12.4: Per run uncovered fraction for map 1 and towers1.

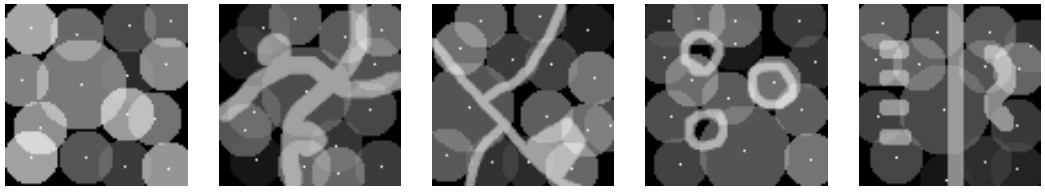


Figure 12.5: Best GA run for each map and towers1 (the minimums of Table 12.2).

Deterministic algorithms already exist; see [35]. The problem is known to be NP-hard. Our solution, of course, will be swarm-based. This is another example of the wide-applicability of swarm techniques and, in a practical sense, might be good enough.

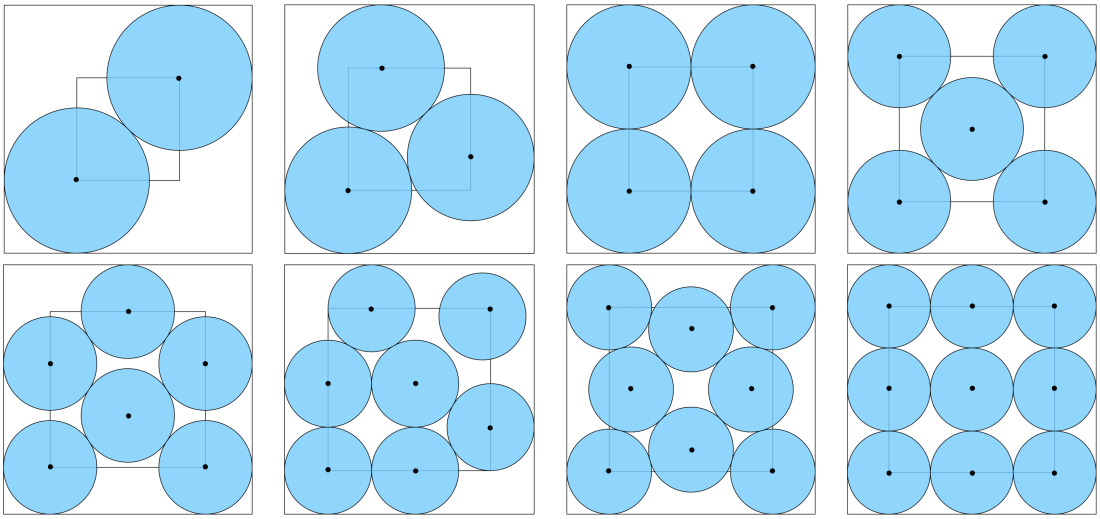


Figure 12.6: Maximal minimal separation of points on the unit square for 2 through 9 points. Circles represent the maximum packing arrangement centered on each point. (By Parcly Taxel - Own work, FAL, https://commons.wikimedia.org/w/index.php?curid=67465446)

Figure 12.6 shows what we are seeking, the best packing of circles in a square. What we're after are the black dots, those on or within the inner square, which is the unit square. The exact distance is known for two through nine points, see [36],

Points	*Distance*
2	$\sqrt{2} = 1.414214$
3	$\sqrt{6} - \sqrt{2} = 1.035276$
4	1
5	$\sqrt{2}/2 = 0.707107$
6	$\sqrt{13}/6 = 0.600925$
7	$2(2 - \sqrt{3}) = 0.535898$
8	$(\sqrt{6} - \sqrt{2})/2 = 0.517638$
9	0.5

We'll present our results in graphical form as deviations from the known maximum separation.

12.2.1 The Code

The code is in `points.py`. We won't walk through the main portion of the code as it is nearly identical to that of `cell.py` above. We need only contemplate the objective function and the mapping between a swarm particle position and our solution, including any boundary constraints.

We want to place N points, (x, y), on the unit square such that the smallest distance between the points is maximized. The mapping becomes straightforward: each swarm particle represents a list of N (x, y) pairs. So, if we seek to place five points, we are in a 10-dimensional space,

$$\boldsymbol{p} = (x_0, y_0, x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4)$$

with x and y constrained to $[0, 1]$.

Therefore, the boundary condition becomes,

```
ndim = 2*int(sys.argv[1])
b = Bounds([0]*ndim, [1]*ndim, enforce="resample")
```

with `ndim` twice the number of points to place, the first command-line argument to `points.py`. For now, we are resampling on boundary violations.

What of the objective function? The problem statement is to maximize the *minimal* distance between the points. So, for any given set of points, we need to find the minimal separation between any pair and use that. In code we get,

```
class Objective:
    def __init__(self):
        self.fcount = 0
    def Evaluate(self, p):
        self.fcount += 1
        n = p.shape[0]//2
        xy = p.reshape((n,2))
        dmin = 10.0
        for i in range(n):
            for j in range(i,n):
                if (i==j):
                    continue
                d = np.sqrt((xy[i,0]-xy[j,0])**2 + (xy[i,1]-xy[j,1])**2)
                if (d < dmin):
                    dmin = d
        return -dmin
```

The constructor sets up `fcount` to count calls to the objective function. The `Evaluate` method reshapes the particle position (`p`) into a set of points (`xy`) running through each pair to calculate the Euclidean distance between them. As distance is symmetric, we need only look at each pair once. Whenever we find a new minimum distance between a pair, we keep it in `dmin`. When all pairs have been examined, we return the negation of `dmin` as the objective function value.

Do look at the `main` function of the `points.py` script. After the search, it dumps the outcome, including the actual point positions, in the output directory, including a plot of the points on the unit square.

12.2.2 The Experiments

For our experiments we'll attempt to place $N = 2, \ldots, 9$ points on the unit square. A simple script, `experiments_points.py`, will do the runs for us,

```
import os
os.system("rm -rf results_points; mkdir results_points")
n = 0
for alg in ["RO","DE","PSO","JAYA","GWO","GA"]:
    for pnts in [2,3,4,5,6,7,8,9]:
        for r in range(8):
            miter = pnts*6000
            cmd = "python3 points.py %d 25 %d %s RI
              results_points/%s_%d_run%d" %
              (pnts,miter,alg,alg.lower(),pnts,r)
            os.system(cmd)
            n += 1
```

We'll use a swarm of 25 particles and adjust the maximum number of iterations to make it a function of the number of points. Intuitively, we feel that placing more points might require more iterations. As previously, we'll use random initialization and leave to the reader experimentation with other

initialization schemes. We use eight runs for each configuration to give us some idea of consistency and mean performance.

Running `experiments_points.py` takes a few hours. When complete, the `results_points` directory contains many results directories. To investigate the results, run `analysis_points.py` to produce a summary and output plots, those presented in Figure 12.7.

Figure 12.7 shows the *deviation* from the ideal point separation, meaning a well-performing search appears as points at or near zero. A first glance at the figure tells us most algorithms did rather poorly, with deviation increasing as a function of the number of points, N.

A second glance at Figure 12.7 reveals a little more. For $N = 2, \ldots, 5$, DE performed very well, with one bad run for $N = 5$. Much the same may be said of GA, though only through $N = 4$, and with small deviations from ideal. For $N = 2$, GWO was reasonably close, but not in a satisfactory way and with one failure. Somewhat surprisingly, Jaya did poorly in all cases.

Overall, DE succeeded in simple cases, followed closely by GA, but none of the other algorithms managed to find the ideal locations, and some were not even close. The algorithms did group, more or less, meaning some level of precision, if not accuracy. Still, GWO, as we've seen before, had occasional fits of lousy performance outside its main cluster of results.

Why did the algorithms fail? Is the task simply too hard? At some level, we would expect the swarms to fail, certainly as N increases, especially so with the knowledge that the problem is NP-hard. However, did we do right by the algorithms in our configuration of the problem?

In a sense, no, we did not. Intuition leads us to believe many solution points lie on the edges of the unit square. Clearly, we don't need an algorithm to show us the ideal positions for the $N = 2$ and $N = 4$ cases; even the $N = 5$ case is straightforward: points on opposite edges, points on all corners, and an additional point in the center. So, we should configure our swarms to take advantage of that fact.

We set the Bounds object to resample; therefore, every time a particle moved a dimension beyond a boundary, we kicked that dimension back to a new random value. In so doing, we made it extremely difficult for the swarm to converge on a boundary position, which is precisely what we need it to do.

Therefore, let's run again, but this time we'll change one tiny piece of code. Instead of `resample`, we'll configure the Bounds object to use `clip`,

```
b = Bounds([0]*ndim, [1]*ndim, enforce="clip")
```

Now, boundary violations are clipped to zero or one.

Figure 12.8 shows the effect of the change. The swarms are much more likely to find the ideal arrangement, or at least get quite close for the given number of swarm iterations. This is true of all algorithms through $N = 4$. For $N = 5$, DE performs well on most runs.

The quality of the search decreases for $N = 6$ and even more so for $N = 7$. The $N = 8$ case is still worse, overall, but PSO gets "trapped" and produces the same wrong value for all runs, 0.5. From Figure 12.6, $N = 8$ leads to the four corners and a smaller inset diamond shape. The PSO results for $N = 8$ found all four corners, but never the inner diamond shape. One run produced four corners and the midpoints of the other four sides, which is the ideal configuration for $N = 9$ with the one point in the center at $(0.5, 0.5)$ removed. Indeed, for $N = 9$, PSO does find the ideal arrangement on all runs. Clearly, something about how PSO works internally, mixed with the clipping boundary condition, is forcing this kind of result, which is wrong in one case but correct in another.

Also of interest is Jaya's performance for $N = 9$. With one exception, Jaya finds the ideal arrangement where it was similarly poor for $N \geq 5$. Jaya looks at the best and worst the current swarm has to offer. It's likely the clipping boundary condition forces the best and worst particle positions into a configuration that finds the ideal in most cases.

