Exploring NK Landscapes: A Hands-on Exercise

Maciej Workiewicz (ESSEC Business School)

2019/08/12

Plan for today

Int	Introduction
1	Exercise 1: Creating and surveying a rugged landscape
2	Exercise 2: Local search and long jumps
3	Exercise 3: Centralization and decentralization of search
D	Discussion

Introduction: Python Code

1	Exercise 1:
2	Exercise 2:
3	Exercise 3:

1_landscape_creation.py
PY File
7.92 KB

2_local_search.py
PY File
3.35 KB

3_decentralized.py
PY File
4.63 KB

Introduction: Using Anaconda Python environment - Spyder

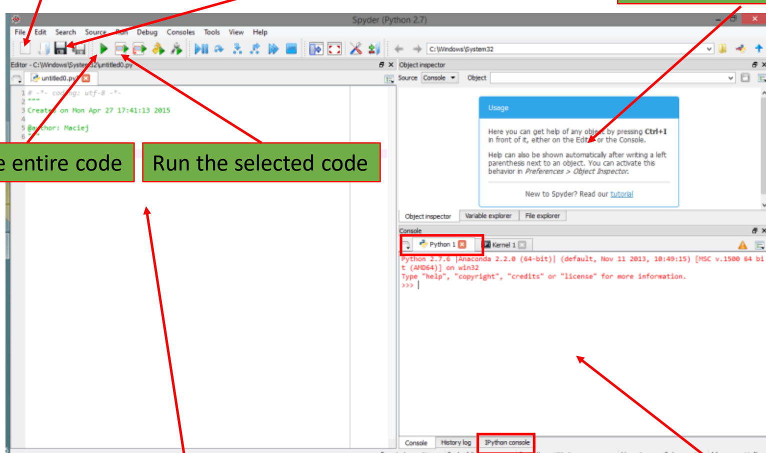
Start a new project

Save current project

information on objects press CNTR + I

Run the entire code

Run the selected code



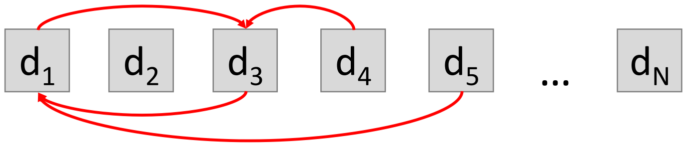
This is where you write your code

Here you see your code executed

Introduction: NK Models

Set of N binary decision variables D: $\{d_1, d_2, d_3, \ldots, d_N\}$

The performance contribution of an i^{th} decision variable depends on its own state (0 or 1) and states of the j other decision variables it depends on $\Pi_i = \Pi_i(d_i^1, d_i^2, \ldots, d_i^j)$



K is the average number of connections ⟶ per decision variable d_i

Π_i is drawn from a uniform distribution U(0, 1)

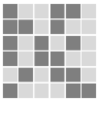
Total performance (fit) is: $\Pi = \frac{1}{N}\sum_{i=1}^{N} \Pi_i$

1 A Rugged Landscape

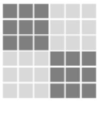
The first module: '1_landscape_creation V2.py' generates landscapes with desired properties (*type of an interaction landscapes* and *K*), calculates some basic statistics of those landscapes and saves the landscape as a binary file for future retrieval. This last step helps with subsequent exercises as we don't have to regenerate NK landscapes each time we run the simulation.

Types of **interaction matrices** with *K*=2

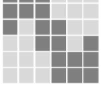
a) random



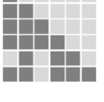
b) modular



c) nearly modular



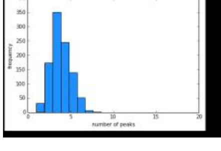
d) diagonal



The output of the module looks as follows:

```
--------------------------------------------------------
Running Module A: NK landscape creation and analysis
--------------------------------------------------------

Summary statistics for IMatrix: 1 K=2
average number of peaks: 3.479
maximum number of peaks: 8.0
minimum number of peaks: 1.0
average maximum value: 0.744161953592
average minimum value: 0.257992599443
 time: 7.37 sec
```



