11

Dispersive Flies Optimisation: A Tutorial

Mohammad Majid al-Rifaie
School of Computing and Mathematical Sciences
University of Greenwich, Old Royal Naval College
London, United Kingdom

CONTENTS

11.1 Introduction

The motivation for studying Dispersive Flies Optimisation (DFO) [1] is the algorithm's minimalist update equation which only uses the flies' position vectors for the purpose of updating the population. This is in contrast to several other population-based algorithms and their variants which besides using position vectors, use a subset of several other vectors: velocities and memories (personal best and global best) in particle swarm optimisation (PSO) [2], mutant and trial vectors in differential evolution (DE) [3], pheromone, heuristic vectors in Ant Colony Optimisation (ACO) [4], and so forth [5].

The inspiration for the algorithm comes from the swarming behaviour of flies over food sources, and their dispersing behaviour when facing a threat. The primary aim of the algorithm is numerical optimisation, which is effectively adjusting several parameters of a certain problem and getting a better solution over time. One of the strengths of DFO, and swarm intelligence algorithms in general, is that they can deal with noisy environments or dynamically changing environments, where the solutions are non-stationary. While

DFO has been applied to discrete problems, it is primarily proposed to deal with continuous search spaces. DFO has been applied to various problems, including but not limited to medical imaging [6], optimising machine learning algorithms [7, 8], training deep neural networks [9], computer vision and quantifying symmetrical complexities [10], analysis of autopoiesis in computational creativity [11] and identification of animation key points from 2D-medialness maps [12].

11.2 Dispersive flies optimisation

The swarming behaviour of the individuals in DFO consists of two tightly connected mechanisms, one is the formation of the swarms and the other is its breaking or weakening. The position vector of a fly in DFO is defined as:

$$\vec{x}_i^t = \left[x_{i0}^t, x_{i1}^t, ..., x_{i,D-1}^t\right], \qquad i \in \{0, 1, 2, ..., N\text{-}1\} \tag{11.1}$$

where i represents the i^{th} individual, t is the current time step, D is the problem dimensionality, and N is the population size. For continuous problems, $x_{id} \in \mathbb{R}$ (or a subset of $\mathbb{R}$).

In the first iteration, when $t = 0$, d^{th} component of i^{th} fly is initialised as:

$$x_{id}^0 = \mathrm{U}(x_{\min,d}, x_{\max,d}) \tag{11.2}$$

This, effectively generates a random value between the lower ($x_{\min,d}$) and upper ($x_{\max,d}$) bounds of the respective dimension, d.

On each iteration, dimensions of the position vectors are independently updated, taking into account:

- current fly's position
- current fly's best neighbouring individual (consider ring topology, where each fly has a left and a right neighbour)
- and the best fly in the swarm.

Therefore, the update equation is

$$x_{id}^{t+1} \quad = \quad x_{i_n d}^t + u(x_{sd}^t - x_{id}^t) \tag{11.3}$$

where,

- x_{id}^t: position of the i^{th} fly in d^{th} dimension at time step t
- $x_{i_n d}^t$: position of $\vec{x}_i^t$'s best *neighbouring* individual (in ring topology) in d^{th} dimension at time step t

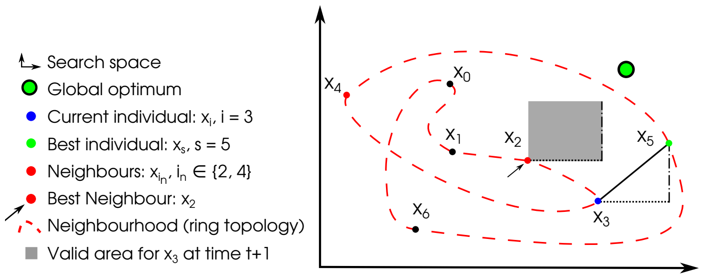
Search space
Global optimum
Current individual: x_i, i = 3
Best individual: x_s, s = 5
Neighbours: x_{i_n}, $i_n \in \{2, 4\}$
Best Neighbour: x_2
Neighbourhood (ring topology)
Valid area for x_3 at time t+1
x_0
x_1
x_2
x_3
x_4
x_5
x_6

FIGURE 11.1
Sample update of x_i, where $i = 3$ in a 2D space.

