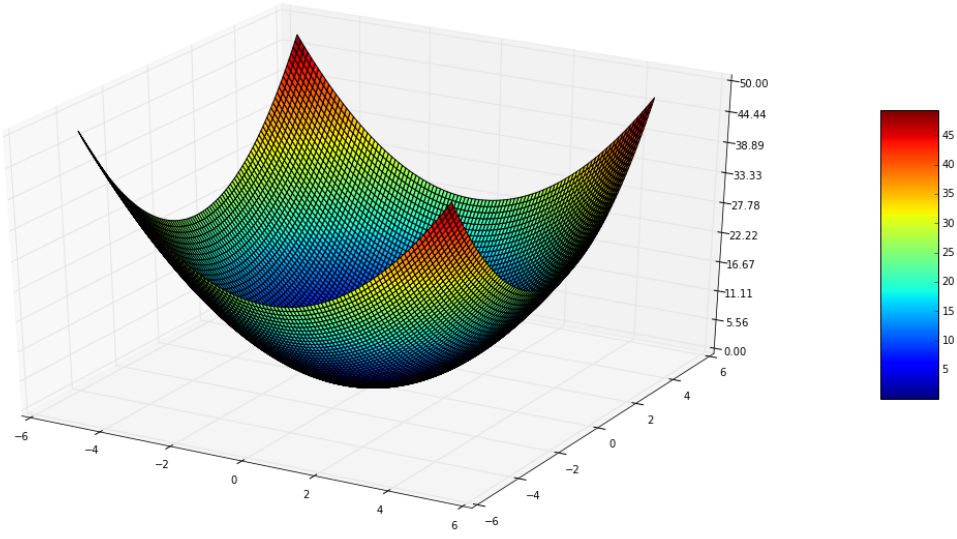
Differential Evolution Tutorial and Using Differential Evolution Implementation in Scipy