For example, a single run of `points.py`, with clipping on boundary violations using this command line,

```
> python3 points.py 13 25 100000 jaya ri results
```

produced the following output,

```
Optimization minimum -0.36602540 (time = 1137.790)
(651 best updates, 3000030 function evaluations)

    (x,y) = (0.00000000, 1.00000000)
    (x,y) = (1.00000000, 1.00000000)
    (x,y) = (0.68301270, 0.81698730)
    (x,y) = (1.00000000, 0.00000000)
    (x,y) = (0.00000000, 0.36602540)
    (x,y) = (0.63397460, 0.00000000)
    (x,y) = (0.18301270, 0.68301270)
    (x,y) = (0.00000000, 0.00000000)
    (x,y) = (0.31698730, 0.18301270)
    (x,y) = (1.00000000, 0.63397460)
    (x,y) = (0.36602540, 1.00000000)
    (x,y) = (0.50000000, 0.50000000)
    (x,y) = (0.81698730, 0.31698730)
```

where according to [36], the ideal separation for 13 points is

$$(\sqrt{3} - 1)/2 = 0.3660254037844386$$

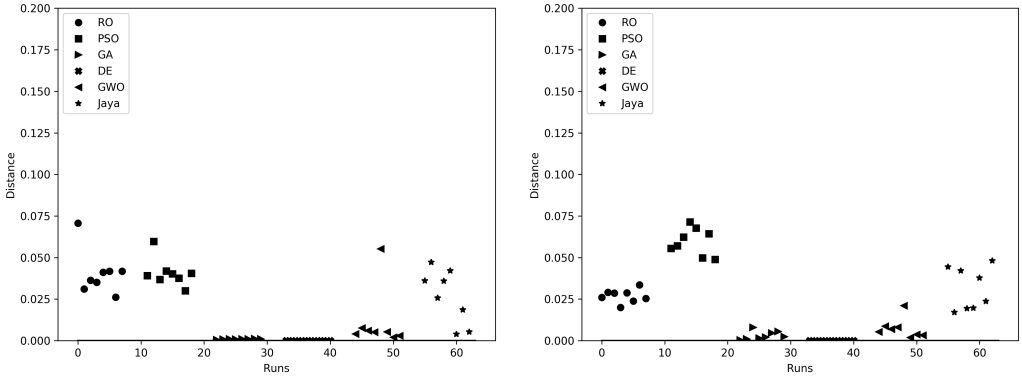
meaning Jaya found the ideal arrangement of 13 points to at least eight decimals of accuracy.

12.3 Summary

The experiments in this chapter dealt with covering an area. In the first case, Section 12.1, we placed simulated cell towers to cover a map while respecting masked regions where no towers could be placed. We were moderately successful in placing different arrangements in a way that maximized coverage, though we did fail on occasion, a well-known characteristic of swarm techniques.

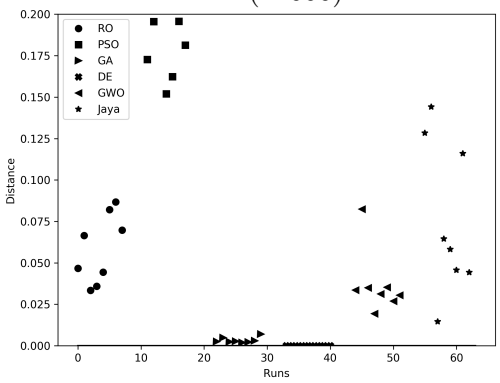
In Section 12.2, we abstracted things a bit and asked how to place points on a unit square to maximize the smallest distance between them. With relatively few swarm iterations, only tens of thousands, we achieved some excellent placements of points, but not until we configured the search correctly by clipping on boundary violations. We might view adding clipping as a form of external knowledge since we could see for basic cases that many points would lie on the edge of the search range, thereby making it essential to allow the swarms to converge at those positions.

Let's leave covering problems behind and move on to yet another task, one quite different from any we've explored previously: simulating a grocery store to learn the best ordering of products to maximize daily revenue.

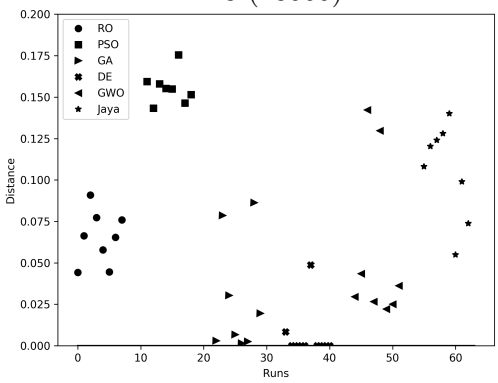


$N = 2$ (12000)

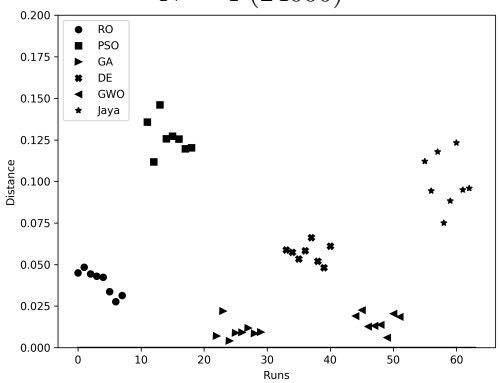
$N = 3$ (18000)



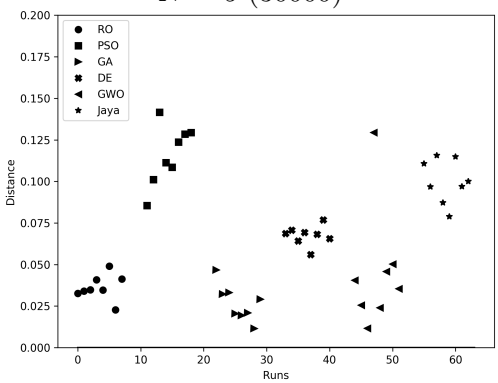
$N = 4$ (24000)



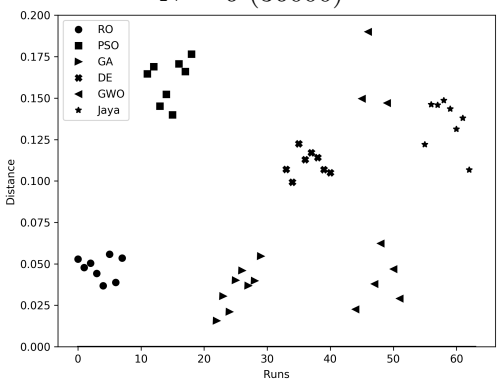
$N = 5$ (30000)



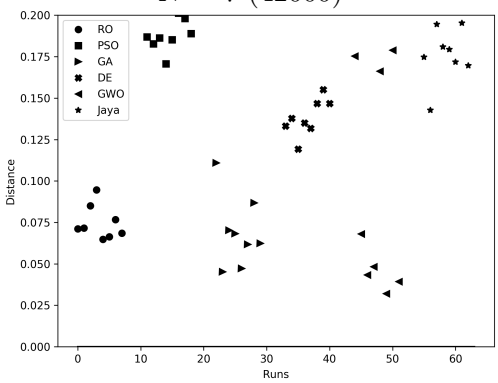
$N = 6$ (36000)



$N = 7$ (42000)

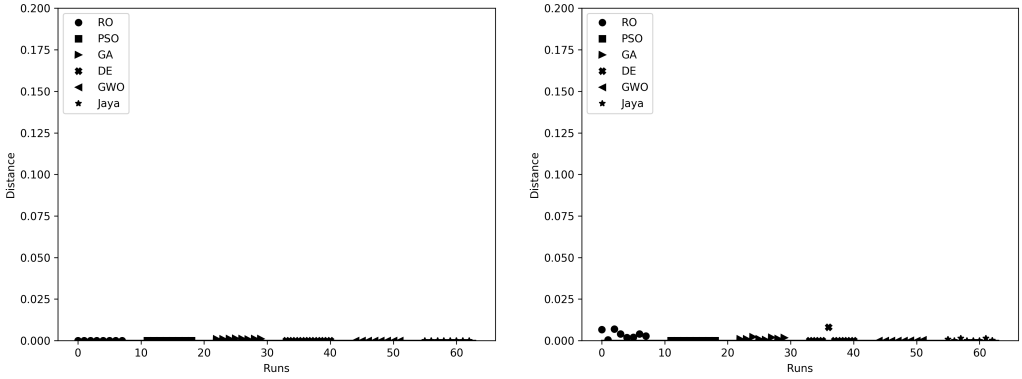


$N = 8$ (48000)

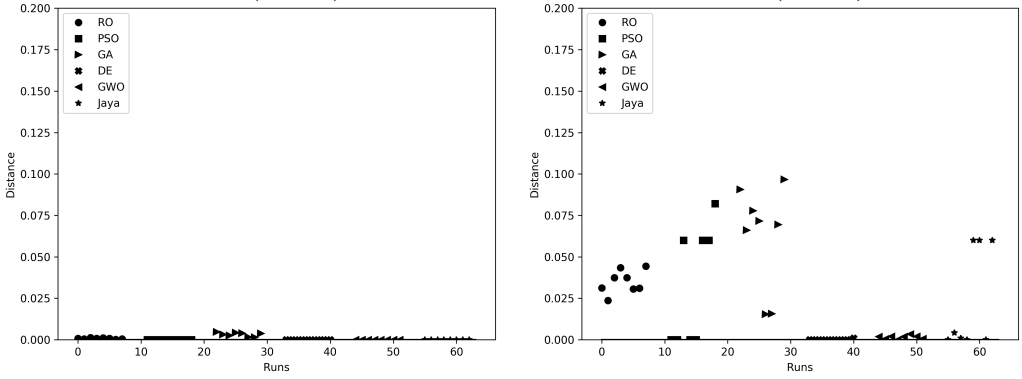


$N = 9$ (54000)

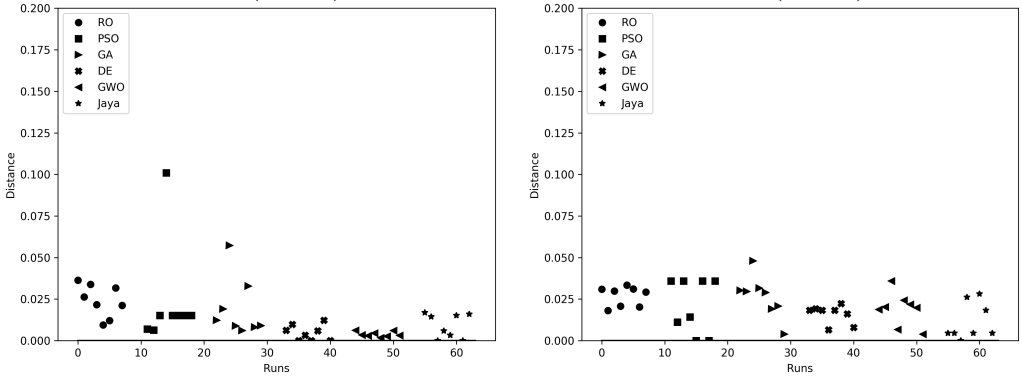
Figure 12.7: Deviation from the ideal point separation for $N = 2, \ldots, 9$ for eight runs of each algorithm using resampling on boundary violations and the given number of swarm iterations.



