Optimization Labs

Manuela Corte Pause

July 2, 2024

Contents

1 Grid Search, Random Seach, Nealder-Mead and Powell **3**
1.1 Grid Search . . . 3
1.2 Random Search . . . 4
1.3 Powell . . . 4
1.4 Nelder-Mead . . . 5

2 DIRECT, Basing Hopping **8**
2.1 DIRECT . . . 8
2.2 Basin Hopping . . . 9

3 Derivative Based Optimization **12**
3.1 Gradient Descent . . . 12
3.2 Newton's Method . . . 13
3.3 BFGS . . . 14

4 Variable Neighbourhood Search **15**
4.1 Variable Neighborhood Search . . . 15
4.2 Reduced Variable Neighborhood Search . . . 16

5 Iterated Local Search, Simulated Annealing **17**
5.1 Iterated Local Search . . . 17
5.2 Simulated Annealing . . . 17

6 Bayesian Optimization **19**

7 Genetic, Evolution and Nature-Inspired Analogies **23**
7.1 Genetic Algorithms . . . 23
7.2 Evolution Strategies . . . 26
7.3 Particle Swarm Optimization . . . 27

8 Multi-Objective Optimization **29**
8.1 Kursawe Function . . . 29
8.2 Multiple-Disk Clutch Brake Optimization . . . 31

9 Design of Experiments **33**

10 Linear Programming **35**
10.1 Exercise 1 35
10.2 Exercise 2 35
10.3 Exercise 3 36

11 Integer and Dynamic Programming **38**
11.1 N Queens 38
11.2 TSP 38
11.3 Knapsack 39

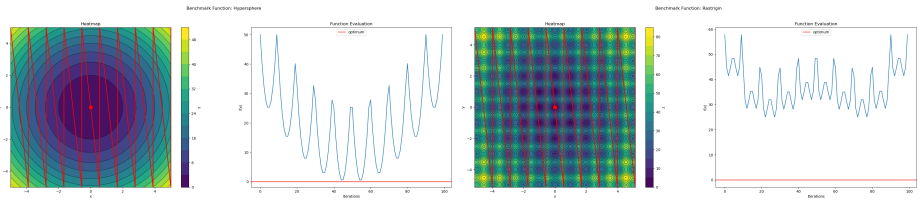
12 Robust Optimization **41**
12.1 Robust KnapSack Problem 41
12.2 Robust Portfolio Optimization 41

1 Grid Search, Random Seach, Nealder-Mead and Powell

1.1 Grid Search

Grid search performs an uninformed search on the entire search space of the optimization function. If we decrease the step size (or equivalently increase the number of steps), we would be able to eventually find the global minimum. However, this comes at the cost of increased computational time which is especially problematic for complex functions. Moreover, in high-dimensional spaces, the number of steps required to find the global minimum increases exponentially.

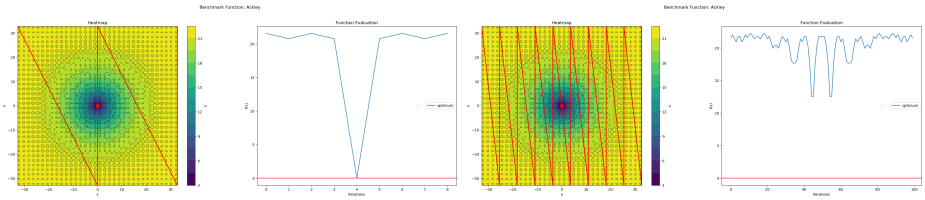
For example below 1 we report the results of grid search for the Rastrigin function in 2D which is a non-convex highly multimodal function and compare then with the much more simple Hypersphere function. We can see that with 100 steps we are able to get an approximation that is rougly comparable despite the difference in complexity of the two functions.



(a) Hypersphere function with 100 steps (b) Rastrigin function with 100 steps

Figure 1: Comparison between grid search on the Hypersphere and Rastrigin functions

Given that an uniformed search is performed, this means that there is no risk of getting stuck in a local minima but at the same time no information abount the actual shape of the function is used to guide the search. This means that the search is not efficient and the number of steps required to find the global minimum can be very high. At the same it time it can also happen that a higher number of steps doesn't result in an improved solution simply because the discretization of the search space ends up sampling points with a worse objective function. For example we can see this in the example below 2 with the Ackley function where 10 steps are able to perfectly find the minimum while 100 steps can't (this is mainly due to the fact the search space is a square and the minimum is exactly in zero so the discretization with 10 steps contains exactly the minimum).

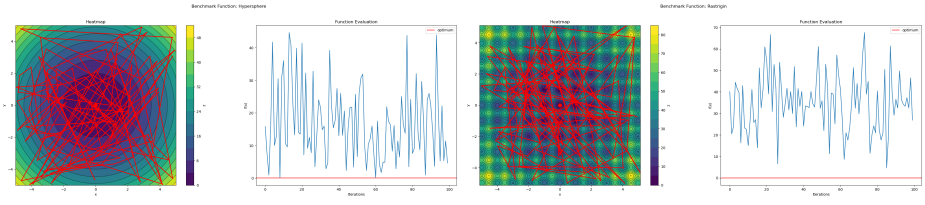


(a) Ackley function with 10 steps (b) Ackley function with 100 steps

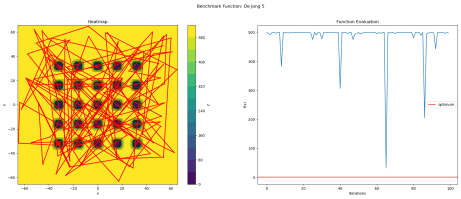
Figure 2: Grid search on the Ackley function

1.2 Random Search

Random search is an uninformed search method that samples the search space randomly. Given that the samples are taken randomly in the search space, all functions are equally hard to optimize (assuming that all fitness functions are equally computationally hard). For example below 3 we can see different functions, all with 100 samples and see that the results are comparable in terms of the quality of the solution found regarless of function.



(a) Hypersphere (b) Rastrigin



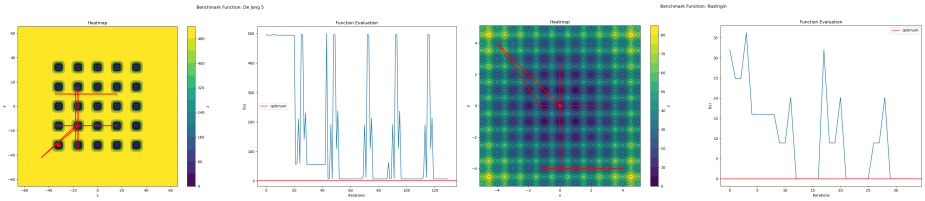
(c) DeJong5

Figure 3: Random search on different functions with 100 samples

1.3 Powell

Powell's method is a conjugate direction method that uses a set of directions to perform a line search. The directions are updated at each iteration to minimize the function. The method is only guaranteed to find a local minimum but in practice it is often able to find the global minimum as well. For example, in the image below 4 we can see the results of Powell's method on the Rastrigin

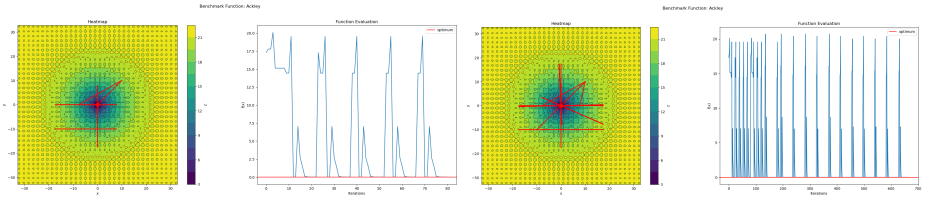
function, where the method is able to find the global minimum, and the DeJong5 function, where it's not able to, even if the functions are both multimodal.