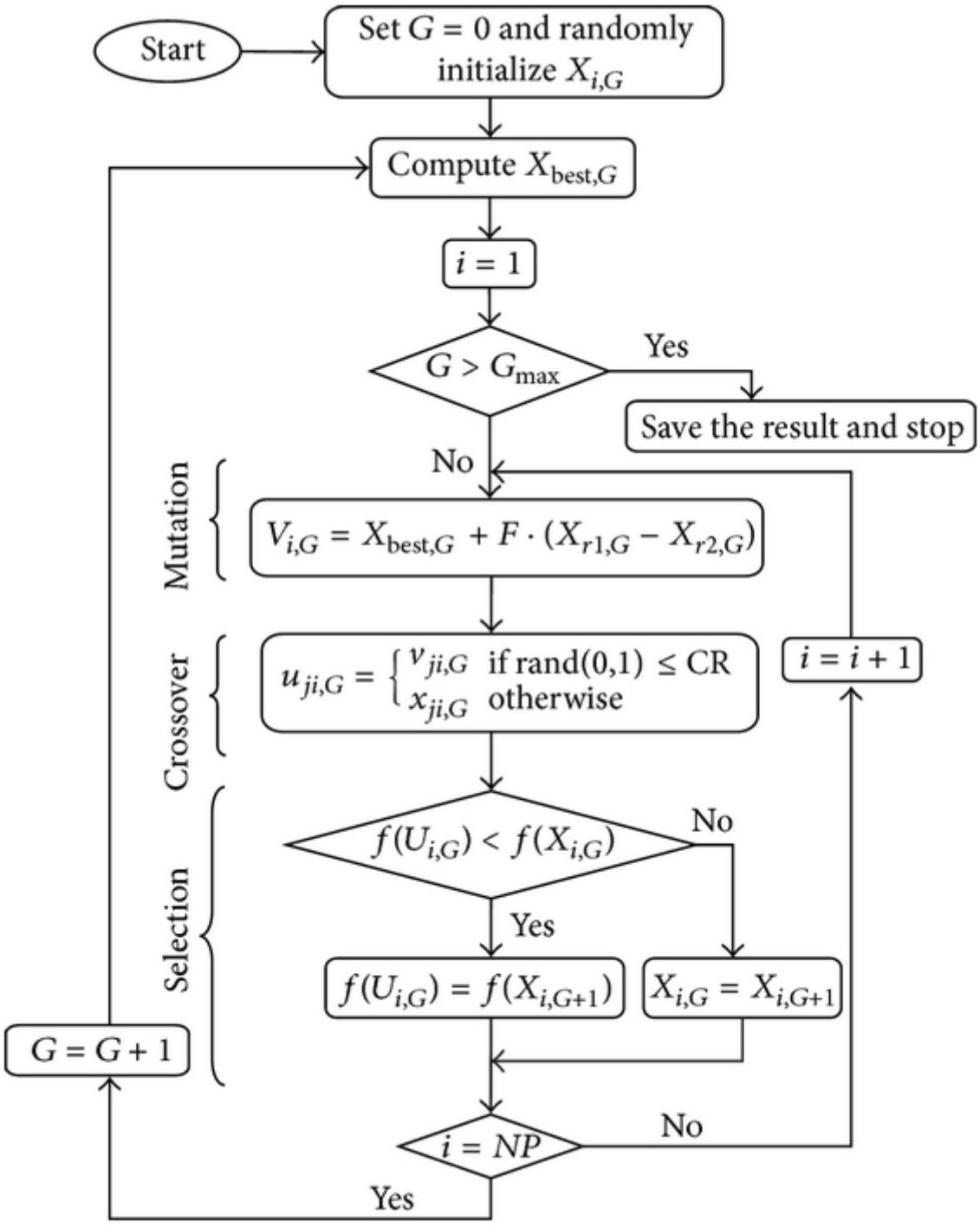
Optimization, Python, Scipy



This text examines the inner workings of differential evolution algorithm and its implementation in Scipy. It will be interesting also to evaluate its efficiency with

different conditions, such as with functions with multiple local minima and multiple variables.

Differential Evolution is a type of evolutionary algorithm, proposed first by Storn and Price [1] . The main differentiator of differential evolution is the use of arithmetic operators, whereas conventional genetic algorithms use binary crossover and mutation operators.



Main stages of differential evolution algorithm [2]:

1. *Initialize* the population:
2. *Mutation*: Select a target vector and create a trial mutant.
3. *Cross-over*: Cross-over rate determines which parameter value would be taken from which parent (target or mutant).
4. *Selection*: Compare the objective function values for child and parent. Whichever has the lower value gets selected.
5. *Termination*: Continue until a termination criteria is satisfied.