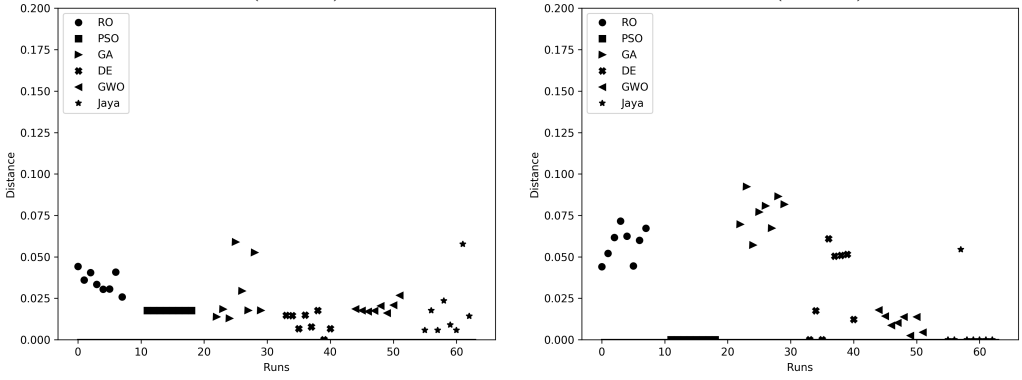
$N = 2$ (12000) $N = 3$ (18000)



$N = 4$ (24000) $N = 5$ (30000)



$N = 6$ (36000) $N = 7$ (42000)



$N = 8$ (48000) $N = 9$ (54000)

Figure 12.8: Deviation from the ideal point separation for $N = 2, \ldots, 9$ for eight runs of each algorithm using clipping on boundary violations and the give number of swarm iterations.

Chapter 13

Grocery Store Simulation

This chapter presents us with a new situation, one where we need to design, configure, and execute a simulation in an attempt to learn something useful in the real world.

Our running example is the following: we own a small grocery store and are interested in learning how to arrange the products to maximize daily revenue.

To accomplish this goal, we first need to design the simulation environment (Section 13.1), then we need to define the shopper agent (Section 13.2) and the store (Section 13.3). Finally, we simulate to see what results (Section 13.4).

13.1 The Design

A real grocery store has a physical arrangement of products in space. Shoppers enter the store at the front and walk through the store to get to the different places where products are located. For example, there might be a bakery aisle, a deli, and a produce section. Dairy products are in one location, while breakfast cereals are in another.

Therefore, our simulation needs to capture a grocery store's physical layout, at least in some abstract sense. We won't go so far as to capture an actual store's physical layout, though one might imagine doing this. Instead, we'll use an idealized physical layout. The important fact for us to realize is our need to model the placement of products in some fashion.

Grocery stores are more or less static in their arrangement of products, but the people who use the stores are not. We need to consider *how* shoppers make use of a particular sequence of products in a store. To do that, we need to model shoppers as active agents where the store, and its arrangement of products, is the environment where the agents exist and interact.

Shoppers are not all the same. A shopper might go to the store to purchase a single product, perhaps sugar for her morning coffee, but, while walking through the store, the shopper sees a piece of candy she likes and buys it on a whim or passes the paper aisle and remembers she needs napkins. Each shopper is unique and has a different set of goals and priorities in selecting products. Our simulation needs to capture this to some degree.

So, we need to simulate the store as an environment in which shoppers are the agents. The collective action of the shoppers will help us determine which environment, meaning arrangement of products, is most effective at reaching our overall goal of maximizing daily revenue.

Let's lay out each part of the simulation design and see what it entails.

13.1.1 Inventory

To link our simulation to reality, we need to stock our store with products. For each product, we need some knowledge of how often it is purchased and how much it costs relative to other products. The Kaggle Groceries Market Basket dataset contains 169 items and actual purchase frequency over a specific time interval.

item	*purchases*	*item*	*purchases*
whole milk	2512	butter	544
rolls/buns	1808	beef	515
yogurt	1371	chocolate	487
root vegetables	1071	chicken	421
shopping bags	968	cream cheese	389
pastry	874	salty snack	371
bottled beer	791	dessert	364
canned beer	763	UHT-milk	328
fruit/vegetable juice	710	berries	326
brown bread	637	onions	304
frankfurter	579	candy	293
coffee	570	misc. beverages	278

Table 13.1: Grocery store products and their purchase frequency.

We'll use a subset of these items, 24 in total, as our product list. For the cost of each item, we'll select it at random from a beta distribution, so the most frequently purchased items are relatively inexpensive and the least often purchased are more expensive.

The 24 products our store will sell, along with their purchase frequency, are in Table 13.1. Notice, the most frequently purchased item is milk, and one of the least is candy. The experiments of Section 13.4 focus on these two products as indicators of how well the product arrangement has maximized revenue.

Do we expect to learn anything new with this simulation regarding how to arrange products in a grocery store? No, we don't. Intuition and experience already tell us frequently purchased products like milk should be at the back of the store to increase the probability of making an impulse purchase. Still, as we'll see, our simulation is sufficiently sophisticated to capture this essential outcome, so the exercise in designing and implementing a swarm optimization solution is worthwhile.

Let's make the product name/cost list. We start by paring down the 169 items in the Kaggle dataset to the 24 of Table 13.1. The full list of product names and purchase frequencies are already stored in the files `item_counts.npy` and `item_names.npy`. From these, we build the products file like so,

```
import numpy as np
import pickle

N = 24
ci = np.load("item_counts.npy")
ni = np.load("item_names.npy")
ci = ci[::2]
ni = ni[::2]
ci = ci[:N]
ni = ni[:N]

t = np.random.beta(3.5,1,size=10000000)
h = np.array(np.histogram(t, bins=len(ci))[0])
h = h / h.sum()
pv = 10.0*h + 1

p = [ci,ni,pv]
pickle.dump(p, open("products.pkl","wb"))
```

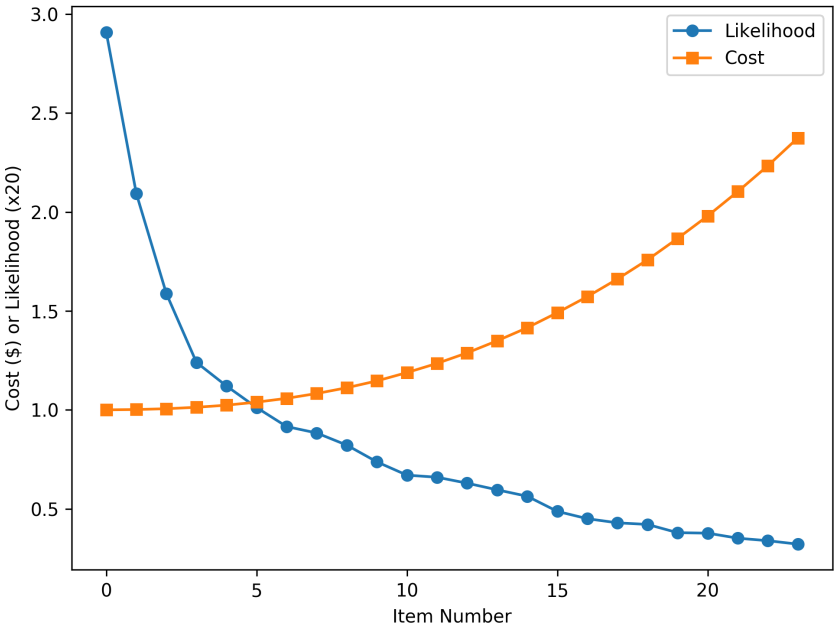


Figure 13.1: Product costs ($) and likelihood of purchase (x20).

We load the full list of counts (`ci`) and names (`ni`). We then keep every other, so our subsample is evenly distributed throughout the list of products. The item count file is already sorted by decreasing frequency of purchase. Finally, we keep the first 24 of these items.

Next, we need to determine a price for each item. The monetary unit is irrelevant; the code will output "dollars", but it doesn't have any actual meaning. What matters is the relative value of each item.

We have the items in decreasing order by purchase frequency. We want the cost to be just the opposite, so rarely purchased items cost more. Therefore, we want to select prices from a distribution biased towards higher costs. We'll do this by setting up a beta distribution, sampling from it many times, though we could have used the probability distribution function as well, and then arrange a histogram with 24 bins as the price for each item.

Therefore, we set `t` to 10 million samples from Beta$(3.5, 1)$ and make a histogram of 24 bins (`h`). Next, we scale `h` by the sum to convert the histogram to a probability distribution, multiply each bin count by 10 and add one to avoid any item cost of zero. The result is in `pv`. A plot of `pv` is in Figure 13.1 showing increasing cost for items as the probability of selecting the item decreases. We store `ci`, `ni`, and `pv` in `products.pkl`. This is our inventory file.

13.1.2 Stores

We seek to learn an arrangement of products for a store. Conceptually, we might think of each arrangement as a separate store, one per particle where the particle's position in the 24-dimensional search space maps to a product order.

Each store is abstracted to its bare minimum. A real store has a specific physical layout. Our conceptual stores are linear. The shopper walks in on the left, moves from product to product along the store until finding the desired product, purchases the product, and departs. Simulating this process is as straightforward as can be: the shopper looks at the list of products, in order, via a simple loop.

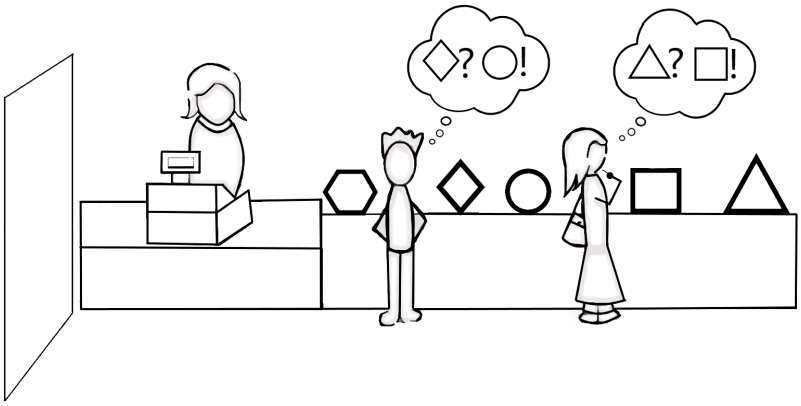


Figure 13.2: Two shoppers traversing the one-dimensional store. (Art credit: Joseph Kneusel)

13.1.3 Shoppers

We've mentioned shoppers several times already, and we've indicated that the shopper's actions based on product arrangement is what we're trying to simulate. Let's be more specific.

For us, a shopper is an instance of the `Shopper` class, which we define in detail in Section 13.2. Conceptually, a shopper is an agent initialized to purchase a randomly selected target product. The randomly selected product is the main reason the shopper is at the store. However, the shopper will also buy three other randomly chosen products if encountered while looking for the target product. These are impulse buys, and they are the key to learning the proper arrangement of products. Without the impulse buy possibility, the shoppers will always find their target, and the arrangement of products will be unimportant.

The target product must be selected appropriately. If the target product was chosen purely at random, we shouldn't expect to be able to learn the best ordering as there is nothing to drive the swarm towards it, even accounting for the impulse buys. However, we have the actual frequency of purchase information for our products, so when we select the target product, we use this information to choose items where those most frequently purchased are most likely to be chosen as the target.

If the products are arranged so the most commonly selected product is the first one encountered, shoppers will often find their target almost immediately and will, therefore, have little opportunity to find their impulse products. To maximize the revenue for a given shopper, we want the target product to be near the rightmost part of the store, the back of the store. That way, the shopper will also hit the impulse buy items and purchase them, too. As we'll see, the swarms figure this out as well. And, as Figure 13.1 demonstrates, impulse products are less often purchased but generate more revenue when they are.