- x^t_{sd}: position of the *swarm*'s best individual in the d^{th} dimension at time step t and $s \in \{0, 1, 2, ..., N\text{-}1\}$
- $u \sim \mathrm{U}(0, 1)$: generated afresh for each dimension update.

The update equation is illustrated in an example in Fig. 11.1 where $\vec{x}_3$ is to be updated. The algorithm is characterised by two main components: a dynamic rule for updating the population's position (assisted by a social neighbouring network that informs this update), and communication of the results of the best found individual to others.

As stated earlier, the position of members of the swarm in DFO can be restarted, with one impact of such restarts being the displacement of the individuals which may lead to discovering better positions. To consider this eventuality, an element of stochasticity is introduced to the update process. Based on this, individual dimensions of the population's position vectors are reset if a random number generated from a uniform distribution on the unit interval $\mathrm{U}(0, 1)$ is less than the *restart threshold*, Δ. This guarantees a restart to the otherwise permanent stagnation over likely local minima.

Algorithm 10 summarises the DFO algorithm. In this algorithm, each member of the population is assumed to have two neighbours (i.e. ring topology).

Algorithm 10

```
1: procedure DFO ($N, D, \vec{x}_{\min}, \vec{x}_{\max}, f$)*
2:     for $i = 0 \to N-1$ do                    ▷ Initialisation: Go through each fly
3:         for $d = 0 \to D-1$ do                ▷ Initialisation: Go through each dimension
4:             $x^0_{id} \leftarrow \mathrm{U}(x_{\min,d}, x_{\max,d})$    ▷ Initialise $d^{\text{th}}$ dimension of fly $i$
5:         end for
6:     end for
7:     while ! termination criteria do            ▷ Main DFO loop
8:         for $i = 0 \to N-1$ do                ▷ Evaluation: Go through each fly
9:             $\vec{x}_i$.fitness $\leftarrow f(\vec{x}_i)$
```

```
10:         end for
11:         $\vec{x}_s = \arg\min [f(\vec{x}_i)], \quad i \in \{0, 1, 2, \ldots, N-1\}$     ▷ Find best fly
12:         for $i = 0 \to N-1$ and $i \neq s$ do     ▷ Update each fly except the best
13:             $\vec{x}_{i_n} = \arg\min [f(\vec{x}_{(i-1)\%N}), f(\vec{x}_{(i+1)\%N})]$     ▷ Find best neighbour
14:             for $d = 0 \to D-1$ do     ▷ Update each dimension
15:                 if $U(0,1) < \Delta$ then     ▷ Restart mechanism
16:                     $x_{id}^{t+1} \leftarrow U(x_{\min,d}, x_{\max,d})$     ▷ Restart within bounds
17:                 else
18:                     $u \leftarrow U(0,1)$
19:                     $x_{id}^{t+1} \leftarrow x_{i_n d}^{t} + u(x_{sd}^{t} - x_{id}^{t})$     ▷ Update the dimension value
20:                     if $x_{id}^{t+1} < x_{\min,d}$ or $x_{id}^{t+1} > x_{\max,d}$ then     ▷ Out of bounds
21:                         $x_{id}^{t+1} \leftarrow U(x_{\min,d}, x_{\max,d})$     ▷ Restart within bounds
22:                     end if
23:                 end if
24:             end for
25:         end for
26:     end while
27:     return $\vec{x}_s$
28: end procedure
```

* INPUT: swarm size, dimensions, lower/upper bounds, fitness function.

11.3 Source code

The source code in this section provides the standard implementation of DFO in three programming languages, Matlab (Listings 11.1 and 11.2), C++ (Listing 11.3) and Python (Listing 11.4). The fitness function used in the code is the Sphere function which is defined below in Eq. 11.4. The fitness function f takes a D-dimensional vector (i.e. one fly, or $\vec{x}_i$) and returns a single value (i.e. fitness value). DFO is tasked to find the optimal value for each parameter or dimension.

$$f(\vec{x}_i) = \sum_{d=1}^{D} x_{id}^2 \qquad \text{where } -5.12 \leqslant x_{id} \leqslant 5.12 \tag{11.4}$$

11.3.1 Matlab

Implementation of DFO in Matlab is provided below.

```
function [sum]=f(X)
  [N,D]=size(X);
 sum=zeros(N,1);
  for i=1:N
    for d=1:D
```

```
      sum(i,1)=sum(i,1)+X(i,d)^2;
    end
  end
end
```