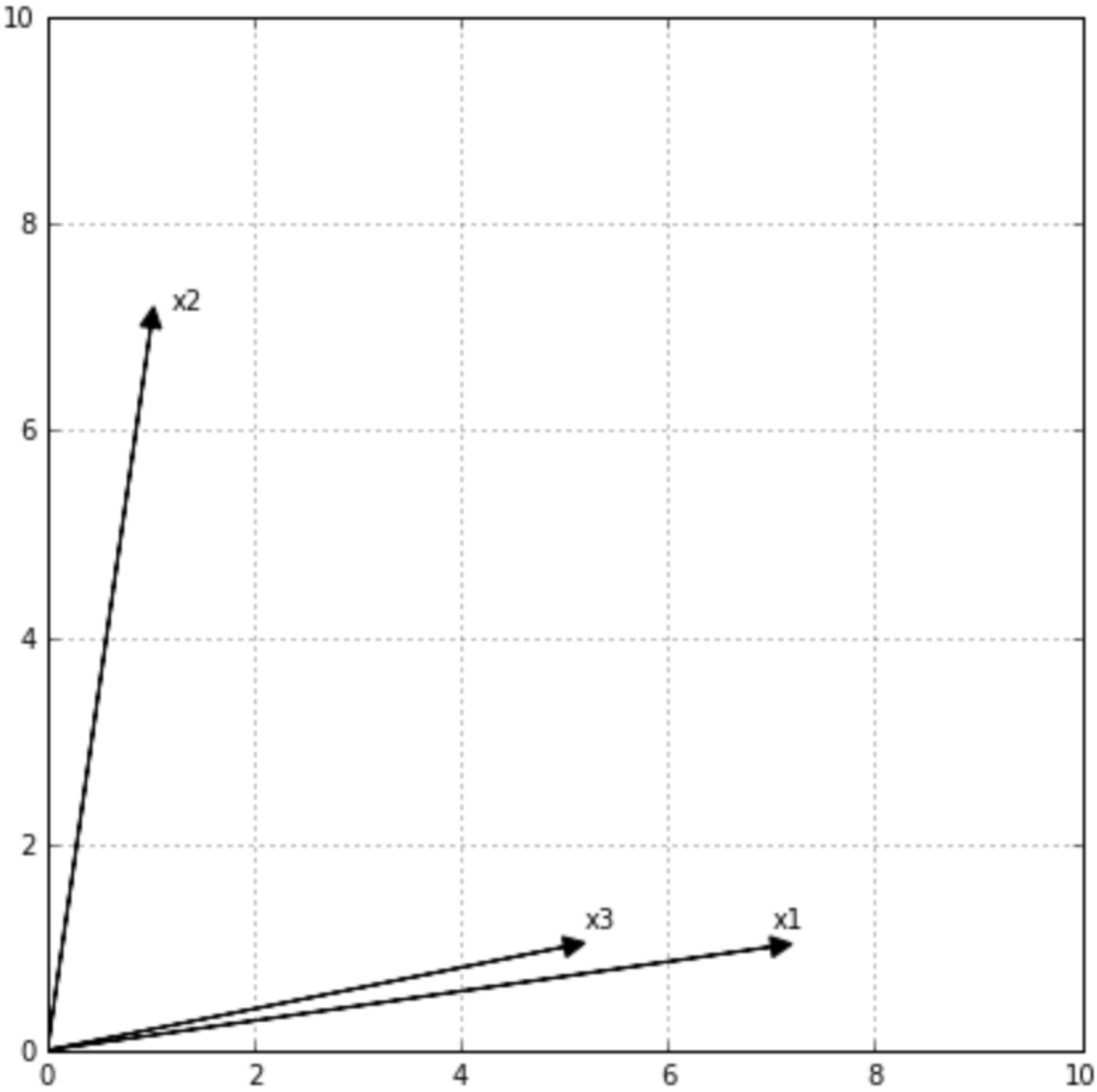
The python modules we shall use in this analysis.

```
%matplotlib inline
from __future__ import division, print_function, absolute_import
from numpy import *
import numpy as np
from datetime import datetime
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from matplotlib import cm

figsize = (7, 7)
```

Differential Evolution Stages

1. Initialize the population

The size of the population is specified by the user. The population size is kept fixed during an optimization run. The population members are real-valued vectors with dimension D which is the number of objective function parameters.
The objective function we try minimize is a 2D Rosenbrock function.