1 Exercise 1: Generating a Rugged Landscape

1. Open the file **1_landscape_creation.py** and review the code.

Creating a random interaction matrix

```python
def imatrix_rand():
    '''
    This function takes the number of N elements and K interdependencies
    and creates a random interaction matrix.
    '''
    Int_matrix_rand = np.zeros((N, N))
    for aa1 in np.arange(N):
        Indexes_1 = list(range(N))
        Indexes_1.remove(aa1)  # remove self
        np.random.shuffle(Indexes_1)
        Indexes_1.append(aa1)
        Chosen_ones = Indexes_1[-(K+1):]  # this takes the last K+1 indexes
        for aa2 in Chosen_ones:
            Int_matrix_rand[aa1, aa2] = 1  # we turn on the interactions with K
    return(Int_matrix_rand)
```

An example of a random interaction matrix

	0	1	2	3	4	5
0	1	1	0	1	0	0
1	0	1	1	1	0	0
2	0	0	1	0	1	1
3	0	0	1	1	0	1
4	0	0	1	0	1	1
5	1	0	0	1	0	1

1 Exercise 1: Generating a Rugged Landscape

Calculating fitness vector of a given combination

```python
def calc_fit(NK_land_, inter_m, Current_position, Power_key_):
    '''
    Takes the landscape and a given combination and returns a vector of fitness
    values for the vector of the N decision variables.
    '''
    Fit_vector = np.zeros(N)
    for ad1 in np.arange(N):
        Fit_vector[ad1] = NK_land_[np.sum(Current_position * inter_m[ad1]
                                          * Power_key_), ad1]
    return(Fit_vector)
```

Last decision variable

Current position={0, 1, 0, 1, 0, 1}

Interactions= {1, 0, 0, 1, 0, 1}

	0	1	2	3	4	5
0	1	1	0	1	0	0
1	0	1	1	1	0	0
2	0	0	1	0	1	1
3	0	0	1	1	0	1
4	0	0	1	0	1	1
	1	0	0	1	0	1

```
In [6]: Power_key
Out[6]: array([32, 16,  8,  4,  2,  1])
```

row = (0+0+0+4+0+1) = 5

NK_land

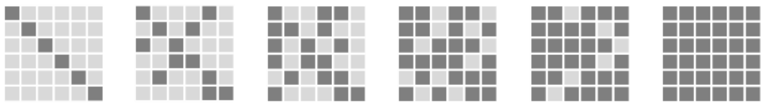
	0	1	2	3	4	5
0	0.824	0.013	0.949	0.886	0.844	0.500
1	0.159	0.129	0.715	0.129	0.838	0.154
2	0.899	0.868	0.292	0.794	0.250	0.413
3	0.587	0.307	0.429	0.023	0.313	0.314
4	0.221	0.919	0.506	0.338	0.588	0.334
5	0.591	0.754	0.175	0.042	0.931	0.030
6	0.206	0.985	0.750	0.783	0.347	0.860
7	0.945	0.410	0.375	0.813	0.027	0.705
8	0.641	0.881	0.871	0.431	0.035	0.964
9	0.751	0.258	0.849	0.168	0.096	0.006
10	0.826	0.917	0.340	0.254	0.582	0.370
11	0.450	0.879	0.388	0.864	0.683	0.514
12	0.120	0.973	0.513	0.101	0.741	0.170

Format Resize Background color

1 Exercise 1: Generating a Rugged Landscape

2. For a random interaction matrix **which_imatrix=1** generate NK landscapes with different levels of **K** (from 0 to 5)