Listing 11.1
Implementation of the fitness function (Sphere) in Matlab (see Eq. 11.4).

```
clear
% INITIALISE PARAMETERS (N: swarm size, D: dimensionality)
N=100; D = 30; delta = 0.001; maxIter=1000;
% LOWER AND UPPER BOUNDS
lowerB(1,1:D)=-5.12; upperB(1,1:D)=5.12;
s = 0;        % INITIAL INDEX OF BEST FLY
X = zeros(N,D); % FLIES MATRIX
fitness(1,1:N) = realmax; % FITNESS VALUES

for i=1:N
  for d=1:D
    X(i,d) = lowerB(d)+rand()*(upperB(d)-lowerB(d));
  end
end

for itr=1:maxIter
  % EVALUATE FITNESS OF THE FLIES AND FIND THE BEST
  fitness = f(X);
  [sFitness, s] = min(fitness);
  disp(['Iteration: ', num2str(itr),'    Best fly index: ', num2str(s
    ), '    Fitness value: ', num2str(sFitness), ] )

  % UPDATE EACH FLY INDIVIDUALLY
  for i=1:N
    % ELITIST STRATEGY (DON'T UPDATE BEST FLY)
    if i == s continue; end

    % FIND BEST NEIGHBOUR FOR EACH FLY
    left=mod(i-2,N)+1; right=mod(i,N)+1; % INDICES: LEFT & RIGHT FLIES
    if fitness(right)<fitness(left) bNeighbour = right;
    else bNeighbour = left; end

    for d=1:D
      % DISTURBANCE MECHANISM
      if rand()<delta X(i,d) = lowerB(d)+rand()*(upperB(d)-lowerB(d));
     continue; end

      % UPDATE EQUATION
      X(i,d) = X(bNeighbour,d) + rand()*( X(s,d)- X(i,d) );

      % OUT OF BOUND CONTROL
      if or( X(i,d) < lowerB(d), X(i,d) > upperB(d) )
        X(i,d) = lowerB(d)+rand()*(upperB(d)-lowerB(d));
      end
    end
  end
end

% EVALUATE FITNESS OF THE FLIES AND FIND THE BEST
fitness = f(X);
[sFitness, s] = min(fitness);

disp( ['Final best fitness=', num2str(sFitness)] )
```

Listing 11.2
Complete DFO code in Matlab.

11.3.2 C++

DFO implementation in C++ is presented here.

```cpp
#include <iostream>
#include <stdlib.h>
#include <cmath>
#include <ctime>
using namespace std;

// FITNESS FUNCTION (SPHERE FUNCTION)
float f(float x[], int D) { // x IS ONE FLY AND D IS DIMENSION
  float sum=0;
  for(int i=0; i<D; i++)
    sum=sum+x[i]*x[i];
  return sum;
}

// GENERATE RANDOM NUMBER IN RANGE [0, 1)
float r() {
  return (float) rand() / (RAND_MAX);
}

int main() {
  srand(time(NULL)); // TO GENERATE DIFFERENT RANDOM NUMBERS
  // INITIALISE PARAMETERS
  int N=100; int D = 30; float delta = 0.001; int maxIter=1000;
  float lowerB[D]; float upperB[D]; // LOWER AND UPPER BOUNDS
  int s = 0;       // INITIAL INDEX OF BEST FLY
  float X[N][D];     // FLIES VECTORS
  float fitness[N];    // FITNESS VALUES

  // SET LOWER AND UPPER BOUND CONSTRAINTS FOR EACH DIMENSION
  for (int d=0; d<D; d++) {
    lowerB[d]=-5.12; upperB[d]=5.12;
  }

  // INITIALISE FLIES WITHIN BOUNDS. MATRIX SIZE: (N,D)
  for(int i=0; i<N; i++)
    for(int d=0; d<D; d++)
      X[i][d] = lowerB[d] + r()*(upperB[d]-lowerB[d]);

  // MAIN DFO LOOP
  for (int itr=0; itr<maxIter; itr++) {
    for(int i=0; i<N; i++) { // EVALUATE EACH FLY AND FIND THE BEST
      fitness[i]=f(X[i],D);
      if (fitness[i] <= fitness[s]) s = i;
    }

    if ( itr%100 == 0) // PRINT RESULT EVERY 100 ITERATIONS
      cout << "Iteration: " << itr << "\t Best fly index: " << s
      << "\t Fitness value: " << fitness[s] << endl;

    // UPDATE EACH FLY INDIVIDUALLY
    for(int i=0; i<N; i++) {
      // ELITIST STRATEGY (i.e. DON'T UPDATE BEST FLY)
      if (i==s) continue;

      // FIND BEST NEIGHBOUR FOR EACH FLY
      int left; int right; int bNeighbour;
      left=(i-1)%N; right=(i+1)%N; // INDICES: LEFT & RIGHT FLIES
      if (fitness[right]<fitness[left]) bNeighbour = right;
      else bNeighbour = left;

      // UPDATE EACH DIMENSION SEPARATELY
      for (int d=0; d<D; d++) {
        if (r() < delta) { // DISTURBANCE MECHANISM
```

```
          X[i][d] = lowerB[d] + r()*(upperB[d]-lowerB[d]); continue;
        }

        // UPDATE EQUATION
        X[i][d] = X[bNeighbour][d] + r()*( X[s][d]- X[i][d] );

        // OUT OF BOUND CONTROL
        if (X[i][d] < lowerB[d] or X[i][d] > upperB[d])
          X[i][d] = (upperB[d]-lowerB[d])*r()+lowerB[d];
      }
    }
  }
  // EVALUATE EACH FLY'S FITNESS AND FIND BEST FLY
  for(int i=0; i<N; i++) {
    fitness[i]=f(X[i],D);
    if (fitness[i] < fitness[s]) s = i;
  }
  cout << "Final best fitness: " << fitness[s] << endl;
  return 0;
}
```