(a) DeJong5 (b) Rastrigin

Figure 4: Powell's method comparison

Another important aspect of Powell's method is the choice of the initial directions. In the graph below 5 we can see that even if the method is able to find the global minimum in both scenarios, the choice of the initial directions has an impact on the number of iterations required to find the minimum.

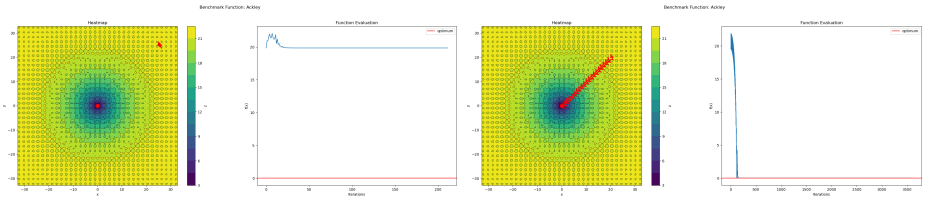


(a) Ackley with directions [[1, 0],[0,1]] (b) Ackley with directions [-1,0],[0,-1]

Figure 5: Powell's method on the Ackley function with different initial directions

1.4 Nelder-Mead

Nelder-Mead is a direct search method that uses a simplex to perform the optimization. Compared to Powell, it's much more likely to get stuck in a local minimum and its performance is dependent on the choice of the initial point. For example below 6 we can see the results of Nelder-Mead on the Ackley function where the method is able to converge to the global minimum with the right initial point but not with a different one.



(a) Ackley with initial point [20,20] (b) Ackley with initial point [25,25]

Figure 6: Nelder-Mead on the Ackley function with different initial points

Below 7 we can see the results of Nelder-Mead on the Rosenbrock function and we can see that the algorithms tends to obtain good results in unimodal functions with hard to approximate optima.

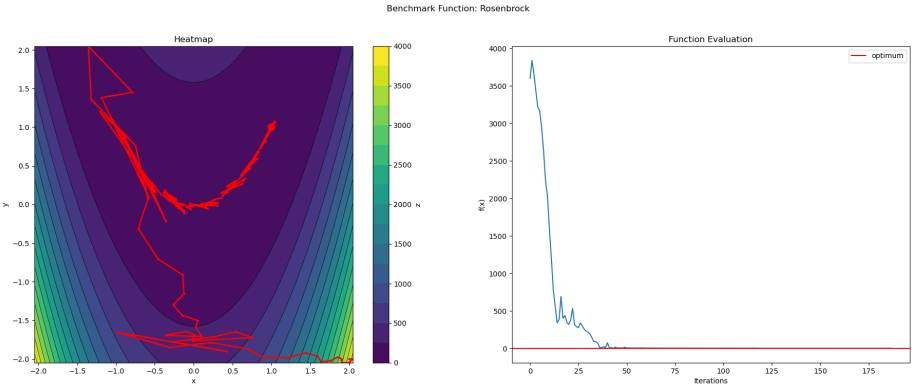
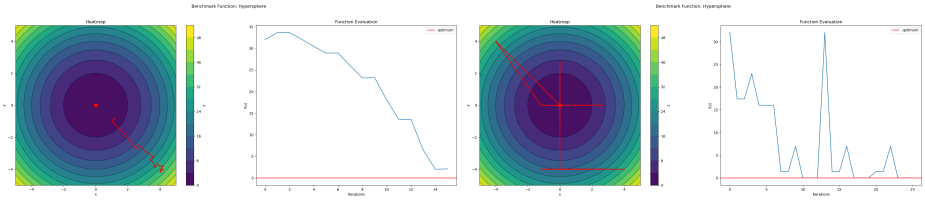


Figure 7: Nelder-Mead on the Rosenbrock function

Even on the functions where the method performs well, the number of iterations required is quite high. For example, below 8 we compare the results of the Nelder-Mead and the Powell algorithm on the Hypersphere function, both with 10 iterations.



(a) Nelder-Mead with 10 iterations (b) Powell with 10 iterations

Figure 8: Nelder-Mead and Powell on the Hypersphere function with 10 iterations

We can see how different methods perform differently on different functions. This is also supported by the No Free Lunch Theorem which states that no optimization algorithm is better than any other when their performance is averaged over all possible functions. This means that the choice of the optimization algorithm is dependent on the specific problem at hand.

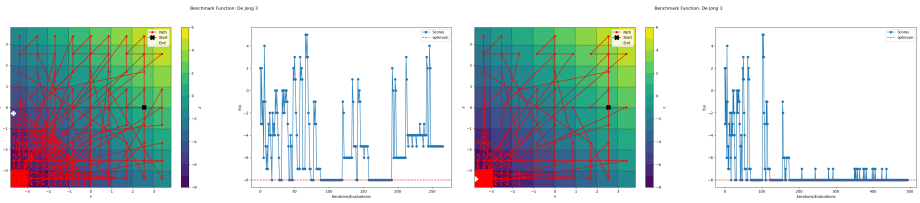
2 DIRECT, Basing Hopping

2.1 DIRECT

DIviding RECTangles (DIRECT) is a partitioning algorithm that recursively subdivides the feasible region into smaller hyperrectangles. The algorithm is as follows:

1. Divide the feasible region into smaller hyperrectangles.
2. Evaluate the function at the center of each hyperrectangle. The division is performed so that the region with the best function value is given the largest space.
3. a set of potentially optimal hyperrectangles is identified and further divided.

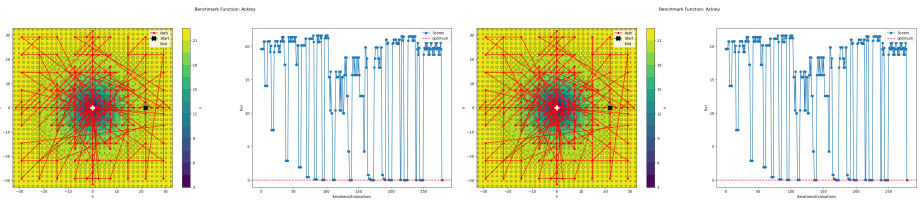
The stopping condition is either based on the maximum number of iterations / function evaluations or on the improvemnt of the function value. This means that the specific tolerance is dependent on the problem being solved. For example in the case of the De Jong's function 9 we can see that different tollerance levels lead to different results.