Fig. 1.1 Examples of random interaction matrices for N=6



K=0 K=1 K=2 K=3 K=4 K=5

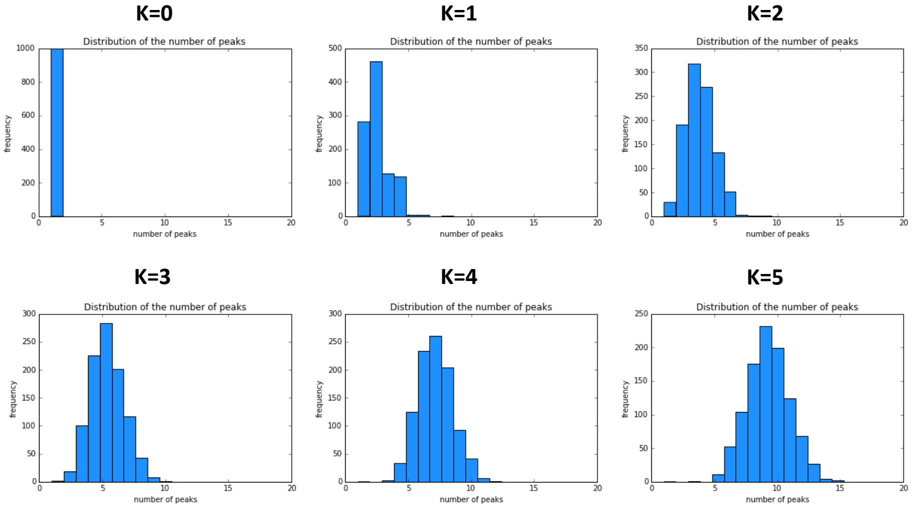
3. Note any observations:
   1. What happens to the average number of local peaks as you increase **K**?
   2. What are the effects on the number of local peaks and the average fitness level of the global peak?

1 The Topography of Rugged Landscapes

Your observations?

1 Rugged Landscapes – non-patterned (random) interactions

Fig. 1.2 Complexity and the number of local peaks: a sample of 1,000 landscapes



1 The Topography of Rugged Landscapes

Fig. 1.3 Average number of peaks

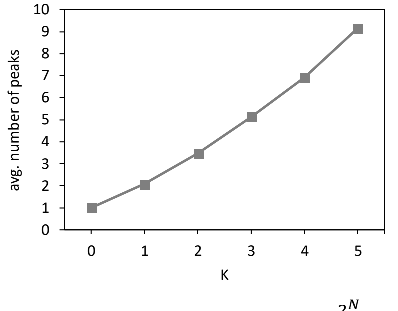
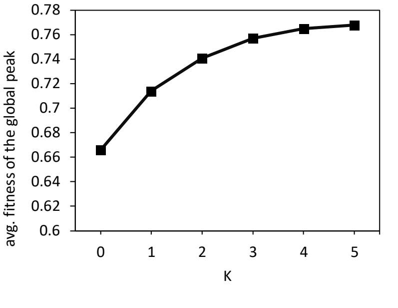


Fig. 1.4 Average fitness of the global peak



Average number of local peaks $= \frac{2^N}{N+1}$

Kauffman 1993:47

* All results for N=6 with a random interaction matrix and 10,000 iterations

2 Exercise 2: Local Search and Long Jumps

1. Open file **2_local_search.py** and review the code.

```python
for i1 in np.arange(i):
    combination = np.random.binomial(1, 0.5, N)  # gen initial combination
    row = np.sum(combination*power_key)  # finding the address in the array
    fitness = NK_landscape[i1, row, 2*N]  # piggyback on work done previously
    max_fit = np.max(NK_landscape[i1, :, 2*N])
    min_fit = np.min(NK_landscape[i1, :, 2*N])
    fitness_norm = (fitness - min_fit)/(max_fit - min_fit)  # normalize 0 to 1
    for t1 in np.arange(t):  # time for local search
        Output2[i1, t1] = fitness_norm
        if np.random.rand() < p_jump:  # check whether we are doing a jump
            new_combination = np.random.binomial(1, 0.5, N)
        else:  # if not, then we simply search locally
            new_combination = combination.copy()
            choice_var = np.random.randint(N)
            new_combination[choice_var] = abs(new_combination[choice_var] - 1)
        row = np.sum(new_combination*power_key)
        new_fitness = NK_landscape[i1, row, 2*N]
        if new_fitness > fitness:  # if we have found a better combination
            combination = new_combination.copy()
            fitness = new_fitness.copy()
            fitness_norm = (fitness - min_fit)/(max_fit - min_fit)
        # otherwise all stays the same as in the previous round
```

Setting up initial location

Determining whether to make a long jump

"Should I stay or should I go?"