Figure 13.2 presents a cartoon of a store and two shoppers. Shoppers enter the store on the left and walk left to right, examining products to find their target product. Each shopper is thinking of a target product (exclamation point) and one of their impulse products (question mark). The shopper on the left has found his impulse purchase before his target, so he'll purchase it along with the target. The shopper on the right has found her target product, but not her impulse product, as it's further still to the right. She'll buy her target but not the impulse product.

The total revenue from a full set of shoppers becomes the objective function value. We have no idea what the maximum can be, so we'll use the negative of the revenue and seek to minimize it.

13.1.4 Running the Simulation

Let's walk through the simulation process. At the start, a swarm is initialized. Each particle of the swarm represents a possible arrangement of products, or, conceptually, a store. We also initialize a

random set of shoppers and pass them to the objective function class instance. This population is used for the entire search.

With initial stores and shoppers in place, each iteration of the swarm lets the shoppers shop at the stores and keeps a tally of the amount of money spent. This is the objective function value for the particles. When all stores have been visited, the swarm updates according to the selected algorithm, and the process repeats until all iterations have been exhausted. Finally, the best arrangement of products found is returned as the result of the search.

We have the design, now let's implement it starting with the Shopper class.

13.2 The Shopper

We intend to use a random collection of shoppers as the means by which we'll evaluate a particular store's arrangement of products. Each shopper is an instance of the `Shopper` class. Let's start with the constructor,

```
def __init__(self, fi, pv):
    self.item_freq = fi
    self.item_values = pv
    self.target = Select(fi)
    self.impulse = np.argsort(np.random.random(len(fi)))[:3]
    while (self.target in self.impulse):
        self.impulse = np.argsort(np.random.random(len(fi)))[:3]
```

The constructor accepts `fi` and `pv`, the list of products sorted by actual purchase frequency (highest to lowest) and the cost of each product generated from the code in Section 13.1.1. We'll see below when we define the `Objective` class how we convert purchase counts to frequencies.

The shopper needs a target product, so that is selected next via `Select`,

```
def Select(fi):
    t = np.random.random()
    c = 0.0
    for i in range(len(fi)):
        c += fi[i]
        if (c >= t):
            return i
```

`Select` returns an index into the list of products using the product frequencies in decreasing order and a random value in $[0, 1)$. The index is the product number the shopper will look for. By summing the frequencies until we match or exceed the random value (`t`), we are more likely to select commonly purchased items instead of rarely purchased items.

With the target product chosen, we now select three impulse products at random. To do that, we use this programming idiom,

```
self.impulse = np.argsort(np.random.random(len(fi)))[:3]
```

where a random set of values matching the number of products is generated and the sort order determined (`np.argsort`). Notice, it's the set of indices that would sort the random list we need, and only the first three, hence `[:3]`.

For example, assume `np.random` has returned,

```
0.4033, 0.6909, 0.9607, 0.1794, 0.1629, 0.6424
```

then `np.argsort` will return,

```
4, 3, 0, 5, 1, 2
```

as reordering the random values this way will sort them from smallest to largest.

For our purposes, we use the first three of these as product numbers, meaning this shopper would consider products 4, 3, and 0 to be impulse purchases. The `while` loop in the constructor for the `Shopper` class is there to ensure the target product is not also selected as an impulse purchase option.

The only other method in `Shopper` is `GoShopping`,

```
def GoShopping(self, products):
    spent = 0.0
    for p in products:
        if (p == self.target):
            spent += self.item_values[p]
            break
        if (p in self.impulse):
            spent += self.item_values[p]
    return spent
```

This method is called by `Evaluate` in the `Objective` class. `Products` is a list of product items in order where the order is generated from the particle position.

The shopper examines the list, item by item, in order, seeking the target. When the target item is found, its value is added to `spent` and the loop ends. While looking for the target, if the current item number (`p`) is in the set of `impulse` items list, the shopper buys it as well. When the target is found, and it always is for our simulation, the total amount spent by the shopper is returned.

The `Shopper` class is used exclusively by the objective function, so this is the appropriate time to detail it. The code is straightforward,

```
class Objective:
    def __init__(self, nshoppers, ci, pv):
        self.nshoppers = nshoppers
        self.item_freq = ci / ci.sum()
        self.item_values = pv
        self.fcount = 0
        self.shoppers = []
        for i in range(nshoppers):
            shopper = Shopper(self.item_freq, pv)
            self.shoppers.append(shopper)
    def Evaluate(self, p):
        self.fcount += 1
        order = np.argsort(p)
        revenue = 0.0
        for i in range(self.nshoppers):
            revenue += self.shoppers[i].GoShopping(order)
        return -revenue
```

where we have only two methods, the constructor and `Evaluate`.

The constructor accepts the number of shoppers to generate, the item purchase list (counts ordered by decreasing purchase frequencies), and each item's assigned price. The counts are converted to frequencies (`item_freq`), thereby mapping them to $[0, 1]$. The number of shoppers, frequencies, values, and call counter (`fcount`) are stored and initialized.

Next, the set of shoppers is created. Recall, this collection is used for the entire simulation. A simple loop appending new `Shopper` instances to `shoppers` is all that's required.

During the search, calls to `Evaluate` pass in the particle position, as always (`p`). We bump the call counter and then map the particle position to the order of products with,

```
order = np.argsort(p)
```

As we'll see, particle positions are continuous vectors in $[0, 1)$ with one element for each product. We have 24 products, so our particle positions are 24-dimensional vectors. The actual elements

of the position are not themselves the product numbers. Instead, the sort order generated by the continuous values creates the item numbers. We saw above how `np.argsort` gives us the set of indices into a vector that will sort it in order.

Why did we do this? We did this because if we wanted the particle positions to be the actual item numbers, we'd need to search a space bounded by $[0, 23]$ with discrete values for each element. It's simpler to let the swarms operate on their natural inputs, namely floating-point values. We've seen other cases where we have enforced discrete values via the `Validate` method of some subclass of `Bounds`, but for this task, it seemed less cumbersome to use the sort order instead. This level of indirection, as it were, from what the particle position represents to what we want to manipulate, works rather well. Also, we've stated previously that one power of swarm techniques is how we map the swarm particles to a solution to our problem. Using the sort order of a continuous vector is just one more example of that.

The `Evaluate` method passes the `order` list to each of the shoppers and tallies the revenue generated by this ordering of products. When done, it returns the negative of this value.

13.3 The Store

Section 13.2 above detailed how we'll implement shoppers. Let's now look at the main portion of `store.py` which implements the store simulation. The layout is quite similar to our previous experiments.

To tailor the simulation to our task, we need only implement the `Objective` class, described above. We're using particle positions in $[0, 1]$, so the boundaries are straightforward; no subclass of `Bounds` is needed.

The `store.py` script starts like all our experiments: it loads the necessary modules, including those of the framework, and expects a command line like this,

```
> python3 store.py products.pkl 50 30 4000 GA RI
```

where `products.pkl` is our product list. See Section 13.1.1 for details on how this file was created. We use the same file for all simulations. Next, we'll use 50 shoppers, a swarm of 30 particles run for 4000 iterations. The selected algorithm is GA with random initialization. We'll examine the script output in the next section; for now, let's look at the relevant code.

First, we parse the command line and load the products file,

```
products = pickle.load(open(sys.argv[1],"rb"))
nshoppers = int(sys.argv[2])
npart = int(sys.argv[3])
niter = int(sys.argv[4])
alg = sys.argv[5].upper()
itype = sys.argv[6].upper()
```

Next, we set up the framework components,

```
ci = products[0]  # product counts
ni = products[1]  # product names
pv = products[2]  # product values
pci = ci / ci.sum()  # probability of being purchased
N = len(ci)          # number of products

ndim = len(ci)
b = Bounds([0]*ndim, [1]*ndim, enforce="resample")

if (itype == "QI"):
    i = QuasirandomInitializer(npart, ndim, bounds=b)
elif (itype == "SI"):
    i = SphereInitializer(npart, ndim, bounds=b)
```

```
else:
    i = RandomInitializer(npart, ndim, bounds=b)

obj = Objective(nshoppers, ci, pv)
```

We split the products input into counts (`ci`), names (`ni`), and costs (`pv`). We set `pci` to the frequency of purchase and `N` to the number of products. We'll use `pci` and `ni` below. All the swarm cares about is `ci` and `pv`.

The number of products in the store fixes the dimensionality of the swarm (`ndim`). Bounds are $[0, 1]$ in all dimensions with resampling on boundary violations. In this case, using clipping makes no sense as the particle position's sorted order is what we are after. Fixing elements at zero or one would break that mapping.

The proper initialization object is created, followed by an instance of the `Objective` class passing in the number of shoppers, the list of product counts, and product costs.

Initialization of the swarm means a random $[0, 1)$ vector for each particle, implying a random ordering of products.

The proper `swarm` object is next. We'll skip the code here; we've seen it many times before. For our experiments, we're using each algorithm in its default configuration. For PSO we use a default instance of `LinearInertia`.

With all this preparation, running the search is anticlimactic,

```
swarm.Optimize()
res = swarm.Results()
```

where everything interesting happens after the run and is contained in the results, `res`.

This script creates no output files. Instead, it generates a report dumped to the screen. We'll capture this output during the experiments of Section 13.4 and parse it to extract the values used to generate the plots, etc.

The output includes statistics like the number of updates, function evaluations, and search time. We also output the maximum revenue and the product ordering that led to it,

```
print()
print("Maximum daily revenue %0.2f (time %0.3f seconds)" %
       (-res["gbest"][-1], en-st))
print("(%d best updates, %d function evaluations)" %
       (len(res["gbest"]), obj.fcount))
print()
print("Product order:")
order = np.argsort(res["gpos"][-1])
ni = ni[order]
pci= pci[order]
pv = pv[order]
for p in range(len(pv)):
    print("%25s  (%4.1f%%) ($%0.2f)" % (ni[p], 100.0*pci[p], pv[p]))
    if (ni[p] == "whole milk"):
        milk_rank = p
    if (ni[p] == "candy"):
        candy_rank = p
print()
print("milk rank = %d" % milk_rank)
print("candy rank = %d" % candy_rank)
print()
```

We know physical grocery stores often put the milk at the back of the store. It's a frequently purchased item, so making people walk through the store to get to it maximizes the likelihood they will make impulse purchases. Likewise, candy is often near the front of the store to maximize an impulse purchase probability. Therefore, if our simulation is working and sufficiently sophisticated,

it should generate product sequences that place candy near the front of the store and milk near the back. This is why the code above reports the milk and candy rankings.

To conclude the script, we split the products, in the selected order, into two halves and report the median probability of being selected and the median product cost,

```
    print("Upper half median probability of being selected = %4.1f" %
          (100.0*np.median(pci[:N//2]),))
    print("                            median product value = %4.2f" %
          (np.median(pv[:N//2]),))
    print("Lower half median probability of being selected = %4.1f" %
          (100.0*np.median(pci[N//2:]),))
    print("                            median product value = %4.2f" %
          (np.median(pv[N//2:]),))
    print()
```

Again, knowing how actual stores are typically arranged, we hope that the swarm generates a product list where less often purchased but more expensive items are near the front of the store (the top half of the product list) while less costly but more frequently purchased items migrate to the back of the store (the bottom half of the list).