Listing 11.3
Complete DFO code in C++.

11.3.3 Python

This part provides the implementation of DFO in Python.

```
import numpy as np

# FITNESS FUNCTION (SPHERE FUNCTION)
def f(x): # x IS A VECTOR REPRESENTING ONE FLY
  sum = 0.0
  for i in range(len(x)):
    sum = sum + np.power(x[i],2)
  return sum

N = 100            # POPULATION SIZE
D = 30             # DIMENSIONALITY
delta = 0.001      # DISTURBANCE THRESHOLD
maxIterations = 1000   # ITERATIONS ALLOWED
lowerB = [-5.12]*D     # LOWER BOUND (IN ALL DIMENSIONS)
upperB = [ 5.12]*D     # UPPER BOUND (IN ALL DIMENSIONS)

# INITIALISATION PHASE
X = np.empty([N,D]) # EMPTY FLIES ARRAY OF SIZE: (N,D)
fitness = [None]*N  # EMPTY FITNESS ARRAY OF SIZE N

# INITIALISE FLIES WITHIN BOUNDS
for i in range(N):
  for d in range(D):
    X[i,d] = np.random.uniform(lowerB[d], upperB[d])

# MAIN DFO LOOP
for itr in range (maxIterations):
  for i in range(N): # EVALUATION
    fitness[i] = f(X[i,])
  s = np.argmin(fitness) # FIND BEST FLY

  if (itr%100 == 0): # PRINT BEST FLY EVERY 100 ITERATIONS
    print ("Iteration:", itr, "\tBest fly index:", s,
           "\tFitness value:", fitness[s])

  # TAKE EACH FLY INDIVIDUALLY
  for i in range(N):
    if i == s: continue # ELITIST STRATEGY

    # FIND BEST NEIGHBOUR
    left = (i-1)%N
    right = (i+1)%N
    bNeighbour = right if fitness[right]<fitness[left] else left

    for d in range(D): # UPDATE EACH DIMENSION SEPARATELY
      if (np.random.rand() < delta):
        X[i,d] = np.random.uniform(lowerB[d], upperB[d])
        continue;

      u = np.random.rand()
      X[i,d] = X[bNeighbour,d] + u*(X[s,d] - X[i,d])

      # OUT OF BOUND CONTROL
      if X[i,d] < lowerB[d] or X[i,d] > upperB[d]:
        X[i,d] = np.random.uniform(lowerB[d], upperB[d])

for i in range(N): fitness[i] = f(X[i,]) # EVALUATION
s = np.argmin(fitness) # FIND BEST FLY

print("\nFinal best fitness:\t", fitness[s])
print("\nBest fly position:\n", X[s,])
```

Listing 11.4
Complete DFO code in Python.