(a) $\epsilon = 0.1$ (b) $\epsilon = 0.001$

Figure 9: DIRECT on De Jong's function with different tolerances

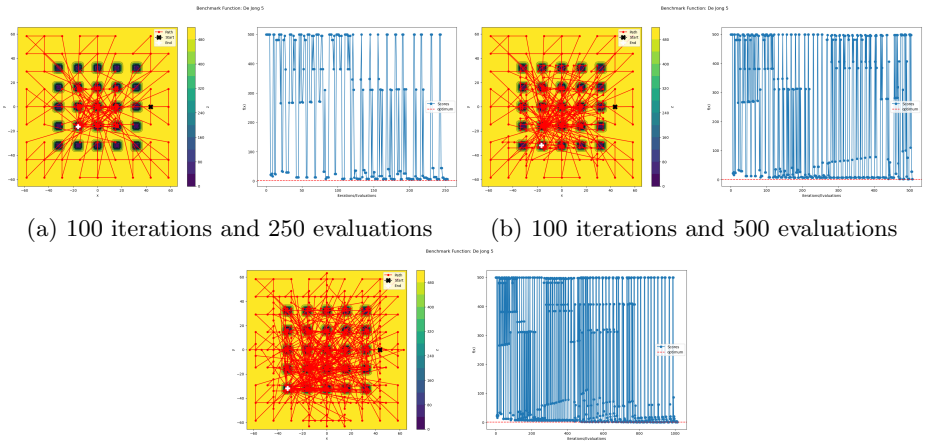
For other functions, such as the Ackley function, the tolerance level does not have an impact on the results, probably bacause the function has a more clear basin of attraction towards the global minimum.



(a) $\epsilon = 0.1$ (b) $\epsilon = 0.001$

Figure 10: DIRECT on Ackley's function with different tolerances

As we would expect, a larger number of iterations leads to a better approximation of the global minimum, in some more extreme cases even changing the basin of attraction of the global minimum. This is the case for the De Jong's function 11 where the minimum found is significantly different for different numbers of evaluations.



(a) 100 iterations and 250 evaluations (b) 100 iterations and 500 evaluations

(c) 100 iterations and 1000 evaluations

Figure 11: DIRECT on De Jong's function with different number of evaluations

Another interesting thing to note is the behaviour of the algorithm with a large number of local minima and how we can see how DIRECT evaluations are much more spread out compared to other functions. This can be seen really well in the Griewank function 12.

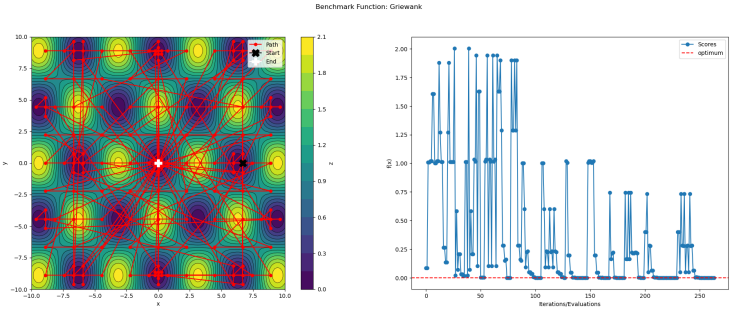


Figure 12: DIRECT on Griewank's function

2.2 Basin Hopping

Basing Hopping is a form of Iterated Local Search with a different starting point each time. It loops through the following steps:

1. *Hopping*: perturbation of the current solution ("jump" to new parts of the search space)

2. *Local Search*: perturbed solution is optimized using a local search method

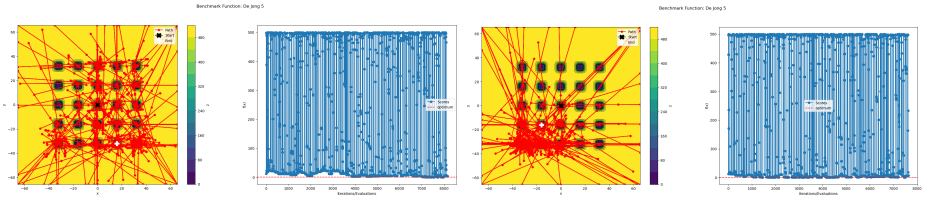
3. *Acceptance* the new solution is accepted or rejected based on an acceptance criterion. This criterion can be defined in different ways; here is defined based on the Metropolis criterion.

It's particularly important to set the temperature and the stepsize parameters correctly as specified in the scipy documentation. In particular:

1. *Temperature*: usually set to be comparable with with the separation of the objective function value of local minima.

2. *Stepsize*: should be set to be comparable to the Distance in the coordinates between local minima.

For example, we can see how the algorithm behaves with the Dejong5 function 13 with different temperatures (10 and 0.1) and that the temperature has a significant impact on the evaluations: The higher temperature leads to a more spread out evaluation of the function because the algorithm is more likely to accept worse solutions and leave the local basin of attraction.



(a) Temperature = 10 (b) Temperature = 0.1

Figure 13: Basin Hopping on De Jong's function with different temperatures and stepsize 20

The importance of the stepsize can be seen by comparing the previous results with the ones obtained with a stepsize of 1 (Figure 14) where we can clearly see that the algorithm isn't able to leave the local basin of attraction.

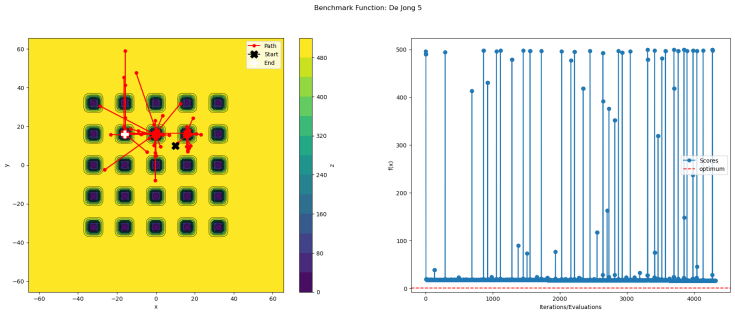


Figure 14: Basin Hopping on De Jong's function with stepsize = 1

It's also interesting to note that, even if the global minimum is found, this isn't necessarily the last evaluation of the function. This makes sense because the only stopping condition is the number of iterations, so the algorithm will continue to evaluate the function even after finding the global minimum.

If we have a unimodal function, such as the Rosenbrock function 15, the algorithm is able to immediately find the global minimum via the local search step.

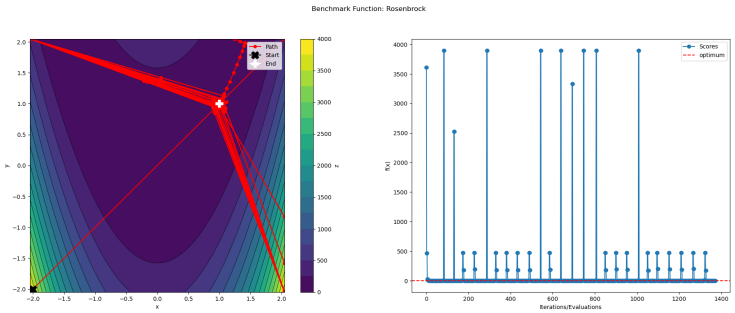


Figure 15: Basin Hopping on Rosenbrock's function

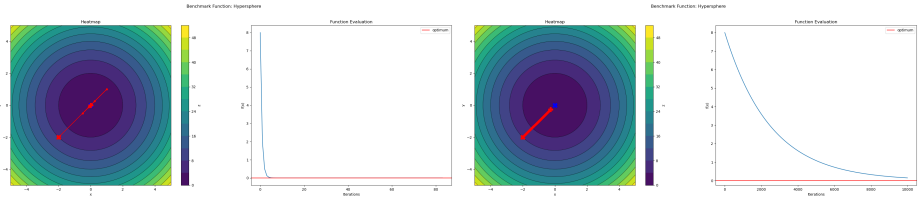
Note: the code was slightly modified to add bounds to the local search algorithm (L-BFGS-B) to avoid the algorithm from going out of bounds.