13.4 The Simulation

A typical run of `store.py` using the command line given in Section 13.3 produces

```
Maximum daily revenue 252.18 (time 809.094 seconds)
(33 best updates, 120030 function evaluations)

Product order:
              canned beer  ( 4.4%) ($1.08)
                    candy  ( 1.7%) ($2.23)
                   butter  ( 3.1%) ($1.29)
                  dessert  ( 2.1%) ($1.76)
                  chicken  ( 2.4%) ($1.49)
                chocolate  ( 2.8%) ($1.42)
                 UHT-milk  ( 1.9%) ($1.87)
                  berries  ( 1.9%) ($1.98)
              salty snack  ( 2.1%) ($1.66)
                   pastry  ( 5.1%) ($1.04)
                   coffee  ( 3.3%) ($1.23)
          misc. beverages  ( 1.6%) ($2.37)
             cream cheese  ( 2.3%) ($1.57)
              brown bread  ( 3.7%) ($1.15)
              frankfurter  ( 3.4%) ($1.19)
    fruit/vegetable juice  ( 4.1%) ($1.11)
                     beef  ( 3.0%) ($1.35)
          root vegetables  ( 6.2%) ($1.01)
             bottled beer  ( 4.6%) ($1.06)
                   yogurt  ( 7.9%) ($1.01)
               whole milk  (14.5%) ($1.00)
                   onions  ( 1.8%) ($2.10)
            shopping bags  ( 5.6%) ($1.02)
               rolls/buns  (10.5%) ($1.00)

milk rank = 20
candy rank = 1

Upper half median probability of being selected =  2.3
                            median product value = 1.58
```

```
Lower half median probability of being selected =  4.3
                            median product value = 1.09
```

At the top, we're told the best arrangement found generated $252.18 for 50 shoppers. We see the run time as well, not optimized, as always. We're also told there were 33 times the best arrangement was updated and a total of 120,030 objective function evaluations. The first number in parentheses is the actual probability of purchase for the item from our list of products. The next number is the cost we assigned to the item.

We're using milk and candy and proxies to tell if our ordering it a good one or not. The lower milk's ranking and the higher candy's, the better we believe the ranking to be. For this run, milk was ranked 21st out of 24 products. Candy was ranked second (counting the top item as rank zero). Therefore, from this alone, we might think we have a good result.

The bottom portion of the report gives us the median selection probabilities, as percents, for the top half of the list and then the bottom half. This is the median of the actual purchase probability. We expect that the lower half probability of being purchased will be higher than the upper half, and it is in this case.

Finally, we get the median value of the top and bottom half of the list. Like the probability of purchase, we expect the top portion of the list to be more expensive than the bottom, and in this case, it is.

Let's configure and run a series of experiments testing each algorithm multiple times to gain a broad picture of how well they do on this task.

13.4.1 Testing the Algorithms

A simple shell script, `store_experiments`, contains instructions for twenty runs of each algorithm with the output of each run piped to a file.

Each run uses a swarm of 30 particles and 4000 iterations. We cannot stop early; we don't know what a good score is, but we hope that 4000 iterations of the swarm will move us in the right direction.

We've used the algorithms extensively to get to this point in the book. We have developed some intuition about them. Let's make predictions on what we expect to see based on that intuition.

As the order of the products changes during the search, we do think pushing more commonly purchased, but cheaper items to the bottom of the list (the back of the store) will lead to improved revenue, so this isn't a blind search for one, magic arrangement of products. As the swarm moves in a specific direction, we expect it to pull other particles with it as the various update rules are followed.

Therefore, we anticipate algorithms with strong interactions between particles will do well. This means DE, PSO, Jaya, and GWO. We've seen Jaya converge slowly in some cases, and we've seen inconsistent performance from GWO, so we'll keep an eye out for those effects.

What about GA and RO? The space we're searching is 24 dimensional and bounded uniformly, so there is a good chance evolution with crossover and mutation might lead to good results. Plus, 4000 iterations is more than we've typically worked with, and we know GA likes to take its time exploring.

RO, on the other hand, is a parallel, local-oriented search with no interaction between particles. Like the proverbial monkeys at their keyboards, RO will explore and explore and find a satisfactory arrangement with enough time. Will 4000 iterations be enough? Might we need to adjust RO's one parameter a bit to cover the space more rapidly? We'll see.

We'll run `store_experiments` as specified above. When the script is done, we'll have six output directories, one for each algorithm, and inside of those directories, we'll have twenty output file like the one above. Running `store_analysis.py` parses the output files, generates an overall summary, and produces a series of plots showing the milk and candy rankings per algorithm.

The mean ± SE milk and candy rankings across all twenty runs and algorithms are,

```
PSO :  milk: 20.60 +/- 0.59 (23, 15), candy: 6.85 +/- 1.22 (18, 0)
DE  :  milk: 21.25 +/- 0.35 (23, 18), candy: 5.50 +/- 1.21 (23, 0)
```

```
RO   :  milk: 19.65 +/- 0.84 (23, 11), candy: 6.70 +/- 0.98 (15, 2)
GWO  :  milk: 20.30 +/- 0.75 (23, 10), candy: 5.45 +/- 0.88 (13, 0)
Jaya:  milk: 20.65 +/- 0.61 (23, 14), candy: 6.50 +/- 1.28 (16, 0)
GA   :  milk: 21.05 +/- 0.75 (23,  9), candy: 5.80 +/- 1.21 (15, 0)
```

The minimum and maximums are also given in parentheses.

If we believe the rankings of milk and candy indicate a good ordering, then we can order the results by the difference in the means, treating larger as better. Doing so gives

DE	15.75
GA	15.25
GWO	14.85
Jaya	14.15
PSO	13.75
RO	12.95

handing the top rank to DE, followed by GA with a strong showing from GWO.

Of course, daily revenue is the objective function we seek to maximize. The mean daily revenue ($\pm$ SE) leads to the following ordering,

DE	\$247.16 $\pm$ 1.35
PSO	\$246.93 $\pm$ 1.25
Jaya	\$245.41 $\pm$ 1.37
GWO	\$245.14 $\pm$ 1.63
GA	\$245.09 $\pm$ 1.93
RO	\$220.59 $\pm$ 2.09

which again gives the nod to DE followed closely by PSO. However, the standard errors are large. A t-test between DE and GA returns $p = 0.39$ indicating there is likely no meaningful difference between DE, PSO, Jaya, GWO, and GA in terms of maximizing revenue. However, a t-test between DE and RO leads to $p < 0.0001$, so it is fair to say that RO is not doing as well as we might hope.

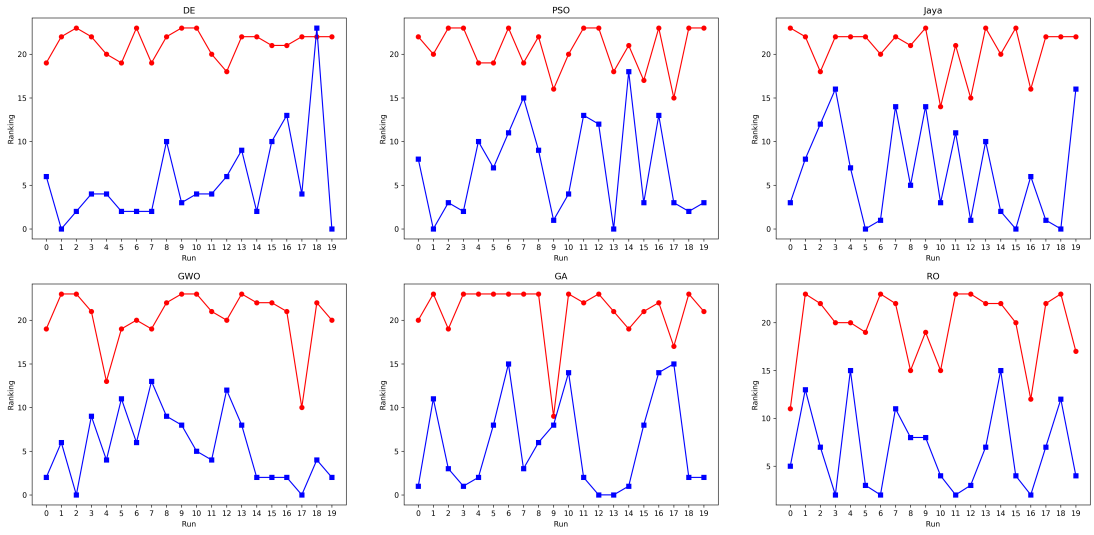


Figure 13.3: Per run rankings of milk (circle) and candy (square) by algorithm.

Figure 13.3 presents the per-run milk (circle) and candy (square) ranks by algorithm. In general, the algorithms do learn to separate the two. Only once, run 18 of DE, sees milk and candy trade places, though GA run 9 comes close. So, the searches are separating our proxy products as we

expect, even RO, but the remaining ordering of the products leads to increased revenue, and here, RO seems to be doing poorly.

The `store_convergence.py` script analyzes a single run of each algorithm using 50 shoppers, 30 particles, and 4000 iterations. We want to see how the swarms converge, and we'll do so by tracking the best revenue found as a function of iteration. The result is Figure 13.4.

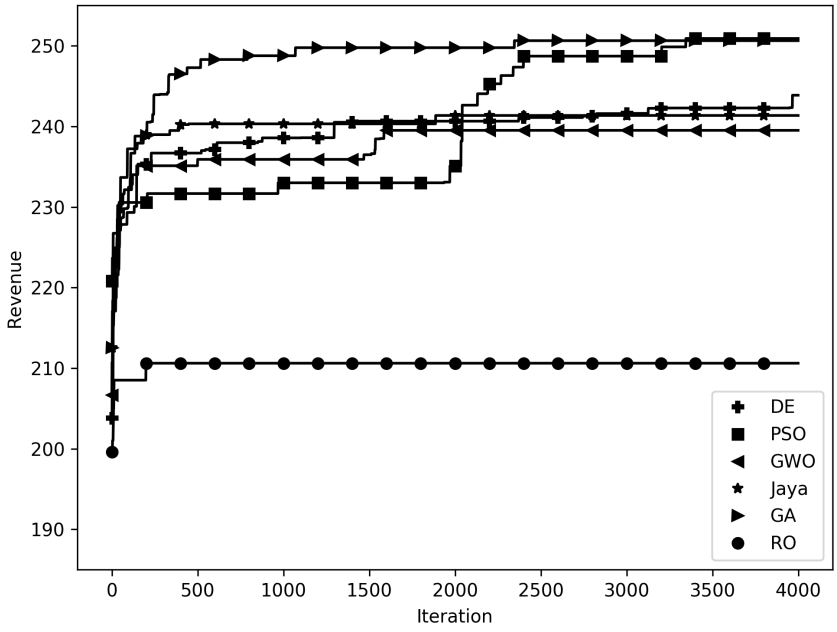


Figure 13.4: Best revenue found by iteration and algorithm type.