A sample output of the code is shown below, where the index and fitness value of the best fly in every 100 iterations are displayed. At the end, the best solution, containing the best parameters found, is also shown.

```
Iteration: 0    Best fly index: 60  Fitness value: 150.5914836416227
Iteration: 100  Best fly index: 68  Fitness value: 0.011859096868779489
Iteration: 200  Best fly index: 53  Fitness value: 2.2510252307575077e-05
Iteration: 300  Best fly index: 90  Fitness value: 6.779722749003586e-08
Iteration: 400  Best fly index: 24  Fitness value: 3.163652073209633e-10
Iteration: 500  Best fly index: 49  Fitness value: 4.503385943119482e-11
Iteration: 600  Best fly index: 28  Fitness value: 1.1633233211569493e-13
Iteration: 700  Best fly index: 20  Fitness value: 2.7396155007581633e-15
Iteration: 800  Best fly index: 15  Fitness value: 2.3009156731064727e-17
Iteration: 900  Best fly index: 10  Fitness value: 4.744041469725476e-19

Final best fitness:  1.214236579001144e-20

Best fly position:
[-3.75694837e-12 -5.51560681e-13  5.40769833e-12  6.60606978e-11
 -4.01610956e-13 -2.74643196e-12 -9.51829640e-12  7.06787789e-12
  2.19263924e-11  1.00330905e-11  3.99326555e-12 -4.08853263e-12
  1.56206199e-11  9.80355426e-12  1.12747027e-11  6.40617484e-12
 -1.61567359e-11 -3.34267744e-11  2.88367151e-11 -4.66779500e-12
 -6.35363397e-12  3.79648533e-12 -2.97971162e-12 -1.58178299e-12
 -5.42664323e-11  7.37563208e-12 -2.74749207e-11  8.82333052e-12
 -1.54070116e-11  9.84007230e-12]
```

Listing 11.5
Sample output: optimising Sphere function using DFO.

11.4 Numerical example: optimisation with DFO

In order to understand the swarming behaviour of the flies in DFO, a step by step example is presented in this section. These steps illustrate how the swarm moves towards the optimal solution. In this example, the number of flies is set to 5 ($N = 5$), the number of dimensions or parameters is set to 2 ($D = 2$) and DFO is run for 5 iterations. The rest of the configuration is the same.

Initially, each fly's position vector (in this example containing only two dimensions) is initialised within the range (in this case, between -5.12 and 5.12). This process is shown in Table 11.1 (lines 0-4). Next, the fitness of each fly is calculated to quantify how "good" the solution provided by each fly is. Then, based on the fitness values (calculated using Eq. 11.4), each fly's best neighbour is selected; the standard DFO uses *ring topology*, which means each fly has a left and a right neighbour, which are based on indices (e.g. a fly with index 3, has two neighbours: fly 2 is the left neighbour and fly 4 is the right neighbour). Once these steps are taken, the update equation (Eq. 11.3) is used to update the value of each dimension in each fly. Prior to updating any of the dimensions, a random number between 0 and 1 is generated and if the randomly generated number is less than the disturbance threshold, Δ[1], a random value between the lower and upper bounds is generated for that dimension, otherwise the update equation is used[2]. Once the new value for each dimension is calculated, it is checked against the constraint, ensuring it fits within the bounds; if the value is outside the valid region, a random number is generated between the lower and upper bounds.

In Table 11.1, the index of each fly is shown on the leftmost column, then the value of the first dimension, $d = 0$, followed by the value of the second dimension, $d = 2$, and the fitness value which is calculated using the values of the first and second dimensions; for instance to calculate the fitness of the first fly, fly 0 or $\vec{x}_0$ in line 0, the following can be (using Eq. 11.4):

$$f(\vec{x}_0) = 2.280657^2 + -4.809789^2 = 28.335468$$

The last column in Table 11.1 represents the index of the best neighbouring fly. The index of the best fly in the population, s, is highlighted in the same column. Using the elitist approach, the best fly in the swarm is kept intact and therefore is not updated, neither by the disturbance mechanism, nor by the update equation. This makes sure that the population does not lose the best solution it has found so far.

For instance, in order to update the first dimension of the first fly, x_{00} (line 0, where in x_{00}, the first 0 refers to the fly index, and the second 0 refers to

[1]Note that the value of Δ is problem dependent.

[2]In this example, none of the random numbers generated was less than Δ.

TABLE 11.1
Running DFO for 5 iterations.