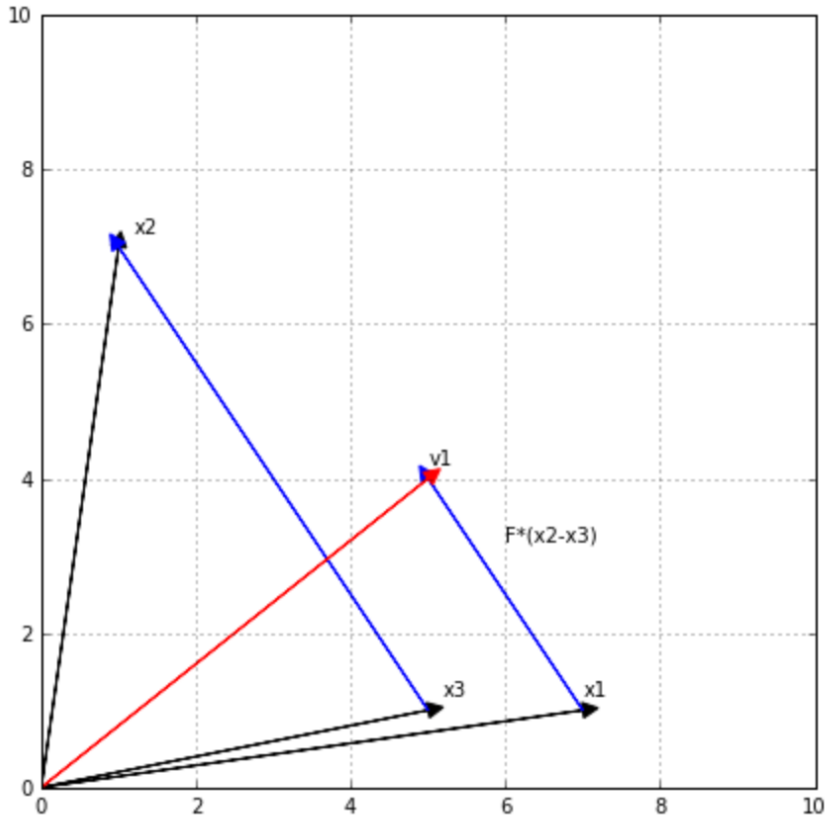

```
rosenbrock(x1): 230436.0
rosenbrock(x2): 3600.0
rosenbrock(x3): 57616.0
```

```
def rosenbrock(X):
    x = X[0]
    y = X[1]
    a = 1. - x
    b = y - x*x
    return a*a + b*b*100.

fig = plt.figure(figsize = figsize)
ax = fig.add_subplot(111)
ax.arrow(0, 0, 7, 1, head_width=0.2, head_length=0.2, fc='k', ec='k')
ax.arrow(0, 0, 1, 7, head_width=0.2, head_length=0.2, fc='k', ec='k')
ax.arrow(0, 0, 5, 1, head_width=0.2, head_length=0.2, fc='k', ec='k')

ax.annotate('x1', xy=(7, 1), xytext=(7, 1.2))
ax.annotate('x2', xy=(1, 7), xytext=(1.2, 7.2))
ax.annotate('x3', xy=(5, 1), xytext=(5.2, 1.2))

plt.xlim(0, 10)
plt.ylim(0, 10)
plt.grid()
plt.show()

```

```
x1=np.array([7.,1.])
print("rosenbrock(x1):",rosenbrock(x1))
x2=np.array([1.,7.])
print("rosenbrock(x2):",rosenbrock(x2))
x3=np.array([5.,1.])
print("rosenbrock(x3):",rosenbrock(x3))
```

2. Mutation (First Evolutionary Operator)

Select a target vector and create a trial mutant:
v1= x1 + F*(x2-x3),
F is the mutation constant determined by the user. The optimum value of F is in the range [0,2]. From [3]: “It has been found recently that selecting F from the interval [0.5, 1.0] randomly for each generation or for each difference vector, a technique called dither, improves convergence behaviour significantly, especially for noisy objective functions.”




```
rosenbrock(x1): 230436.0
rosenbrock(x2): 3600.0
rosenbrock(x3): 57616.0
Weighted difference multiplied by the mutation constant F is: [-2.  3.]
Trial mutant vector v1 : [ 5.  4.]
```

```
fig = plt.figure(figsize = figsize)
ax = fig.add_subplot(111)
ax.arrow(0, 0, 7, 1, head_width=0.2, head_length=0.2, fc='k', ec='k')
ax.arrow(0, 0, 1, 7, head_width=0.2, head_length=0.2, fc='k', ec='k')
ax.arrow(0, 0, 5, 1, head_width=0.2, head_length=0.2, fc='k', ec='k')

ax.arrow(5, 1, -4, 6, head_width=0.2, head_length=0.2, fc='b', ec='b')
ax.arrow(7, 1, -2, 3, head_width=0.2, head_length=0.2, fc='b', ec='b')
ax.arrow(0, 0, 5, 4, head_width=0.2, head_length=0.2, fc='r', ec='r')


ax.annotate('x1', xy=(7, 1), xytext=(7, 1.2))
ax.annotate('x2', xy=(1, 7), xytext=(1.2, 7.2))
ax.annotate('x3', xy=(5, 1), xytext=(5.2, 1.2))
ax.annotate('v1', xy=(5, 4), xytext=(5, 4.2))
ax.annotate('F*(x2-x3)', xy=(5, 4), xytext=(6, 3.2))


plt.xlim(0, 10)
plt.ylim(0, 10)
plt.grid()
plt.show()

x1=np.array([7.,1.])
print("rosenbrock(x1):",rosenbrock(x1))
x2=np.array([1.,7.])
print("rosenbrock(x2):",rosenbrock(x2))
x3=np.array([5.,1.])
print("rosenbrock(x3):",rosenbrock(x3))

F = 0.5
wd = F*(x2-x3)
print("Weighted difference multiplied by the mutation constant F is:", wd)
v1  = x1 + F*(x2-x3)
print("Trial mutant vector v1 :", v1)
```

3. Recombination (Second Evolutionary Operator)

In recombination stage, a trial vector ui is built by determining for every vector component, if the corresponding component should be copied from the target vector x1 or the mutated vector v1. The decision is based on a random variable randj that is compared to the user-defined control parameter CR.