All the algorithms, except RO, show a rapid increase in revenue early on, which we might expect given random initialization. However, after about 100 iterations, most algorithms slow dramatically except GA and PSO, though the latter has a long period of little improvement followed by several jumps at later iterations. Jaya plateaus quickly and shows little change after that. GWO makes one jump around iteration 1500, and DE takes small steps throughout the search. Figure 13.4 is a single snapshot of the overall performance of the algorithms. We saw above that repeated applications lead to virtually equivalent performance between all the algorithms except RO. RO's performance in Figure 13.4 is definitely lackluster.

13.4.2 Working with RO

What makes RO do poorly on this task? We can look at the ordering of products generated by RO and, for comparison, DE, to see if anything pops out at us.

Figure 13.6 shows the final product order generated by a single search each for RO and DE. The swarm was slightly larger than what we've used above, 40 particles, and the search ran for 16,000 iterations. The final revenue generated, recalling that revenue is dependent somewhat on the randomly generated set of shoppers for each search, was \$245.32 for RO and \$254.67 for DE, following the typical ordering we saw above, though the RO revenue was higher. We'll explore this effect shortly.

The plots of Figure 13.6 show us what we hoped to see. On the left, for both algorithms, more expensive items were near the front of the store (shoppers view products from left to right), though DE had more such items than RO. For RO, the middle part of the store contained most of the expensive items. And, for both, the back of the store (right) contained cheaper items, on average, as we hypothesized.

The right side of Figure 13.6 shows the same product ordering, but the y-axis is now the cost of the product times the probability of it being purchased. Here, we expect to see less frequently

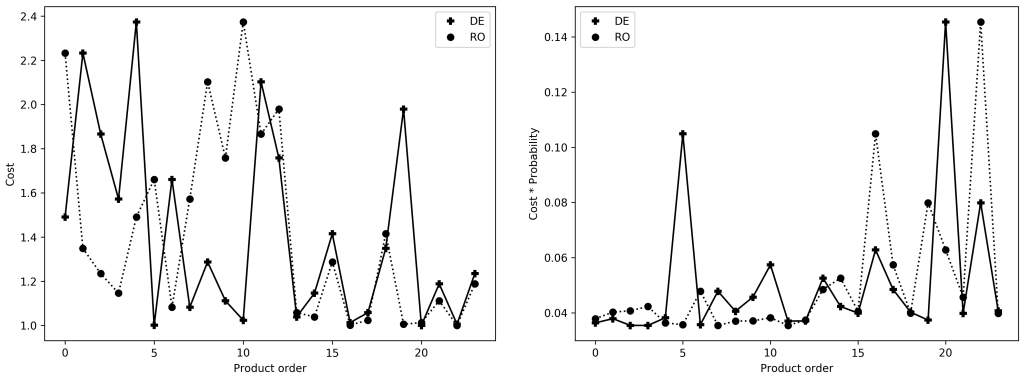


Figure 13.5: (Left) Product order and cost. (Right) Product order and cost times probability of selecting the product.

purchased items on the left and more regularly purchased items on the right. In a sense, the cost times the probability is an expectation of the amount of revenue generated by the product given its current placement. Again, overall, the less frequently purchased items ended up where expected, as did the more regularly purchased items. Notice, for DE, one item near the front of the store has a higher probability of being purchased than the others. This item is bread rolls with an actual purchase frequency of about 10%. Would this sort of ordering repeat itself? Is DE, and by extension the other algorithms except for RO, capturing something more beyond the straightforward product order we anticipated we'd see? Repeated experiments might show something, but it's also likely that the placement of bread rolls was due to the particular set of shoppers generated for the DE search.

We saw above that RO performed poorly relative to the other algorithms. The product ordering run for 16,000 iterations, instead of only 4000, showed a marked improvement in RO's results. We know RO is slow and will not converge quickly; its movement through the search space is controlled by the value of η, the scale factor. Let's explore RO's performance as a function of η.

The script `store_eta_experiments` uses `store_ro.py`, a slight variation of `store.py`, to run a series of experiments where η is varied from 0.01 to 2.0. The `store_eta_analysis.py` script generates a summary of the experiments for six runs at each η value. For all experiments, we persist in using 50 shoppers, 30 particles, and 4000 iterations.

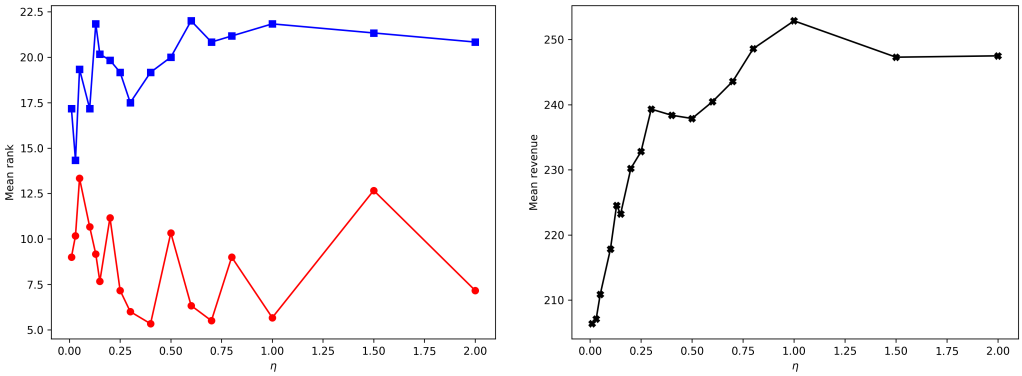


Figure 13.6: (Left) Milk and candy positions as a function of η. (Right) Revenue as a function of η.

Figure 13.6 shows us the results. On the left, we see the maximum separation between milk and candy positions occurs when $\eta = 1.0$. Likewise, at $\eta = 1.0$, we get the largest revenue of \$252.87, in line with the results found by the other algorithms. Therefore, in this case, it pays to adjust the swarm's parameters.

13.4.3 Varying the Number of Shoppers

All of our experiments fixed the number of shoppers at 50. What effect might we see if we vary the number of shoppers?

Let's use Jaya as a representative algorithm. We'll fix the random number seed when selecting the initial swarm configuration and again after setting up the `Objective` instance. Therefore, the only variation between runs will be the random configuration of the N shoppers. We'll vary N from a low of 10 to a high of 1000.

The code is in `store_shoppers.py` and it makes use of a slight variation of `store.py` to fix the random number seed, see `store_shop.py`.

Running `store_shoppers.py` creates multiple output files in the `shoppers` directory. We'll create two NumPy files by hand using the `.txt` output to capture the best revenue (`revenue.npy`) and another with the corresponding number of shoppers (`nshoppers.npy`). As the number of shoppers increases, so does the revenue per day, so we'll look at the revenue per shopper.

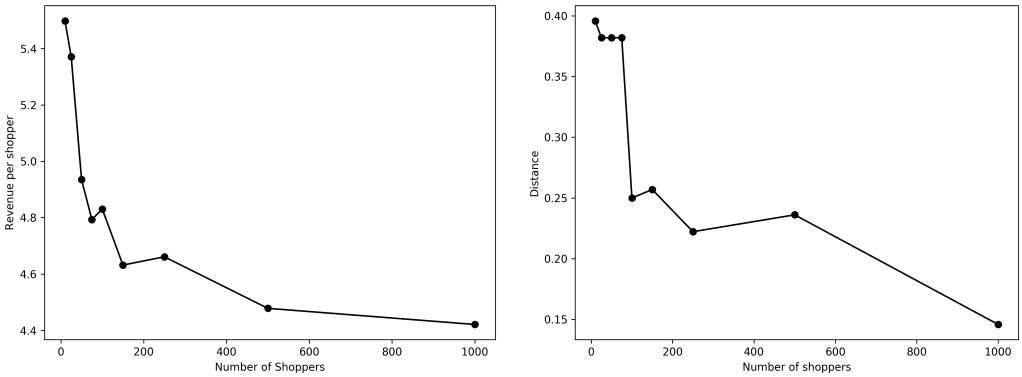


Figure 13.7: (Left) Revenue per shopper. (Right) Distance between product order found and order implied by the actual purchase frequency.

The left side of Figure 13.7 presents the revenue per shopper for the best product ordering as a function of the number of shoppers. We might expect the revenue per shopper to be roughly constant; however, this is not the case. As the number of shoppers increases, the revenue per shopper decreases. Why?

The swarms seem to be finding product arrangements tailored to the particular set of shoppers, one that is fitted to the "peculiarities" of their buying habits and therefore leads to a specific arrangement that maximizes revenue. In a sense, this is much like the overfitting that can happen when training a neural network – the network becomes very good at the minutiae of the training set but loses the ability to generalized to new data.

To test this idea, we'll examine the order of the products as a function of the number of shoppers. As our one-dimensional store is simple, we might expect over time that the product order will become closer and closer to the product order found when arranging the products from least likely to be purchased to most likely.

Each search generated a product order found by applying `np.argsort` to the best swarm `gpos` vector. We also know that the products are arranged in `products.pkl` from most frequently purchased to least purchased. Therefore, if we calculate a distance between the product order found by the swarm and the reverse of the product order, we'll have a measure of how far the swarm order is from the "ideal." We're using integers as product identifiers, so the previous sentence boils down to measuring a distance between the swarm product order and the vector

$$(0, 1, 2, 3, \ldots, 22, 23)$$

since we have 24 products. We'll use the absolute value of the difference between components and sum to get a single number. If the swarm's product order matches the reverse of the product purchase

frequencies, the distance is zero. So, smaller distance implies an order more like the expected "ideal" order. The code we need is in the `product_order.py` script. The distance calculation is

```
def dist(b):
    a = np.arange(len(b))[::-1]
    w = np.arange(len(b))
    n = np.argsort(b)
    z = np.abs(w-a).sum()
    return np.abs(a-n).sum() / z
```

where `b` is the best `gpos` found by the swarm, `a` is the ideal order, and `w` is the worst possible order, the flip of `a`. First, sort `b` to get `n`, the swarm product order, then calculate `z`, the largest possible distance, and return the distance between `n` and `a` as a fraction of this largest possible distance. The result is the right side of Figure 13.7.

The distance decreases as a function of the number of shoppers. By the time we have 1000 shoppers, any benefit to maximizing revenue by tailoring product arrangement to the particulars of the crowd has been washed out and the resulting product order is quite close to the ideal.

In this chapter, we developed a grocery store simulation. We showed it was sufficiently realistic to capture the essential tenet of grocery store product layout: put the most frequently purchased items in the back. We learned that all of our swarm algorithms were capable of discovering this fact, though we needed to tweak RO's η parameter a bit to get it to work well.

Optimizing characteristics of a system with swarms using simulation opens up to us a wide range of possible applications. It seems likely that swarms would work well even with very sophisticated simulation environments and multiple agents. We might want to optimize some characteristics of the environment, as we did here with product placement, or the agents themselves.

Chapter 14

Discussion

We covered a lot of ground in this book, both figuratively and (quasi)-literally via the cell tower experiment. In this final chapter, let's pull everything together with a brief discussion. Specifically, let's conclude the book by offering some final thoughts on each of the algorithms, followed by some thoughts on the book as a whole.

