IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

February 14, 2017

1 GLOBAL OPTIMIZATION (PARTICLE-SWARM ALGORITHM)

Particle swarm optimization (PSO) is a technique that uses swarm intelligence to find a global optimum of a user-defined multi-variate function. Swarm intelligence systems consist of a number of agents that communicate with other agents in the swarm, exchanging information about the local and neighboring environment. By this exchange of information, the swarm moves through a multi-variate function space converging towards a global optimum.
To cite the inventors of PSO, the "particle swarm algorithm imitates human (or insects) social behavior. Individuals interact with one another while learning from their own experience, and gradually the population members move into better regions of the problem space".
Underlying a particle swarm optimization is a **fitness function** that for each particle can be easily evaluated and measures the "degree of success" of our optimization. PSO is based on particles that move according to a particle velocity associated with each particle. The computation of the particle velocity has three component (see Figure 1.1): (i) an **inertial** component, (ii) a **cognitive** component, and (iii) a **social** component. The inertial component maintains a fraction of the current velocity vector, by continuing along the same direction as during the previous step. The cognitive component is based on the difference between the current position and the particle's best position (in terms of the fitness function) during its evolution. This component accounts for memory effects of the optimization process as witnessed by each single agent. Finally, the social component of the new velocity vector is proportional to the difference of the current agent position and the current best position of the entire swarm. This final component accounts for the communication of the best ob-

tained results (up to this point) to all agents, inducing a kind of flocking towards the global optimum.

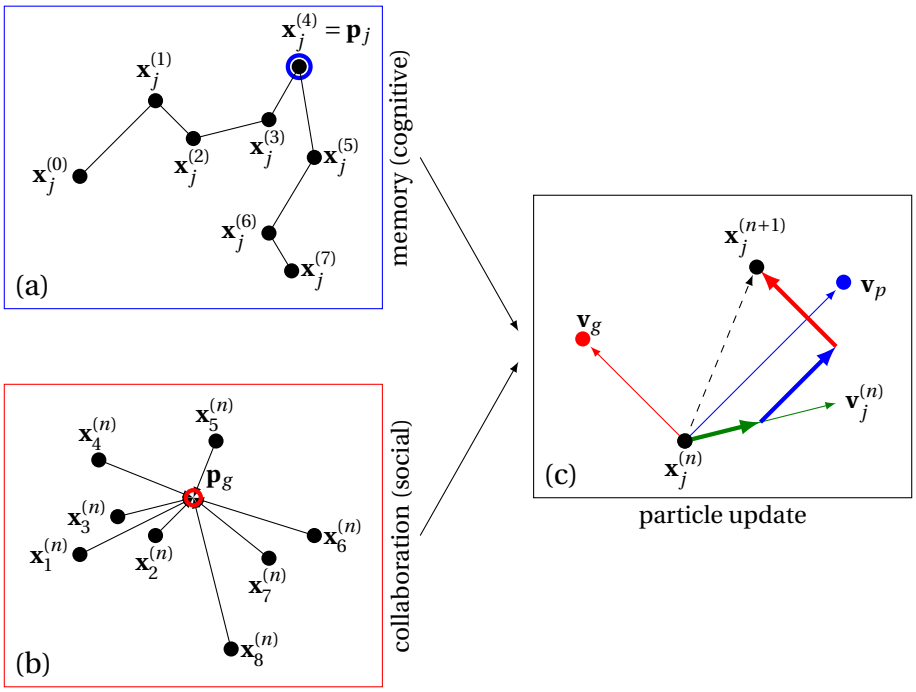

Figure 1.1: Principles of particle swarm optimization, combining a cognitive (a) and social (b) component. For the cognitive component, the best position for each particle j throughout the iterations is determined and stored as $\mathbf{p}_j$. For the social component, the best position throughout the iterations and over the entire swarm is determined and stored as $\mathbf{p}_g$. The velocity vector for each particle's update (c) involves a fraction of the previous velocity vector (in green), a random portion of the velocity $\mathbf{v}_p$ towards the particle's best position (in blue) and a random portion of the velocity $\mathbf{v}_g$ towards the globally best position (in red). The combined velocity vector determines the new position $\mathbf{x}_j^{(n+1)}$ of the j-th particle.