L	**Fly Index**	**d=0**	**d=1**	**Fitness value**	**Best neighbour**
0	**0**	2.2806566	-4.80978936	28.3354681843	4
1	**1**	-2.13229778	4.71940929	26.8195179113	2
2	**2**	1.25796613	4.70359891	23.7063214781	3
3	**3**	-2.43396026	-0.90766546	6.7480191207	4
4	**4**	0.06228462	1.33995725	1.7993648107	s = 4
5	**Itr = 1**	**d=0**	**d=1**	**Fitness value**	**Best neighbour**
6	**0**	-0.91245846	2.61537071	7.6727443985	4
7	**1**	2.13769454	4.19262811	22.1478683732	0
8	**2**	-2.84773347	-4.11611456	25.0519850402	3
9	**3**	0.6558135	3.10292087	10.058209244	4
10	**4**	0.06228462	1.33995725	1.7993648107	s = 4
11	**Itr = 2**	**d=0**	**d=1**	**Fitness value**	**Best neighbour**
12	**0**	1.01468105	0.53914852	1.3202587652	1
13	**1**	0.34869085	0.20424149	0.1632998901	s = 1
14	**2**	3.05708686	2.96239515 *	18.1215650971	1
15	**3**	-0.02827646	0.42791449	0.1839103648	4
16	**4**	0.06228462	1.33995725	1.7993648107	3
17	**Itr = 3**	**d=0**	**d=1**	**Fitness value**	**Best neighbour**
18	**0**	0.14988694	-0.0675497	0.0270290553	4
19	**1**	0.34869085	0.20424149	0.1632998901	2
20	**2**	-0.04768011	-0.11545245	0.0156026608	s = 2
21	**3**	0.33950867	1.17054436	1.4854402283	2
22	**4**	0.38255993	0.04292476	0.1481946319	0
23	**Itr = 4**	**d=0**	**d=1**	**Fitness value**	**Best neighbour**
24	**0**	0.20121156	0.00658197	0.0405294143	4
25	**1**	-0.13562958	-0.24398233	0.0779227572	2
26	**2**	-0.04768011	-0.11545245	0.0156026608	1
27	**3**	-0.38006057	-0.27793257	0.2216925515	4
28	**4**	-0.01290562	-0.08049597	0.0066461562	s = 4
29	**Itr = 5**	**d=0**	**d=1**	**Fitness value**	**Best neighbour**
30	**0**	-0.20867294	-0.13664574	0.0622164545	1
31	**1**	0.01618933	-0.02789224	0.0010400715	s = 1
32	**2**	0.0404749	-0.00654758	0.0016810879	1
33	**3**	0.153082	0.10787423	0.0350709497	2
34	**4**	-0.01290562	-0.08049597	0.0066461562	3

the dimension number), the update equation is used:

$$\begin{aligned} x_{00} &= x_{40} + u(x_{40} - x_{00}) \\ &= 0.06228462 + 0.4394\ (0.06228462 - 2.2806566) \\ &= -0.91245846 \end{aligned}$$

where $u = 0.4394$ is a random number between 0 and 1. Note that in this instance, the best neighbouring fly and the best fly in the swarm are identical ($\vec{x}_4$). The same process can be repeated for x_{01}:

$$\begin{aligned} x_{01} &= x_{41} + u(x_{41} - x_{01}) \\ &= 1.33995725 + u(1.33995725 - (-4.80978936)) \\ &= 2.61537071 \end{aligned}$$

The updated values can be seen in Table 11.1, line 6; looking at the fitness value, it is evident that the fly has a better fitness value after the update (i.e. from 28.34 to 7.67). Next, let's update both dimensions of the second fly $\vec{x}_1$ whose values can be seen in line 1:

$$\begin{aligned} x_{10} &= x_{20} + u(x_{40} - x_{10}) \\ &= 1.25796613 + u(0.06228462 - (-2.13229778)) \\ &= 2.13769454 \\ x_{11} &= x_{21} + u(x_{41} - x_{11}) \\ &= 4.70359891 + u(1.33995725 - 4.71940929) \\ &= 4.19262811 \end{aligned}$$

The same process can be repeated for the rest of the flies. Note that in iteration 2, the second dimension of the third fly, x_{21} in line 14, which is highlighted with an asterisk (*), had been out of bounds and therefore the algorithm generated a random value within the allowed bounds for that dimension.

The optimisation process from the initialisation of the flies and throughout each iteration is illustrated in Fig. 11.2, showing the population's convergence towards the known optimal solution, i.e. (0,0).