Of the dozens and dozens of potential metaheuristic algorithms out there, six were selected for our framework and experiments. The selection process was not scientific but based on experience, diversity of approach, and a desire to include the tried-and-true with the up-and-coming. Additionally, and particularly for the newer algorithms, ease of use and implementation were essential criteria. After all, the point of building the framework was to illustrate in code how the algorithms function and to provide a core set of basic implementations for you to use when developing your own versions for various projects.

Perhaps the most valuable lesson from this book is that the "no free lunch" theorem is true. It is necessary to be familiar with many different approaches to metaheuristics to select a good algorithm for a particular task, or, at least, to have something else to try should your favorite algorithm fail, as it most assuredly will at some point.

Differential Evolution

Differential evolution has been around for some time and is the topic of many research papers and books. Like seemingly all metaheuristics, variations abound. Our implementation focused on two standard approaches with slight tweaks, such as adding the "toggle" option to shift between "random" and "best" modes.

It's plain that DE is a robust algorithm. However, DE was a terrible option for the 0-1 knapsack experiment and performed relatively poorly at image segmentation.

Differential evolution seems to operate best when the search space is continuous and the objective function is more mathematical. For example, the 0-1 knapsack problem is discrete, binary, and combinatorial. DE's penchant for rapid convergence, seen repeatedly in our convergence plots, indicates exploitation trumps exploration more often than not. All the same, differential evolution should be a prominent tool in your swarm optimization toolbox – well-worn and frequently applied.

Particle Swarm Optimization

Particle Swarm Optimization is another go-to algorithm, one that I have used frequently over the years.

There exist a legion's worth of PSO variants. We used canonical PSO for most of our experiments even though the code supports the bare-bones variant. Similarly, we did not use neighborhoods, though the ring topology is an option. To keep the code simple, we ignored more complex but arguably better neighborhood topologies like von Neumann, which I've found particularly helpful

at times. Therefore, we shouldn't be too critical of PSO's performance in our experiments. PSO is a robust, general-purpose algorithm. I think it is suitable for both highly numerical tasks with a continuous, smooth search space and more abstract search spaces. However, the latter case is likely best served by PSO variants tailored to discrete spaces or which emphasize exploration over exploitation.

For the melody experiments, PSO generated pleasant melodies with variation. DE converged strongly to a particular melody in the set of similar melodies and minimized variation. From a creativity perspective, PSO might be the better algorithm in this case.

Particle swarm optimization should be a primary component of your swarm optimization toolbox. At some point, do take time to explore some of the many variations, including the von Neumann topology, and the literature, including books.

Grey Wolf Optimizer

As a newer algorithm, I had high hopes for the Grey Wolf Optimizer – doubly so because it has (virtually) no adjustable parameters. While it did not entirely disappoint, there is a tendency to behave erratically, with excellent results followed by complete failures. Granted, we seldom adjusted the one parameter and saw that we needed to do so for our example problem in Part I of the book.

GWO did converge more rapidly than Jaya in many cases, which was helpful. I would continue to work with this algorithm as the plethora of references implies suitability for many tasks. It is worth noting that GWO, along with RO, was well-suited to the 0-1 knapsack experiment when other algorithms like DE, Jaya, and GA were complete failures. Additionally, GWO performed reasonably well for most of the other experiments, occasional erratic outbursts aside.

Jaya

Our other new algorithm, Jaya, has no tunable parameters. The simplicity of the algorithm is a strength. Many algorithms, like PSO, focus on the best position the swarm has found, but Jaya also considers the worst and tries to move away from it. The candidate position tried for each particle is a straightforward blending of the two, of the best and the worst. Additionally, unlike PSO, the candidate position is only selected should it result in an improved objective function value.

However, we saw several instances where Jaya took many iterations to converge or failed to converge at all. With no tunable parameters, Jaya either works well for a task, or it doesn't.

The `CandidatePositions` method in `Jaya.py` uses two uniform $[0, 1)$ random vectors to scale the difference between the current particle position and the swarm best and worst. Might it be possible to add a tuning parameter here to balance the two terms? We do not have space to explore that option, but it isn't hard to do, should a curious reader wish to try it.

The elegance of Jaya makes it an algorithm to be aware of and try. Jaya performed best on image segmentation and enhancement. It was least-suited to the 0-1 knapsack and, somewhat surprisingly, the merging of melodies. Many have been victorious with Jaya, so it's worth keeping in your back pocket to pull out from time to time.

Genetic Algorithm

The genetic algorithm functions differently from the other algorithms, though it has a passing similarity to differential evolution and random optimization. GA is similar to DE as both use genetic crossover and mutation concepts, though DE's version is more sophisticated. GA is similar to RO in that there is only an implicit rule favoring better positions in the search space for the entire swarm. RO is simpler still, but in GA, the increased likelihood of selecting a more fit candidate for crossover is all that drives the swarm towards better positions in the search space. There is no intention, only a hopeful percolation of reasonable solutions via breeding and random mutation.

The convergence rate for GA is known to be slow at times. This makes sense as in place of an update rule using important positions known from the entire swarm, like PSO, GWO, or Jaya, GA relies on improving positions by breeding with the more fit members of the swarm.

GA performed well on image segmentation, cell tower placement, and the grocery store simulation experiments. Conversely, GA performed poorly on the 0-1 knapsack, curve fitting, image registration, and image enhancement tasks. We know GA evolves the swarm by random mixing and mutation and not by directly accounting for the objective function values. Instead, objective function values are used implicitly by increasing the probability of selecting a more suitable breeding partner. Therefore, we might expect GA to work well for tasks where the search space is not smooth. For example, suppose the task is such that one position in the search space has a very different objective function value than a position close to it. In that case, GA might be able to accomplish the goal where other algorithms fail as the other algorithms will, by design, attempt to use knowledge about the positions of the other particles to generate a new set of positions. For example, I've been successfully applying this GA algorithm to the problem of genetic programming, of evolving code to solve a particular task.

The thought above might be an explanation for GA's overall performance. For example, the grocery store simulation might be such that a minor tweak to the arrangement leads to a very different daily revenue. Additionally, the experiments where GA performed poorly are primarily ones where the objective function is continuous and might be expected to be smooth. Thus, a slight change in position results in a similar small change in the objective function value. It is easy to believe this of curve fitting, image registration, and our approach to image enhancement, as all of these need carefully tuned positions to return an acceptable result.

However, GA performed poorly on the 0-1 knapsack, a likely counter-example to the notion of working well in search spaces that are not smooth. Similarly, GA performed well on image segmentation via fitting multiple Gaussians to the image histogram. Again, this might be thought of as a smooth search space. So, the thought above is to be taken with a grain of salt as a possible contributing factor and not a definitive explanation. Regardless, GA is widely used, typically in another form, and worth including in your optimization toolbox to be a light to you in dark places, when all other lights go out.

Random Optimization

Random optimization is perhaps the oldest and most obvious approach to derivative-free optimization. Barring any auxiliary knowledge of the search domain, it makes sense to start somewhere and move to a new position when somewhere still better is found. If starting somewhere, why not start many "somewheres" and use a swarm of non-interacting particles with an overseer tracking the best position found overall? The extreme simplicity of the algorithm invites modification. We used a reasonable method of selecting new candidate positions: a random position based on a Gaussian distribution. Other selection methods, perhaps a Levy flight, might make sense as well. Or, what about tracking the number of candidate positions tried by iteration and, if little or no improvement happens for some number of iterations, jump to a new, random point in the search space anyway? Such modifications are straightforward to experiment with, adjust `CandidatePositions` in `RO.py` to reflect whatever you have in mind and give it a go.

Random optimization performed well on the 0-1 knapsack problem. One reason might be that RO is all about exploration and has no swarm-wide mechanism to force exploitation. Therefore, becoming trapped in a local minimum might be more challenging for RO, even if it takes longer to find a good solution.

Random optimization was also reasonably good at segmenting grayscale images via fitting the histogram as a sum of Gaussians. This was somewhat unexpected since RO was not good at curve fitting. The difference might lie in the level of precision necessary. A reasonable set of Gaussians, whose center positions set the partitioning of the histogram, might be something random exploration can find. However, falling into a deep minimum to get parameters to many significant figures requires a level of exploitation beyond what RO can typically provide.

Random optimization might be suitable for tasks where "good enough" is a valid solution, but if the goal is to fine-tune a solution to several digits of accuracy, whatever that phrase means for the task, then RO might not be the best option.

Denouement

If you are still with me at the end of the book, congratulations. I hope you enjoyed your journey and have already discovered places where you can apply the techniques and algorithms we explored. After all, the book's point is to introduce you to metaheuristics so that you might use them elsewhere. This book has been a direct attempt at proselytization to spread awareness of metaheuristic algorithms and their diverse utility. Hopefully, you have been converted.

We started the book by introducing the idea of metaheuristic algorithms. We selected a representative set of algorithms from among the masses and built a Python framework for our experiments. This was Part I of the book. In Part II, we explored the algorithms by conducting a series of experiments. The experiments were selected to be diverse in their goals, interesting enough to read through, and simple enough to implement, thereby giving the algorithms and their performance center place at all times.

I had considerable fun developing this book, and I've wanted to write this book for some time as I feel swarm approaches are underappreciated. I also wrote this book because I get a kick out of using randomness for constructive purposes. After all, randomness generating something useful is precisely how we, i.e., humanity, got here in the first place.

To move on from this book, I suggest falling down the rabbit hole before you when entering "metaheuristics" or "swarm optimization" into any Internet search engine. Additionally, there are many technical books on swarm algorithms, books far more detailed than this one.

The idea that most nature-inspired algorithms capture some similar essence (a quintessence?) is supported by the fact that, however loosely, and sometimes very loosely, the algorithms are "nature-inspired." Evolution has solved the optimization problem in many different ways. Always the tinkerer, evolution uses what is at hand to generate multiple, yet similar, solutions to a set of frequently encountered problems. Eventually, humans came along, hijacked those solutions, refined them, and set them loose to do our bidding – randomness generating order, which then used randomness to generate subordinate order. This analogy also indicates that it's likely most nature-inspired algorithms are highly similar, even to the point of being largely interchangeable, and there is little reason to prefer one over another. It's perhaps best to stick with the tried-and-true classics, at least at first.

This book presented a powerful approach to problem-solving, one that is widely applicable, highly generic, and often elegant. It implemented and discussed the algorithms giving you a code template for your use. Please do use it, and if you arrive at a brilliant solution to an otherwise intractable problem, fantastic.
Keep calm and swarm on!

Bibliography

[1] Kenneth Sorensen, Marc Sevaux, and Fred Glover. A history of metaheuristics. *arXiv preprint arXiv:1704.00853*, 2017.

[2] Jörg Stork, Agoston E Eiben, and Thomas Bartz-Beielstein. A new taxonomy of continuous global optimization algorithms. *arXiv preprint arXiv:1808.08818*, 2018.

[3] Krishna Gopal Dhal, Swarnajit Ray, Arunita Das, and Sanjoy Das. A survey on nature-inspired optimization algorithms and their application in image enhancement domain. *Archives of Computational Methods in Engineering*, 26(5):1607–1638, 2019.

[4] Ümit Can and Bilal Alataş. Physics based metaheuristic algorithms for global optimization. 2015.