3 Derivative Based Optimization

3.1 Gradient Descent

Gradient descent is a first order optimization method, which means that it uses the first derivative of the function to find the minimum. The algorithm works by taking steps in the opposite direction of the gradient of the function, which is the direction of the steepest descent. The step size is controlled by the learning rate, which is a hyperparameter.

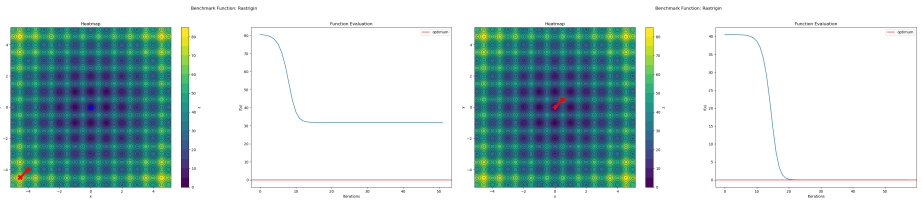
If the learning rate is too big then the gradient descent can overshoot the minimum and only then adjust to convergence. If the learning rate is too small then the gradient will take more iterations to converge but for convex functions the gradient descent will always converge at some point given enough steps 16.



(a) Gradient descent with learning rate 0.75 (b) Gradient descent with learning rate 0.4

Figure 16: Gradient descent on a 2D hypersphere

For multimodal functions (many local minima), the gradient descent is only guaranteed to converge to the nearest local minimum, thus the starting point is very important.



(a) Starting point[-4.5,-4.5] (b) Starting point [0.5,0.5]

Figure 17: Gradient descent on the Rastrigin function

It the learning rate is large, gradient descent can also lead to numerical instability and overflow 18.

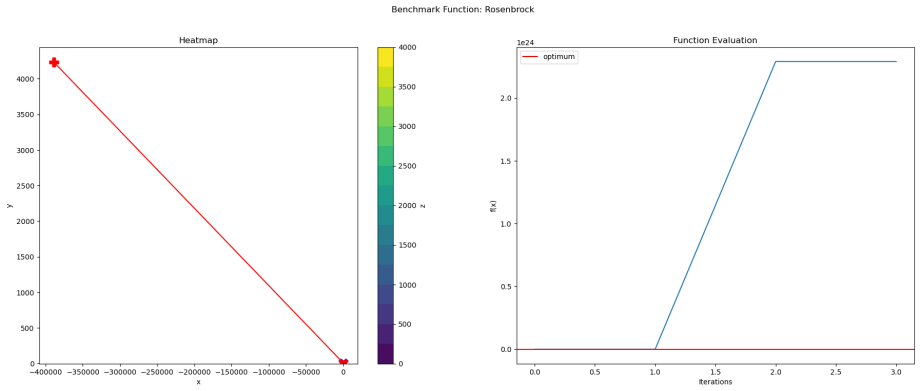


Figure 18: Gradient descent with learning rate 0.01 on the Rosenbrock function. The algorithm diverges.

3.2 Newton's Method

Newton's method is a second order optimization method, which means that it uses the second derivative of the function to find the minimum. Thus it's more informative than the gradient descent (it doesn't just use the slope of the function, but also the curvature). It can converge faster than gradient descent but it's computationally expensive since it needs to compute the inverse of the Hessian matrix.

Moreover, it has the advantage of having one less parameter to tune (the learning rate) since the algorithm uses instead information about the second derivate of the function to compute the step size. Even if convergence is on average faster, the same convergence guarantees as the gradient descent apply.

ON the majority of the functions this method doesn't converge or may even founds a worse solution that the original one. Given that this behaviour is not expected, we assume there is a problem in the implementation of the algorithm. We can see this behaviour in the Rastrigin function 19.

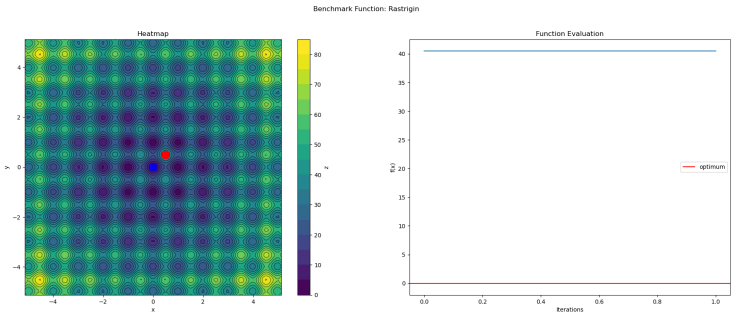
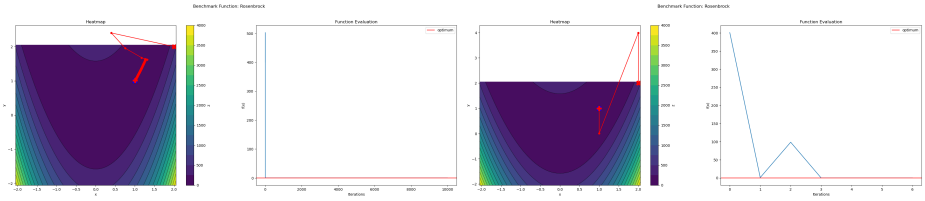


Figure 19: Newton's method on the Rastrigin function

3.3 BFGS

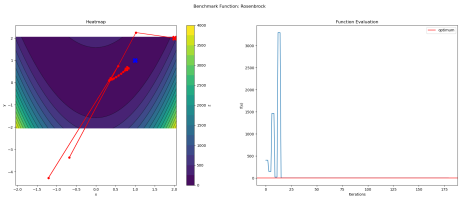
BFGS is a quasi-Newton method, which means that it's a second order optimization method that doesn't need to compute the Hessian matrix. It's a good compromise between the gradient descent and Newton's method, since it's faster than the gradient descent but doesn't have the same overhead of computing the Hessian matrix as Newton's methods.

Below 20 we report an comparison between the three methods on the Rosenbrock function starting from point [2.0,2.0] and compare the number of iterations needed. It's interesting to notice that the BFGS terminates before reaching the global minimum and even changing the tolerance doesn't help in this case. Nevertheless, we obtain the expected result where the Newton's method converges in the fewest iterations, followed by the BFGS and then gradient descent.



(a) Gradient descent (10000 iterations)

(b) Newton's method (7 iterations)



(c) BFGS (92 iterations)

Figure 20: Comparison between Gradient Descent, Newton's Method and BFGS on the Rosenbrock function

4 Variable Neighbourhood Search

4.1 Variable Neighborhood Search

Variable Neighborhood Search is a optimization algorithm that combines local search with a neighborhood change strategy. The algorithm is made up of three main components:

- Shaking: generates a new solution in the neighborhood of the current solution.
- Local Search: improves the solution generated by the shaking step. This is done by applying either first-improvement or best-improvement.
- Move or Not: if the solution generated by the local search is better than the current best solution, the current solution is replaced by the new one and the size of the neighborhood is resetted to k_{min}. Otherwise, the size of the neighborhood is increased in the hope of finding a better solution in a larger neighborhood.