The final expression for the velocity update of each particle is then given as

$$\mathbf{v}_j^{(n+1)} = \omega \mathbf{v}_j^{(n)} + \mathbf{R}_1 \otimes \left(\mathbf{p}_j - \mathbf{x}_j^{(n)}\right) + \mathbf{R}_2 \otimes \left(\mathbf{p}_g - \mathbf{x}_j^{(n)}\right) \tag{1.1a}$$

$$\mathbf{x}_j^{(n+1)} = \mathbf{x}_j^{(n)} + \mathbf{v}_j^{(n+1)} \tag{1.1b}$$

with $\mathbf{x}_j$ denoting the position of the j-th particle and $\mathbf{v}_j$ its velocity. The superscript $^{(n)}$ stands for the iteration counter. The vector $\mathbf{p}_j$ stores the personal best of the j-th particle; the vector $\mathbf{p}_g$ stands for the global optimum across the entire swarm. The symbol $\otimes$ represents elementwise multiplication. The coefficient ω measures the inertial effects and regulates how much the new velocity follows the old one. The two coefficient vectors $\mathbf{R}_1$

and $\mathbf{R}_2$ consist of a deterministic scalar component and a random vector component. The deterministic part allows the user to weight the various components of the velocity update formula, e.g., how strongly we should follow a current global optimum. The random part (chosen as a uniformly distributed random number between zero and one) introduces a probabilistic component to the optimization algorithm that allows the exploration of other parts of the problem space and helps avoid the convergence to local rather than global optima.
The computation of the velocity vector for the new, updated particle position balances the concept of solution-space exploration and solution-space exploitation. The exploration part is induced by the random coefficients (as well as the coefficients) and allows the probing of solutions away from the current optima. The exploitation part is accomplished by the communication of the global optimum to all particles, which yields a general tendency of the particles to follow a "leading" particle (with the currently best solution). The design of an effective weighing strategy for exploration and/or exploitation is a challenging task and common to many optimization algorithms.
The above algorithm in its most primitive form can be used to find optima of complex, high-dimensional functions that are unconstrained by additional conditions. However, in most practical cases, we have to deal with side constraints either on the solution space itself or by enforcing supplementary conditions. Constraints can be in the form of equalities (equaitons) or inequalities (bounds).

1.1 Accounting for constraints

The incorporation of side constraints for our optimization problem requires additional modifications to the general particle swarm algorithm.
We formulate a general optimization problem as

$$f(x_1, x_2, \ldots, x_n) \longrightarrow \text{opt} \tag{1.2a}$$
$$g_j(x_1, x_2, \ldots, x_n) = 0 \qquad j = 1, \ldots, J \tag{1.2b}$$
$$h_k(x_1, x_2, \ldots, x_n) \leq 0 \qquad k = 1, \ldots, K \tag{1.2c}$$
$$l_i \leq x_i \leq h_i \qquad i = 1, \ldots, n \tag{1.2d}$$

where $x_1, \ldots, x_n$ denotes our n-dimensional solution vector, and f is the cost function we wish to optimize. The functions g_j impose J equality constraints that the solution has to satisfy to be feasible. Similarly, the functions h_k impose K inequality constraints. Finally, the last condition restricts the solution vector to given intervals in its various components.

The simplest constraints are on the solution itself, i.e., equation (1.2d). They are enforced by monitoring the solution after each update and **clipping** the components of the solution vector back to the permissible interval. Due to inertial part of the update formula, it is reasonable to expect that the same clipped particle will leave the interval again in the next iteration. For this reason, the velocity component corresponding to the clipped solution component is also changed in sign. The technique is referred to as **velocity mirroring**.

We also enforce limits on the velocities, after each iteration, to avoid excessive speeds across the solution space, but instead encourage a more careful exploitation of the solution space.

Besides the ranges for the solution, additional constraints come into two forms: equality and inequality constraints, and there are several techniques how to handle them in the algorithm. For inequality constraints, the most common ones are (i) feasibility checks (which we will use), and (2) adding the constraints to the cost function via penalty terms. In the first technique, we simply check whether a potential new solution $\mathbf{x}_j^{(n+1)}$ satisfies all inequality constraints; only then will it be accepted as a legitimate solution. Otherwise it will be rejected. For simple (and liberal) constraints and for rather low-dimensional problems, this technique works quite well; for higher-dimensional problems or very stringent constraints, additional modifications to the algorithm are necessary. The second strategy adds the constraints to the cost function via penalty terms: in this manner, not satisfying the constraints causes additional cost and is thus discouraged. Again, difficulties arise for high-dimensional and highly restricted optimization problems and special care has to be exercised.
Equality constraints are more difficult to deal with in the particle swarm algorithm. Two general lines are followed in the PSO-community. A nonlinear Newton-Raphson solver for the equality constraint is formulated, and an approximate Jacobian is formulated based on the current particle position, which is then used to advance the particles while observing the equality constraints. A second technique approaches the problem by formulating a repair function that projects the non-feasible solution back onto a feasible one. This projection is applied after each iteration.