2 Exercise 2: Local Search and Long Jumps

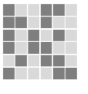
2. Run the code for different values of `K`. Keep `which_imatrix=1` and `p_jump=0`
3. Note any observations:
    1. What is the average fitness level achieved for different landscapes?
    2. Which types of NK landscapes are easier to scale for a locally searching agent?
4. Now play with the `p_jump` variable. Consider the following questions:
    1. What is the effect of adding random jumps `p_jump` to a locally searching agent?
    2. Can you explain the results?

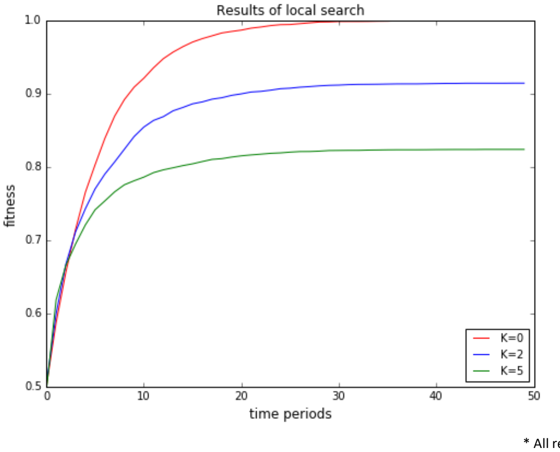
2 Local Search and Long Jumps

Your observations?

2 Local Search on a rugged landscape

Random Interaction Matrix

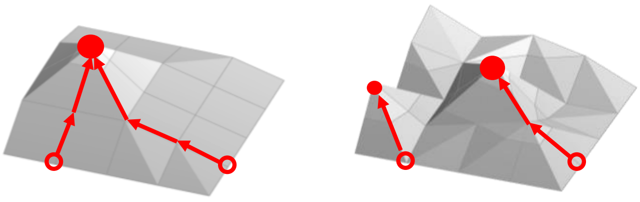




* All results for 1,000 iterations

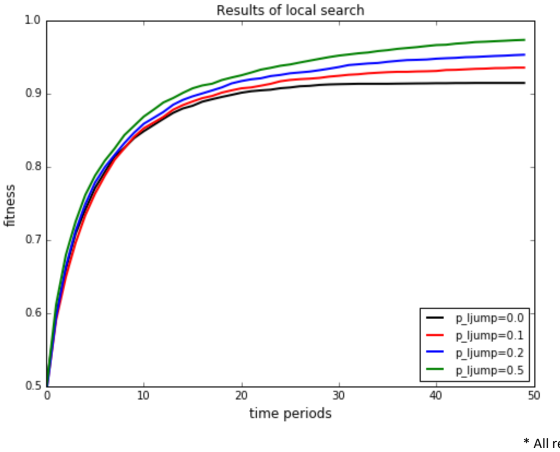
2 Local Search on a Rugged Landscape

simple (K=0) complex (K>0)