In our example we want to solve the Knapsack-01 problem and thus the neighborhood structure N_k is defined as the set of all solution that can be obtained by switching k bits in the current solution. It's easy to conclude that the performance of the algorithm is highly dependent on the maximum number of bits that can be switched 1. Obviously the quality of the solution achieved is also dependent on the stochastic component of the algorithm so we present all following results as the average of 100 runs

k	quantity	capacity
2	11.0	8.81
5	13.89	9.52
7	14.36	9.57
10	14.59	9.64

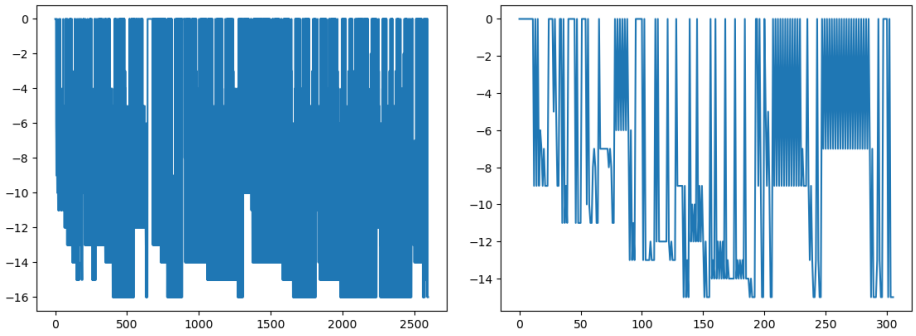
Table 1: Different neighborhood structures

On the other hand if we have a diverse enough neighborhood structure, the initial point is not as important, as the algorithm will be able to explore the search space anyway 2.

starting point	quantity	capacity
(1, 0, 0, 1, 0, 1, 0, 1, 0, 1)	14.48	9.66
(0, 1, 0, 1, 1, 1, 0, 1, 0, 0)	14.72	9.76
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)	15.19	9.92
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)	14.62	9.9

Table 2: Different starting points with $k = 10$

The other important choice that can be made is the choice of the local search algorithm. Obviously, best-improvement yields better results than first-improvement, but it is also more expensive. In our case, the problem is small enough that there isn't a big difference between the two 21.



(a) Best-improvement (b) First-improvement

Figure 21: VNS with different local search algorithms

4.2 Reduced Variable Neighborhood Search

Reduced Variable Neighborhood Search is a variant of the VNS algorithm that skips the most expensive step of VNS which is the local search.

Similarly to VNS, if the neighborhood structure is diverse enough the starting is not very important 3 but if the neighborhood is limited then it becomes much more important that in simple VNS given it's the only way we have to explore the search space. For example in table 4 we can see that for $k = 2$ the algorithm was unable to find a valid solution in the majority of the runs.

starting point	quantity	capacity
(1, 0, 0, 1, 0, 1, 0, 1, 0, 1)	12.33	9.23
(0, 1, 0, 1, 1, 1, 0, 1, 0, 0)	13.09	9.42
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)	12.85	9.41
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)	12.53	9.02

Table 3: Different starting points with $k = 10$

k	quantity	capacity
2	1.11	0.95
5	7.37	5.74
7	11.57	8.57
10	12.54	9.17

Table 4: Different neighborhood structures

5 Iterated Local Search, Simulated Annealing

In this lab, we will implement two different optimization methods for the Knapsack problem: Iterated Local Search and Simulated Annealing.

5.1 Iterated Local Search

Iterated Local Search (ILS) is a stochastic local search method that generates a sequence of solutions generated by an embedded heuristic. It can be seen as a family of algorithms based on the same basic procedure:

1. Generate an initial solution s.
2. Local search to obtain a local minimum $s*$.
3. Repeat until a stopping criterion is met:
   (a) Perturb $s*$ to obtain a new solution s'.
   (b) Apply a local search procedure to s' to obtain a local minimum $s*'$.
   (c) Accept either $s*$ or $S*'$ as the new current solution based on some acceptance criterion.

ILS can be seen as a general framework that can be instantiated in many ways by choosing different local search procedures, perturbation mechanisms and intensities, and acceptance criteria. In this instance, we are basically reimplementing the Variable Neighborhood Search (VNS) algorithm (see lab 4 for more details).

5.2 Simulated Annealing

Simulated Annealing is a optimization algorithm inspired by the annealing process in metallurgy. The algorithm starts with an initial solution and iteratively moves to a new solution. The new solution is accepted if it improves upon the current solution or, if it's worse,depending on a probability that decreases with time. The probability of accepting a solution that is worse than the current solution is given by the Metropolis criterion $p = e^{-(f(x')-f(x))/T}$. The temperature T is a parameter that controls the probability of accepting a worse solution and is decreased over time according to a cooling schedule ($T = \alpha T$).

The first parameter that we need to evaluate is the temperature T 5. As we can see, the best performance is achieved with $T = 1$ or smaller which makes sense since the difference in the objective function is small (remember the observation made abount temperature in Basing Hopping).

T	quantity	capacity
0.1	14.62	9.79
0.5	14.64	9.88
1.0	14.65	9.83
5.0	13.72	9.61
10.0	12.94	9.22

Table 5: Different temperatures

The initial solution doesn't seem to be important for the performance of the algorithm 6

starting point	quantity	capacity
(1, 0, 0, 1, 0, 1, 0, 1, 0, 1)	14.72	9.82
(0, 1, 0, 1, 1, 1, 0, 1, 0, 0)	14.61	9.85
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)	14.79	9.84
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)	14.63	9.8

Table 6: Different starting points with $k = 10$ and $T = 1$

Differently from ILS and VNS, the neighbourhood structure is not as important for the performance of the algorithm 7 probably due to the different accepting criteria used that allows the algorithm to explore the search space more freely. Even though the difference is small, the best performance is still achieved with $k = 10$.

k	quantity	capacity
2	11.85	9.5
5	14.49	9.81
7	14.63	9.83
10	14.70	9.79

Table 7: Different neighborhood structures and $T = 1$

The last parameter that we need to evaluate is the cooling schedule. We can see that the performance of the algorithm is not very sensitive to the choice of the cooling schedule 8.

k	quantity	capacity
0.1	14.58	9.74
0.3	14.65	9.80
0.5	14.70	9.81
0.7	14.70	9.79
0.9	14.45	9.74

Table 8: Different cooling schedules and $T = 1$ and $k = 10$

6 Bayesian Optimization

Bayesian Optimization is a optimization algorithm that uses a probabilistic model to approximate the objective function *(surrogate model)* and uses this model to guide the search for the optimal solution. The general procedure can be described as follows:

1. choose surrogate function that approximates the real objective function (prior)
2. repeat until stopping condition
   (a) given a number of observations (computed from the real objective function), update the surrogate function (posterior distribution). Default choice is usually a Gaussian Process
   (b) optimize a cheap *acquisition function / utility function* based on the posterior distribution to find the new point to sample. Also has the responsability of balancing exploration and exploitation

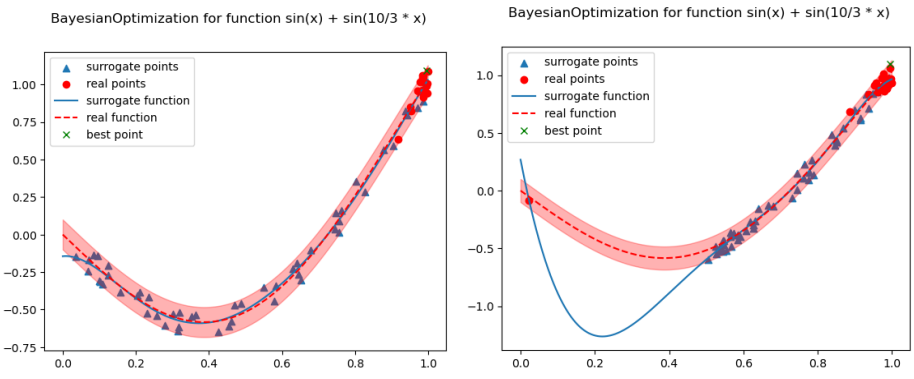
Choice of Prior In this lab, we'll use three different prior distribution to sample the initial points for the optimization algorithm. The three distributions are:

- *Prior1*: Uniform in the range $[0, 1]$
- *Prior2*: Uniform in the range $[0, 0.5]$
- *Prior3*: Uniform in the range $[0.5, 1]$

Prior	**Mean**	**Std. Deviation**
Prior1	(0.9967, 1.0966)	(0.0026, 0.0198)
Prior2	(0.9963, 1.0999)	(0.0043, 0.0206)
Prior3	(0.6991, 0.7897)	(0.4528, 0.4687)

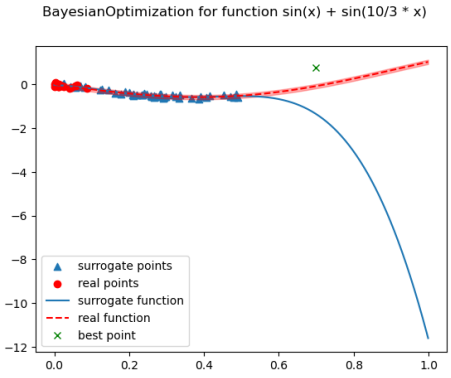
Table 9: Best point found with Objective1 and UCB acquisition function

We can see that Prior3 achieves worse results due to the fact that the prior distribution doesn't cover the optimal region. In particular the standard deviation is very high, which means tha the Bayesian Optimzation found good solutions in some cases but it's not consistent.



(a) Prior1

(b) Prior2



(c) Prior3

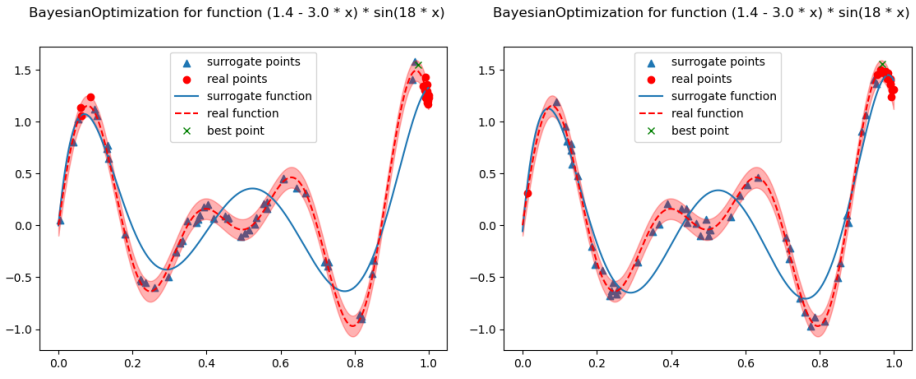
Figure 22: Objective1 with different prior distribution

It's also interesting to note how the Gaussian Process differently approximate the real objective function based on the prior distribution: Prior1 better approximates the real function since it covers the entire range of the function.

Acquisition Function The results are roughly the same for all the acquisition functions, with EI slightly worse than the others.

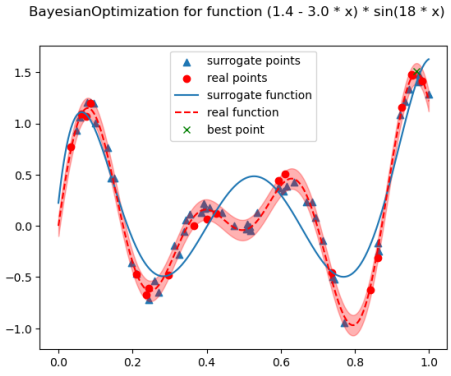
Prior	Mean	Std. Deviation
UCB	(0.9715, 1.5490)	(0.0055, 0.0202)
LCB	(0.9688, 1.5527)	(0.0070, 0.0297)
PI	(0.9661, 1.5107)	(0.0101, 0.0556)
EI	(0.9693, 1.4752)	(0.0142, 0.0649)

Table 10: Best point found with Objective1 and UCB acquisition function

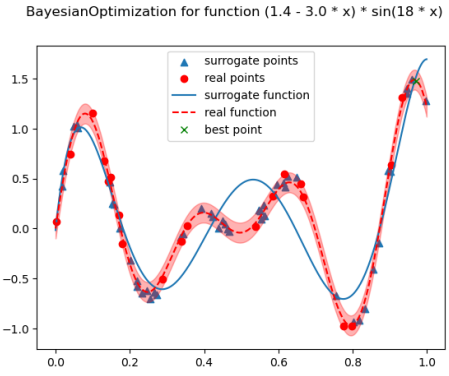


(a) UCB

(b) LCB



(c) PI



(d) EI

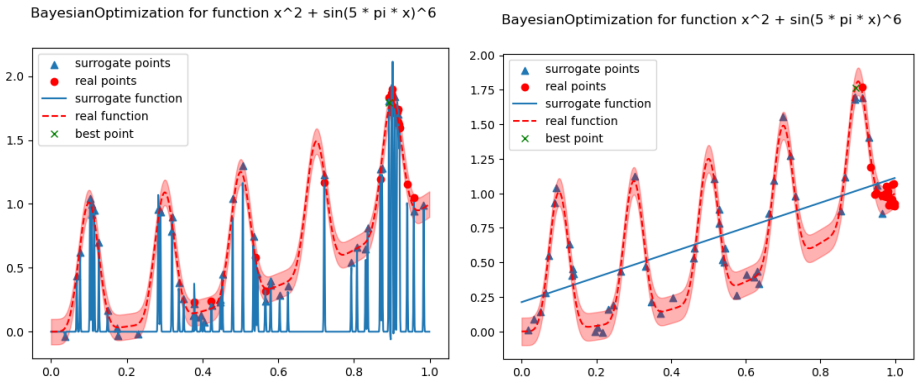
Figure 23: Objective2 with different acquisition functions

We can see that the acquisition functions have two main behavious: UCB and LCB focus right away on the optimal region while PI and EI are more focused on the exploration of the space.

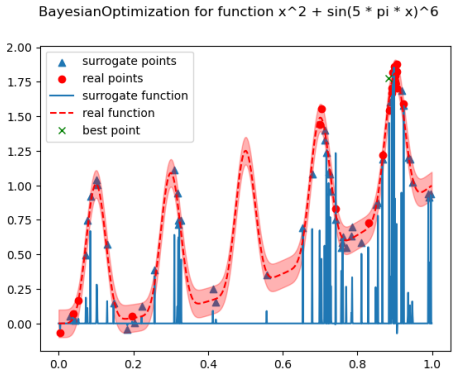
Kernel	**Mean**	**Std. Deviation**
RBF	(0.8909, 1.7960)	(0.0500, 0.1031)
DotProduct	(0.8937, 1.7634)	(0.0399, 0.1153)
ExpSineSquared	(0.8828, 1.7768)	(0.0599, 0.1307)

Table 11: Best point found with Objective3 and UCB acquisition function

Kernel The results are roughly the same for all the kernels and all have quite high variance.



(a) RBF (b) DotProduct



(c) ExpSineSquared

Figure 24: Objective3 with different kernels

We can see that differently than without the kernel, the Gaussian Process in not able to approximate the real function well but the sampled points are still in the optimal region. Moreover, in some cases the Gaussian Process with the kernel isn't able to converge, possibly due to kernel's hyperparameters that are not well tuned.