1.2 Implementation

```
1  import numpy as np
2  import scipy as sp
3  import math as ma
4
5  def sphere(x):
6      # spherical fitness functional
7      c = ((x-1)**2).sum(axis=0)
8      return c
9
10 def rosenbrock2(x):
11     # 2D Rosenbrock fitness functional
12     xx = x[0,]
13     yy = x[1,]
14     a  = 2.
15     c  = (a-xx)**2 + 100.*(yy-xx**2)**2
```

```
    return c

def rosenbrock3(x):
    # 3D Rosenbrock fitness functional
    xx = x[0,]
    yy = x[1,]
    zz = x[2,]
    c  = (1.-xx)**2 + (1.-yy)**2 + \
         100.*(yy-xx**2)**2 + 100.*(zz-yy**2)**2
    return c

def rastrigin2(x):
    # 2D Rastrigin fitness functional
    xx = x[0,]
    yy = x[1,]
    a  = 10.
    c  = 2*a + xx**2 - a*np.cos(2*ma.pi*xx) + \
         yy**2 - a*np.cos(2*ma.pi*yy)
    return c

def rastrigin3(x):
    # 3D Rastrigin fitness functional
    xx = x[0,]
    yy = x[1,]
    zz = x[2,]
    a = 10.
    c  = 3*a + xx**2 + yy**2 + zz**2 - \
         a*np.cos(2*ma.pi*xx) - a*np.cos(2*ma.pi*yy) - \
         a*np.cos(2*ma.pi*zz)
    return c
```

```
if __name__ == '__main__':

    #------ particle swarm optimization (PSO)

    # number of variables and swarm size
    nvar   = 2
    nswarm = 30

    # bounds on the variable x
    xMin = -5
    xMax =  5
```

```
12
13      # bounds on the velocity
14      vMax = 0.1*(xMax-xMin)
15      vMin = -vMax
16
17      # maximum number of iterations
18      itMax = 5000
19      # inertia and inertia damping
20      w = 1
21      wdamp = 0.99
22      # acceleration coefficients
23      c1 = 2
24      c2 = 2
25
26      # initialization
27      x     = np.random.uniform(xMin,xMax,(nvar,nswarm))
28      v     = np.zeros((nvar,nswarm),float)
29      c     = rosenbrock2(x)
30      p     = x.copy()
31      g, ig = c.min(0),c.argmin(0)
32      gx    = x[:,ig]
33
34      # PSO main loop
35      for i in range(1,itMax+1):
36
37          # inertia damping
38          w = w*wdamp
39
40          for j in range(0,nswarm):
41
42              # random weights for cognitive and social behavior
43              Rcog = np.random.uniform(0,1,(1,nvar))
44              Rsoc = np.random.uniform(0,1,(1,nvar))
45
46              # update velocity (inertia + cognitive + social)
47              v[:,j] = w*v[:,j] + c1*Rcog*(p[:,j] - x[:,j]) + \
48                       c2*Rsoc*(gx - x[:,j])
49
50              # apply velocity limits
51              v[:,j] = np.clip(v[:,j],vMin,vMax)
52
53              # update position
54              x[:,j] += v[:,j]
55
```

```
56              # velocity mirroring
57              imirr      = ( (x[:,j] < xMin).nonzero() or \
58                             (x[:,j] > xMax).nonzero() )
59              v[imirr,j] = -v[imirr,j]
60
61              # apply variable limits
62              x[:,j] = np.clip(x[:,j],xMin,xMax)
63
64              # evaluation of best cost
65              cc  = rosenbrock2(x[:,j])
66              ccp = rosenbrock2(p[:,j])
67
68              # update particle-best and globally best solution
69              if cc < ccp:
70                  # update particle-best solution
71                  p[:,j] = (x[:,j]).copy()
72                  if cc < g:
73                      # update globally best solution
74                      g  = cc.copy()
75                      gx = (x[:,j]).copy()
76                      print(g)
```

1.3 Extension to multi-objective particle swarm optimization (MO-PSO)

In many applications of optimization, we have to deal with multiple objectives that have to be satisfied simultaneously. In most cases, these objective are conflicting, and an improvement along one objective can deteriorate the optimal solution when measure in another objective. The task then is to compute an optimal solution that satisfies all constraints and optimally balances the multiple objectives. Problems of this type are referred to as multi-objective optimization problems. A classical example is the optimization of a portfolio of financial instruments: we commonly cannot simultaneously maximize the return of the portfolio and minimize its volatility. A superior portfolio return usually induces a certain amount of volatility.

For multi-objective optimization, we have to introduce the idea of **domination**: we say that a solution vector **a** dominates over a solution vector **b**, if **a** is no worse than **b** in all given objectives (fitness functions), and if **a** is strictly superior than **b** in at least one of the multiple objectives (fitness functions).

We trace each solution in a higher-dimensional objective space, where each coordinate corresponds to the fitness of the solution regarding the respective coordinate. For the above example of portfolio optimization, we plot the return and the volatility of each solution in a two-dimensional plane. As the particle swarm moves through problem space and as the above dominant concept is applied, we converge towards a front in this multi-variate

fitness/objective space: this front is known as the **Pareto front**.