Also from [3] "…setting CR to a low value, e.g. CR=0.2 helps optimizing separable functions since it fosters the search along the coordinate axes. On the contrary this choice is not effective if parameter dependence is encountered, something which is frequently occuring in real-world optimization problems rather than artificial test functions. So for parameter dependence the choice of CR=0.9 is more appropriate."

And in the scipy documentation [4], we have "Increasing this value allows a larger number of mutants to progress into the next generation, but at the risk of population stability."

```
x1(target): [ 7.  1.]
v1(mutant vector): [ 5.  4.]
Random value is: 1.5927390451028942
Random value is: -0.25195428695712796
Child of x1(target) and v1 (trial mutant) is: [7.0, 4.0]
```

```
print("x1(target):",x1)
print("v1(mutant vector):",v1)

# Control parameter CR
CR =  0.9

child =[]
for i in range(len(v1)):
    rand_num = np.random.randn()
    print("Random value is:", rand_num)
    if rand_num <= CR:
        child.append(v1[i])
    else:
        child.append(x1[i])

print("Child of x1(target) and v1 (trial mutant) is:",child)
```

4. Selection

Compare the objective function values for child and parent. Whichever has the lower value gets selected and placed in the new population.

```
rosenbrock(x1): 230436.0
rosenbrock(child): 202536.0
The selected population member is: child
```

```
print("rosenbrock(x1):",rosenbrock(x1))
print("rosenbrock(child):",rosenbrock(child))

pop_member_fun = {"x1":rosenbrock(x1), "child":rosenbrock(child)}
print("The selected population member is:",min(pop_member_fun, key=pop_member_fun.get)
```

5. Termination

Several criteria can be used as termination criteria for differential evolution:

1. Standard deviation of vector in population: Standard deviation of the vectors of a defined threshold. This method is also selected as termination criteria in scipy

implementation with one variation: normalization by the mean of the population (below).

Other criteria for termination:
2. Reference based criteria: If we know the optimum, we can set the error measure as the difference to the optimum.
3. Improvement based criteria: Number of iterations or function evaluations can be set to a set value.

```
def convergence(self):
        """
        The standard deviation of the population energies divided by their
        mean.
        """
        return (np.std(self.population_energies) /
                np.abs(np.mean(self.population_energies) + _MACHEPS))
print("The selected population member is:",min(pop_member_fun, key=pop_member_fun.get)
```

Scipy Implementation of Differential Evolution
The function signature of differential evolution in scipy is as follows:

Here is the breakdown of scipy implementation parameters according to general differential evolution stages:
1. *Initialize the population*: Popsize
2. *Mutation*: Mutation constant defines above.
3. *Cross-over*: Recombination
4. *Selection*
5. *Termination*: Tol value, also in callback : convergence. Population standard deviation represents the termination criteria in scipy. When it gets below the tol value, the solver is stopped.

```
scipy.optimize.differential_evolution(func, bounds, args=(), \
strategy='best1bin', maxiter=1000, popsize=15, tol=0.01, \
mutation=(0.5, 1), recombination=0.7, seed=None, callback=None,\
 disp=False, polish=True, init='latinhypercube', atol=0)
```

References

1. Storn, R and Price, K, Differential Evolution – a Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, Journal of Global Optimization, 1997, 11, 341 – 359.
1. D. Mandal, A. Chatterjee, and A. K. Bhattacharjee, "Design of Fully Digital Controlled Shaped Beam Synthesis Using Differential Evolution Algorithm," International Journal of Antennas and Propagation, vol. 2013, Article ID 713680, 9 pages, 2013. doi:10.1155/2013/713680
2. http://cci.lbl.gov/cctbx_sources/scitbx/differential_evolution.py
3. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html