2 Local Search with Long Jumps on a Rugged Landscape

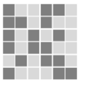
Random Interaction Matrix, K=2

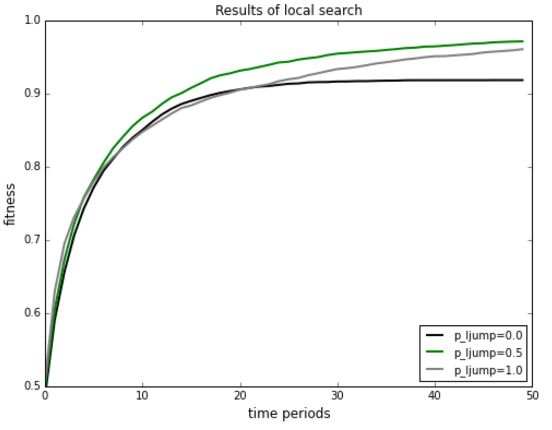


* All results for 1,000 iterations

2 Local Search with Long Jumps on a Rugged Landscape

Random Interaction Matrix, K=2

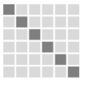


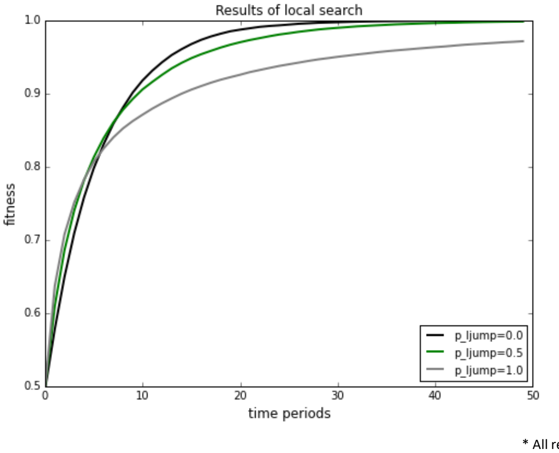


* All results for 1,000 iterations

2 Local Search with Long Jumps on a Rugged Landscape

Random Interaction Matrix, K=0





* All results for 1,000 iterations

3 Exercise 3: Centralization and Decentralization of Local Search

1. Open file **3_decentralized.py** and review the code

```
new_combination = combination.copy()
new_combA = combination[:int(N/2)].copy()
new_combB = combination[int(N/2):].copy()
choice_varA = np.random.randint(0, int(N/2))
choice_varB = np.random.randint(0, int(N/2))
new_combA[choice_varA] = abs(new_combA[choice_varA] - 1)
new_combB[choice_varB] = abs(new_combB[choice_varB] - 1)
new_combination[:int(N/2)] = new_combA.copy()
new_combination[int(N/2):] = new_combB.copy()

row = np.sum(new_combination*power_key)  # find address for new comb
new_fitA = np.mean(NK_landscape[i1, row, N:(int(N+N/2))])  # fitness goal 1
new_fitB = np.mean(NK_landscape[i1, row, (int(N+N/2)):int(N*2)])  # fitness goal 2
if new_fitA > fitA:
    combination[:int(N/2)] = new_combA.copy()
if new_fitB > fitB:
    combination[int(N/2):] = new_combB.copy()
row = int(np.sum(combination*power_key))
fitness = np.mean(NK_landscape[i1, row, N:2*N])  # final fitness
```

Split the decision vector

Compare separately

* All results for 1,000 iterations

3 Exercise 3: Centralization and Decentralization of Local Search

2. Run the code for `which_imatrix=1`, and `reorg=50`. Try different values of `K`
3. Note any observations:
    1. What is the effect of decentralizing local search?
    2. What would be an organizational analogy of such parallel search?
4. With `K=2`, introduce reorganization `reorg` at some period between 1 and 49.
    1. What do you observe? Can you explain the results?

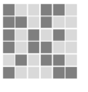
* All results for 1,000 iterations

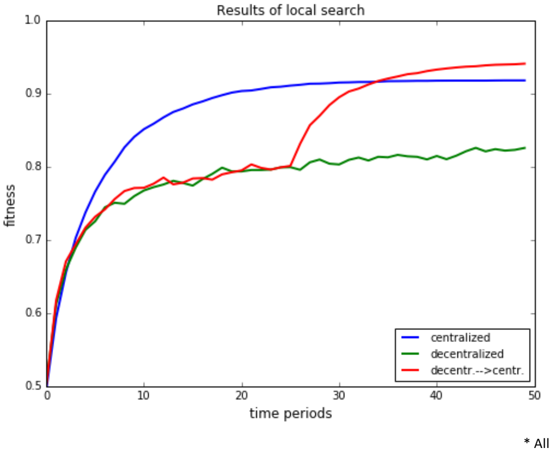
3 Centralization and Decentralization of Local Search

Your observations?

3 Centralization and Decentralization of Local Search

Random Interaction Matrix, K=2

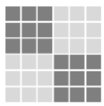
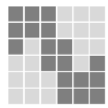
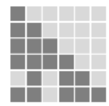




* All results for 1,000 iterations

3 Exercise 3: Continued

1. Open file **1_landscape_creation.py** again
2. This time set K=2 and examine other types of interaction matrices, i.e., set `which_imatrix` to **2, 3,** and **4**.