7 Genetic, Evolution and Nature-Inspired Analogies

7.1 Genetic Algorithms

Genetic algortithms are a class of metaheuristics ispired by the process of natural selection where the best individuals are the ones with the best fitness/objective function value. The general schema of GA is as follows:

1. Generate an initial population of individuals.
2. Evaluate the fitness of each individual.
3. Repeat until the termination condition is met:
   (a) Select individuals for reproduction.
   (b) Crossover and mutate the selected individuals.
   (c) Evaluate the fitness of the new individuals.
   (d) Replace the old population with the new population.

Except if differently specified, the following parameters are used for all following experiments:

- Population size: 50
- Number of generations: 50
- Objective function: Ackley
- Gaussia mutation with stadard deviation 1.0
- Number of simulations: 10

Mutation only GA The GA with only mutation has slightly worse performance for high mutation rates probably because the perturbation is too big to precisely converge to the optimum but in general it achieves satisfactory performances. The results are shown in Table 12.

Mutation rate	Average fitness	Standard deviation
0.1	0.0435	0.0244
0.2	0.0355	0.0183
0.3	0.0425	0.0297
0.4	0.0568	0.0233
0.5	0.0621	0.0177
0.6	0.1061	0.0585
0.7	0.0733	0.0601
0.8	0.1021	0.0596
0.9	0.1222	0.0819
1.0	0.1787	0.0863

Table 12: Mutation rate only

Crossover only GA We now test the case where only crossover is used and we notice how the algorithm is not able to find a good solution. This is because without mutation there is no introduction of new genetic material so the algorithm is not able to explore the search space. The results are shown in Table 13.

Crossover rate	Average fitness	Standard deviation
0.1	7.997	0.0863
0.2	8.234	0.0863
0.3	7.376	0.0863
0.4	5.875	0.0863
0.5	5.798	0.0863
0.6	5.224	0.0863
0.7	4.660	0.0863
0.8	5.539	0.0863
0.9	6.538	0.0863
1.0	6.366	0.0863

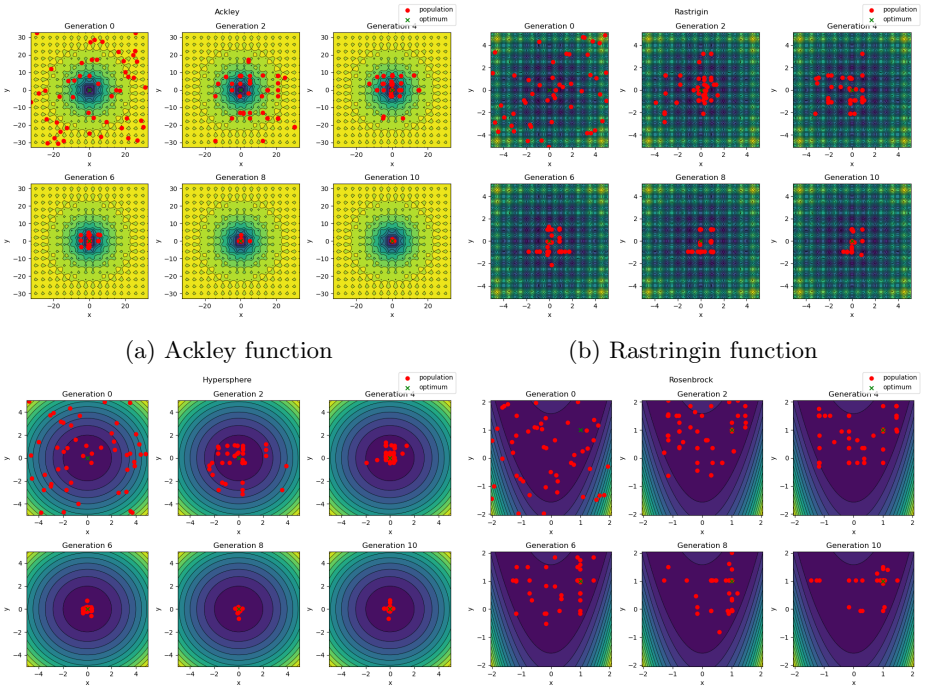
Table 13: Crossover rate only

Tournament size We now test the tournament size selection method. We can see that the best performances are obtained for large tournament sizes. This is because the selection pressure is higher and the Ackely function has many local minima but none of them are "deceptive" so the algorithm can forego exploration and focus on exploitation. The results are shown in Table 14.

Tournament size	Average fitness	Standard deviation
1	0.8596	0.9068
5	0.0233	0.0157
10	0.0435	0.0138
20	0.0238	0.0135
30	0.0206	0.0164
40	0.0124	0.0084
50	0.0188	0.0188

Table 14: Diferent tournament sizes with mutation rate 0.2 and crossover rate 0.7

Compare different objective functions All the results are shown in Figure 25 with mutation rate 0.2, crossover rate 0.7. The results are comparable for all functions taking into account that different functions have different characteristics, the used parameters are not optimized for each function and the optimization process is given a small number of generations.



(a) Ackley function

(b) Rastringin function

(c) Hypersphere function

(d) Rosenbrock function

Figure 25: Comparison of different objective functions for the GA algorithm

7.2 Evolution Strategies

Evolution Strategies are a class of metaheuristics that are similar to genetic algorithms that evolve the mutation rate together with the solution.

Unless specified, the following parameters are used:

- Population size: 50
- Number of generations: 50
- Objective function: Ackley
- Strategy: None
- Number of simulations: 10

Offspring size We can see that the performance of the algorithm is better for larger offspring sizes. This is because the algorithm is able to explore the search space better but with the tradeoff of computational complexity. The results are shown in Table 15.

Offspring size	Average fitness	Standard deviation
50	0.0920	0.0699
100	0.0787	0.0303
200	0.0325	0.0238
300	0.0236	0.0099
400	0.0104	0.0108
500	0.0282	0.0131

Table 15: Different offspring sizes with mutation rate 0.2

Strategy We now test different strategy to evolve the mutation rate:

- None: the mutation rate is fixed
- Global: each individual evolves a single mutation rate (sphere)
- Individual: each individual has a mutation rate for each gene / dimension (axis-parallel ellipsoid)
- Correlated: each individual has a mutation rate for each gene and the interaction between every pair of genes (general ellipsoid)

We can see that the best performance is obtained with the global strategy probably because there is no skewed narrow valley to be found in the Ackley function that would require a more complex strategy. The results are shown in Table 16.

Strategy	Average fitness	Standard deviation
None	4.57e-02	1.91e-02
Global	1.41e-06	4.09e-07
Individual	1.75e-04	2.51e-04
Correlated	6.38e-02	1.10e-01

Table 16: Different strategies with offspring size 100

7.3 Particle Swarm Optimization

Particle Swarm Optimization is a population-based optimization technique that is inspired by the social behavior of flocks of birds. The algorithms aims at finding emerging social behaviours starting from few simple local rules. These rules are:

- each position in the environment is associated with a reward
- each particle has memory of the best position it visited
- each particle gets information from its neighbours

Population size / Generation ratio We compare the performances of the algorithm by changing the number of individuals in the population and the number of generations but keeping the ratio constant (so to have the same number of evaluations). We can see that the performance is better for larger populations and smaller number of generations. The results are shown in Table 17.