[5] Kenneth Sörensen. Metaheuristicsthe metaphor exposed. *International Transactions in Operational Research*, 22(1):3–18, 2015.

[6] Fouad Bennis and Rajib Kumar Bhattacharjya. *Nature-Inspired Methods for Metaheuristics Optimization: Algorithms and Applications in Science and Engineering*, volume 16. Springer, 2020.

[7] Xin-She Yang. *Nature-inspired algorithms and applied optimization*, volume 744. Springer, 2017.

[8] Modestus O Okwu and Lagouge K Tartibu. *Metaheuristic Optimization: Nature-Inspired Algorithms Swarm and Computational Intelligence, Theory and Applications*, volume 927. Springer Nature, 2020.

[9] Patricia Melin, Oscar Castillo, and Janusz Kacprzyk. *Nature-inspired design of hybrid intelligent systems*. Springer, 2017.

[10] Xin-She Yang. *Nature-inspired optimization algorithms*. Academic Press, 2020.

[11] Michael A Lones. Mitigating metaphors: A comprehensible guide to recent nature-inspired algorithms. *SN Computer Science*, 1(1):49, 2020.

[12] Albert Y Zomaya. *Handbook of nature-inspired and innovative computing: integrating classical models with emerging technologies*. Springer Science & Business Media, 2006.

[13] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-International Conference on Neural Networks*, volume 4, pages 1942–1948. IEEE, 1995.

[14] James Kennedy. Bare bones particle swarms. In *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No. 03EX706)*, pages 80–87. IEEE, 2003.

[15] Maurice Clerc and James Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[16] Frans Van Den Bergh and Andries Petrus Engelbrecht. A study of particle swarm optimization particle trajectories. *Information sciences*, 176(8):937–971, 2006.

[17] R Rao. Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems. *International Journal of Industrial Engineering Computations*, 7(1):19–34, 2016.

[18] Seyedali Mirjalili, Seyed Mohammad Mirjalili, and Andrew Lewis. Grey wolf optimizer. *Advances in engineering software*, 69:46–61, 2014.

[19] Ravipudi Venkata Rao. *Jaya: an advanced optimization algorithm and its engineering applications*. Springer, 2019.

[20] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[21] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[22] Daniela Zaharie. Parameter adaptation in differential evolution by controlling the population diversity. In *Proceedings of the international workshop on symbolic and numeric algorithms for scientific computing*, pages 385–397, 2002.

[23] Josef Tvrdík. Differential evolution with competitive setting of control parameters. *Task quarterly*, 11(1-2):169–179, 2007.

[24] Philip R Bevington and D Keith Robinson. Data reduction and error analysis for the physical sciences mcgraw-hill. *New York*, 19692:235–242, 1969.

[25] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[26] Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1(1):24–35, 2019.

[27] Apurba Gorai and Ashish Ghosh. Gray-level image enhancement by particle swarm optimization. In *2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*, pages 72–77. IEEE, 2009.

[28] Tahereh Hassanzadeh, Hakimeh Vojodi, and Fariborz Mahmoudi. Non-linear grayscale image enhancement based on firefly algorithm. In *International Conference on Swarm, Evolutionary, and Memetic Computing*, pages 174–181. Springer, 2011.

[29] Sanjay Agrawal and Rutuparna Panda. An efficient algorithm for gray level image enhancement using cuckoo search. In *International Conference on Swarm, Evolutionary, and Memetic Computing*, pages 82–89. Springer, 2012.

[30] PP Sarangi, BSP Mishra, Banshidhar Majhi, and S Dehuri. Gray-level image enhancement using differential evolution optimization algorithm. In *2014 international conference on signal processing and integrated networks (SPIN)*, pages 95–100. IEEE, 2014.

[31] Swagat Kumar Behera, Satyasis Mishra, and Debaraj Rana. Image enhancement using accelerated particle swarm optimization. *International Journal of Engineering Research & Technology*, 4(03):1049–1054, 2015.

[32] Lalit Maurya, Prasant Kumar Mahapatra, and Garima Saini. Modified cuckoo search-based image enhancement. In *Proceedings of the 4th International Conference on Frontiers in Intelligent Computing: Theory and Applications (FICTA) 2015*, pages 625–634. Springer, 2016.

[33] Ammar Mansoor Kamoona and Jagdish Chandra Patra. A novel enhanced cuckoo search algorithm for contrast enhancement of gray scale images. *Applied Soft Computing*, 85:105749, 2019.

[34] Supriya Dhabal and Dip Kumar Saha. Image enhancement using differential evolution based whale optimization algorithm. In *Emerging Technology in Modelling and Graphics*, pages 619–628. Springer, 2020.

[35] Marco Locatelli and Ulrich Raber. Packing equal circles in a square: a deterministic global optimization approach. *Discrete Applied Mathematics*, 122(1-3):139–166, 2002.

[36] HT Croft, KJ Falconer, and RK Guy. Unsolved problems in geometry, 1991. *A26*.

Index

0-1 knapsack, 120
    results, 125

bryozoa, 162

cell towers, 197
    results, 203
circles, 204
    discussion, 210
    results, 209
coin, 162
crane fly, 162
curve fitting, 128
    results, 134

differential evolution
    Candidate method, 100
    CandidatePositions method, 99
    class skeleton, 97
    configuring, 97
    discussion, 228
    nomenclature, 96
    Step method, 98
    update equations, 95

experiments
    0-1 knapsack, 120
    cell towers, 197
    circles, 204
    curve fitting, 128
    grocery store, 213
    image enhancement, 171
    image registration (3 parameter), 159
    image registration (4 parameter), 161
    image segmentation, 164
    merging melodies, 180
    music, 179
    neural network, 139
    novel melodies, 187
    similar melodies, 185
    standard test functions, 116

framework
    Bounds class, 17
    components, 15
    Done class, 23
    general approach, 13
    LinearInertia class, 23
    Objective function class, 16
    QuasirandomInitializer class, 19
    RandomInertia class, 23
    RandomInitializer class, 19
    SphereInitializer class, 21

gaussian test equation (2D), 39
gaussian test equation (5D), 91
    algorithm results, 108
genetic algorithm
    algorithm, 80
    class skeleton, 82
    crossover, 80
    Crossover method, 84
    discussion, 229
    Evolve method, 83
    Mutate method, 84
    mutation, 81
    Step method, 83
global optimization, 4
Grey Wolf Optimizer
    class skeleton, 66
    criticism, 75
    discussion, 229
    Initialize method, 67
    Step method, 68
    update equations, 65
grid search, 4
grocery store, 213
    adjusting random optimization, 224
    number of shoppers, 226
    products, 214
    results, 222, 223
    shoppers, 216
    simulation, 219

image enhancement, 153, 171
    results, 175
image registration, 153
image registration (3 parameter), 159
    results, 160
image registration (4 parameter), 161
    results, 163

image segmentation, 153, 164
    results, 169

Jaya
    CandidatePositions method, 64
    class skeleton, 63
    discussion, 229
    update equation, 63

metaheuristics, 4
music
    Bach chorales, 179
    merging melodies, 180
    merging melodies (results), 183
    novel melodies, 187
    novel melodies (discussion), 196
    novel melodies (results), 193
    similar melodies, 185
    similar melodies (results), 187
    slip jig melodies, 180
    tools, 179
musical modes, 188

neural network, 139
    results, 148
NIST standard curve fit functions, 133
normalized mutual information, 155
    implementation, 156

particle swarm optimization
    algorithm, 44
    bare bones, 48
    BareBonesUpdate method, 55
    class structure, 49
    Constructor, 50, 51
    defaults, 50
    discussion, 228
    Initialize method, 52
    inspiration, 44
    neighborhood, 45
    NeighborhoodBest method, 54
    Optimize method, 51
    parameter relationship, 49
    position equation, 47
    RingNeighborhood method, 55
    Step method, 53
    velocity equation, 46

random optimization
    algorithm, 28
    CandidatePositions method, 37
    class structure, 33
    Constructor, 34
    discussion, 230
    Done method, 36
    Evaluate method, 37
    Initialize method, 35
    Optimize method, 34
    Results method, 38
    Step method, 36
    update equation, 30

scikit-image, 173
scikit-learn, 142
script
    add.py, 162, 163
    analysis.py, 204
    analysis_maker.py, 194, 195
    analysis_points.py, 209
    beale.py, 111
    brute_force.py, 122, 126
    cell.py, 198, 200, 202, 207
    curves.py, 130–134
    curves_plot.py, 134
    DE.py, 97
    de_gaussian.py, 100, 102
    de_gaussian_5d.py, 106
    de_gaussian_5d_plot.py, 105
    de_gaussian_cr.py, 103
    de_gaussian_f.py, 104
    dispersion_plot.py, 70
    download_dataset.py, 143
    easom.py, 111
    enhance.py, 173, 174
    experiment_match.py, 186
    experiments.py, 203, 204
    experiments_maker.py, 193
    experiments_points.py, 208, 209
    extract_frames.py, 159
    fxy.py, 24
    fxy_convergence.py, 71
    fxy_failures.py, 69, 70
    fxy_gaussian.py, 39, 40, 56, 58, 60, 68, 70, 72, 84
    fxy_gaussian_algs.py, 90
    fxy_gaussian_ga_cr.py, 86, 87
    fxy_gaussian_ga_eta.py, 89
    fxy_gaussian_ga_f.py, 88
    fxy_gaussian_ga_multi.py, 92
    fxy_precision.py, 72
    fxy_precision2.py, 72, 73
    fxy_pso_precision.py, 73
    fxy_runtime.py, 74
    GA.py, 82
    GWO.py, 66
    Jaya.py, 63, 229
    jaya_5d.py, 108
    kmeans.py, 168
    knapsack.py, 121, 123–125

make_sample_plot.py, 129
make_sample_plot.py, 129
melody_maker.py, 188, 192, 193, 196
melody_match.py, 185, 186
melody_merge.py, 180–182, 185, 186
merge_test_images.py, 169
midi_dump.py, 180
nn.py, 144, 145, 148
points.py, 207–209
points_plot.py, 71
problem_generator.py, 122, 126
process_images.py, 175
product_order.py, 227
PSO.py, 49, 56
rastrigin.py, 111
rigid.py, 156, 158–163
rigid_pairs.py, 159
rigid_scale.py, 159, 161, 163
RO.py, 33, 38, 230
rosenbrock.py, 111, 117
rosenbrock_de.py, 118
score_test_images.py, 169
segment.py, 165, 167–169
segment_test_images.py, 168
sharpness.py, 163
sphere.py, 111
store.py, 219, 221, 225, 226
store_analysis.py, 222
store_convergence.py, 224
store_eta_analysis.py, 225
store_eta_experiments, 225
store_experiments, 222
store_ro.py, 225
store_shop.py, 226
store_shoppers.py, 226
test_functions.py, 115, 116, 133
standard test functions, 111
swarm algorithms
nature-inspired, 9
nature-inspired (list), 11
random optimization, 28
taxonomy, 8
swarm optimization
agent (definition), 5
algorithm (definition), 5
essence, 4
general algorithm, 6
objective function (definition), 6
particle (definition), 5
representing, 3
search space (definition), 5
swarm (definition), 5