2  **3**  **4** 

3. Note the results. How do they compare to those you obtained in Exercise 1?
4. Now, go back to **3_decentralized.py**
5. Run the code for different types of NK landscapes just created (set `K`=2, `which_imatrix`= 2, 3, and 4)
6. What changes do you observe compared with the first set of results of Exercise 3?

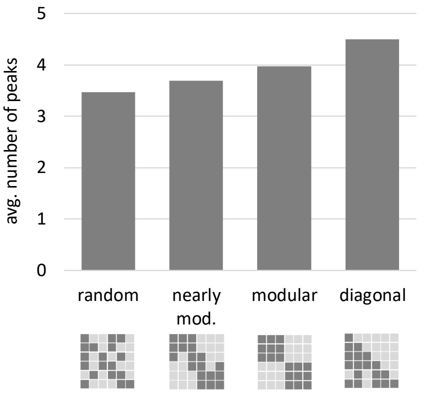
* All results for 1,000 iterations

3 Centralization and Decentralization of Local Search

Your observations?

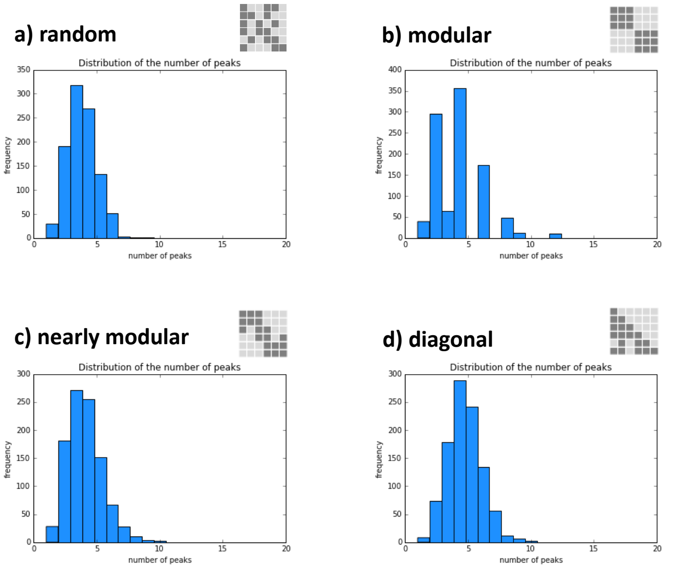
3 More on The Topography of Rugged Landscapes

Figure 3 Avg. number of peaks for different IM, each with K=2



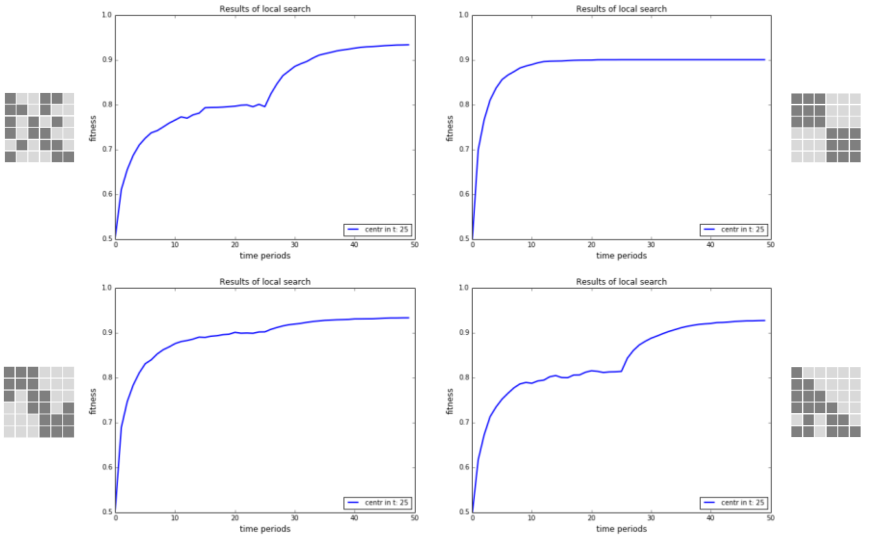
* Results for N=6 and 10,000 iterations

3 Rugged Landscape with K=2



3 Centralization and Decentralization of Local Search

With reorganization in the 25th round



* All results for 1,000 iterations