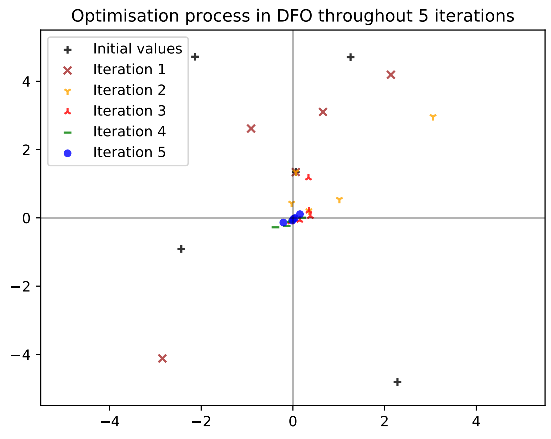

FIGURE 11.2
Illustrating the optimisation process in the example. The figure shows that after each iteration, the population is getting closer and closer ("converging") to the optimum point $(0, 0)$.

11.5 Conclusion

This paper provides an introduction to the main principles of the standard DFO algorithm. The high-level description of the algorithm is first provided, covering the main aspects of the algorithm and its update equation. Then an annotated pseudocode is presented, illustrating the detailed order of the steps. This is then complemented by providing a complete code of the DFO algorithm in Matlab, C++ and Python. The paper is finalised by presenting a step by step numerical example of how the algorithm guides the population to perform the optimisation. It is now possible to simply change the fitness function to that of another problem, specify the dimensionality of the problem (the number of the parameters to be optimised), the population size and disturbance threshold, and then run the code.

References

1. M.M. al-Rifaie. "Dispersive Flies Optimisation" in *Proc. of the 2014 Federated Conference on Computer Science and Information Systems*, vol. 2, pp. 529-538, 2014.
2. J. Kennedy. "The particle swarm: social adaptation of knowledge" in *Proc. of IEEE International Conference on Evolutionary Computation*, pp. 303-308, 1997.
3. R. Storn, K. Price. "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces". Journal of Global Optimization, vol. 11(4), pp. 341-359, 1997.
4. M. Dorigo, G.D. Caro, L.M. Gambardella. "Ant algorithms for discrete optimization". Artificial Life, vol. 5(2), pp. 137-172, 1999.
5. M.M. al-Rifaie. "Perceived simplicity and complexity in nature" in Proc. of AISB Annual Convention - Symposium VIII on Computational Architectures for Animal Cognition, pp. 299-305, Bath, United Kingdom, 2017.
6. M.M. al-Rifaie, A. Aber. "Dispersive Flies Optimisation and medical imaging" in Recent Advances in Computational Optimization, pp. 183-203, 2016.
7. H. Alhakbani. "Handling Class Imbalance Using Swarm Intelligence Techniques, Hybrid Data and Algorithmic Level Solutions". PhD Thesis, Goldsmiths, University of London, London, United Kingdom, 2018.

8. H.A. Alhakbani, M.M. al-Rifaie. “Optimising SVM to classify imbalanced data using dispersive flies” in *Proc. of the 2017 Federated Conference on Computer Science and Information Systems, FedCSIS 2017*, pp. 399-402, 2017.
9. H. Oroojeni, M.M. al-Rifaie, M.A. Nicolaou. “Deep neuroevolution: Training deep neural networks for false alarm detection in intensive care units” in *Proc. of European Association for Signal Processing (EUSIPCO) 2018*, pp. 1157-1161, 2018.
10. M.M. al-Rifaie, A. Ursyn, R. Zimmer, M.A.J. Javid. “On symmetry, aesthetics and quantifying symmetrical complexity” in *Proc. of International Conference on Evolutionary and Biologically Inspired Music and Art*, pp. 17-32, 2017.
11. M.M. al-Rifaie F.F. Leymarie, W. Latham, M. Bishop. “Swarmic autopoiesis and computational creativity”. Connection Science, vol. 29(4), pp. 276-294, 2017.
12. P. Aparajeya, F.F Leymarie, M.M. al-Rifaie. “Swarm-Based Identification of Animation Key Points from 2D-medialness Maps” in Proc. of EvoMUSART - Computational Intelligence in Music, Sound, Art and Design, Lecture Notes in Computer Science, vol. 11453, Springer, pp. 69-83, 2019.