Population size	Generations	Average fitness	Standard deviation
20	125	7.99e-16	1.74e-15
40	62	1.02e-11	5.92e.12
60	41	7.53e-08	3.05e-08
80	31	4.30e-06	1.99e-06
100	25	4.05e-05	4.05e-05

Table 17: Different population sizes and generation numbers with mutation rate 0.2

Topology We now test different topologies for the algorithm:

- Ring: each particle gets information from the N closest particles
- Star: each particle gets information from all other particles

The best performances are obtained by the star topology because the each particle gets information by a larger number of particles. To confirm this trend, we can see that the performance of the ring topology increases with the number of neighbours. The results are shown in Table 18.

Topology	Average fitness	Standard deviation
Star	1.62e-09	1.16e-09
Ring 2	4.58e-04	3.67e-04
Ring 5	2.37e-07	3.24e-07
Ring 10	1-90e-08	2.31e-08
Ring 20	4.22e-09	2.24e-09
Ring 30	3.18e-09	1.97e-09
Ring 40	2.80e-09	1.71e-09

Table 18: Different topologies with population size 40 and generation number 62

8 Multi-Objective Optimization

Multi-objective optimization is characterized by the presence of multiple, possibly conflicting, objectives. This means that the optimal solution for any single objective doesn't necessarily correspond to the optimal solution for the other objectives. What we get instead is a set of solutions that are optimal in the sense that no other solution is better in all objectives. This set of solutions is called the Pareto front.

The state of the art in multi-objective optimization is the NSGA-II algorithm, which is a genetic algorithm that uses a non-dominated sorting approach to create fronts of solutions that are not dominated by any other solution. The algorithm then selects the best solutions from the fronts, ensuring that the population is diverse and that the Pareto front is well represented (maximimization of the crowding distance).

8.1 Kursawe Function

The Kursawe function is a simple test function for multi-objective optimization. It is defined as follows:

$$\begin{cases} f_1(x) = \sum_{i=1}^{n-1} \left[-10 \exp\left(-0.2 \sqrt{x_i^2 + x_{i+1}^2} \right) \right] \\ f_2(x) = \sum_{i=1}^{n} \left[|x_i|^{0.8} + 5 \sin(x_i^3) \right] \end{cases} \tag{1}$$

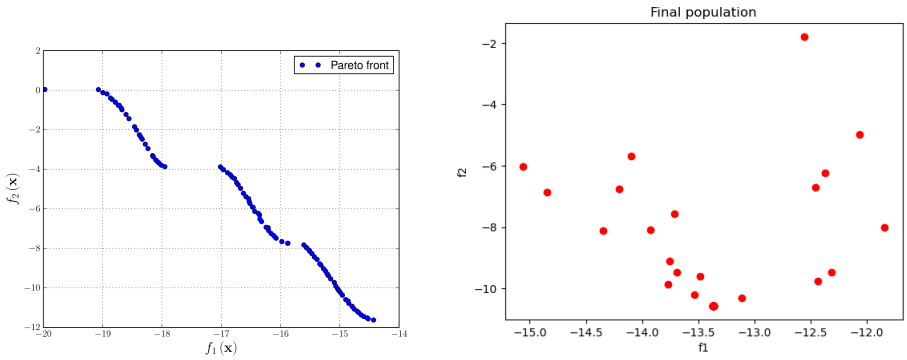
As a baseline, we use a simple Genetic Algorithm that combines the two objectives into a single one by taking a weighted sun of the two objectives in Table 19.

w1	w2	Fitness Mean	Fitness Std.	Best Fitness
1.0	0.0	(-19.986, 0.085)	(0.048, 0.041)	(-19.914, 0.018)
0.0	1.0	(-12.913, -10.672)	(0.500, 0.362)	(-13.010, -10.748)
0.5	0.5	(-13.358, -10.562)	(0.056, 0.045)	(-13.375, -10.557)
0.3	0.7	(-13.154, -10.708)	(0.049, 0.024)	(-13.152, -10.717)
0.7	0.3	(-19.909, 0.095)	(0.041, 0.021)	(-19.909, -0.0001)

Table 19: Genetic Algorithm results for the Kursawe function with different weights

We can see two different behavious depending on which one of the objectives is weighted more, but no solution is intrinsically better than the other.

Below we can compare the fitness of the final population of the Genetic Algorithm with the theoretical Pareto front of the Kursawe function.



(a) Theoretical Pareto front (b) Genetic Algorithm (0.3, 0.7) weights

Figure 26: Comparison of the theoretical Pareto front and the final population of the Genetic Algorithm

We also solve the Kursawe function using the NSGA-II algorithm. The algorithm is able to find a set of solutions that are close to the theoretical Pareto front. The final population, sorted in the fronts created by the algorithm, is shown below.

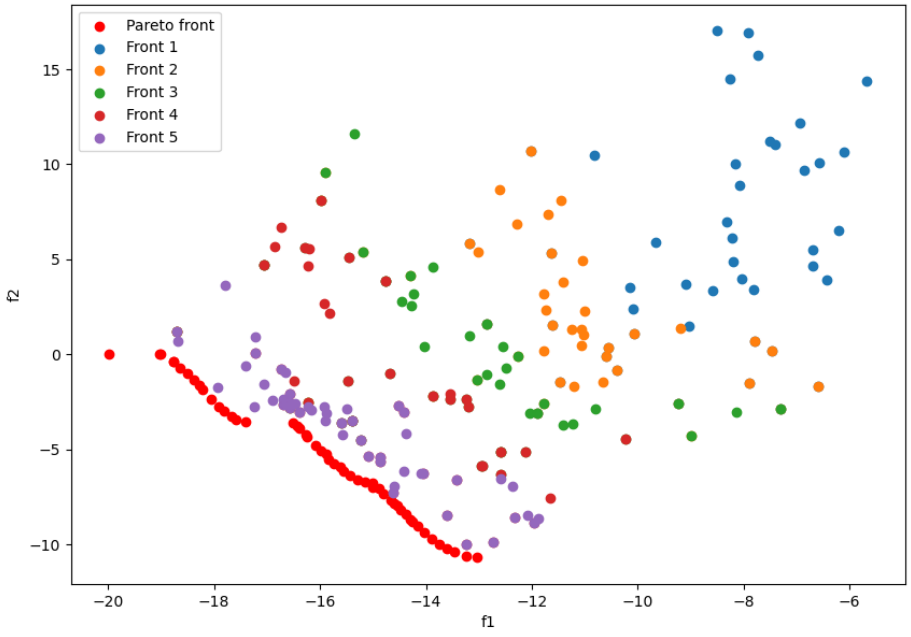


Figure 27: NSGA-II final population

Objective	Best Fitness Mean	Best Fitness Std.
f1	-19.9022	0.0521
f2	-10.5181	0.1668

Table 20: NSGA-II results for the Kursawe function

We can see how the simple genetic algorithm is able to find solutions that are close in one of the objectives to the NSGA-II results, but the final population doesn't resemble the Pareto front.

8.2 Multiple-Disk Clutch Brake Optimization

Real-world problem consisting of the optimization of five different parameters concerning the design of a multiple-disk clutch brake. The parameters are:

1. $r_i \in [60, 61, ..., 79, 80]mm$ - inner radius of the disks
2. $t_o \in [90, 91, ..., 109, 110]mm$ - outer radius of the disks
3. $F \in [600, 610, ..., 990, 1000]N$ - force applied to the disks
4. $t \in [1, 1.5, 2, 2.5, 3]mm$ - thickness of the disks
5. $Z \in [2, 3, 4, 5, 6, 7, 8, 9, 10]$ - number of disks

The two conflicting objectives are:

1. minimization of the break system mass
2. minimization of the stopping time

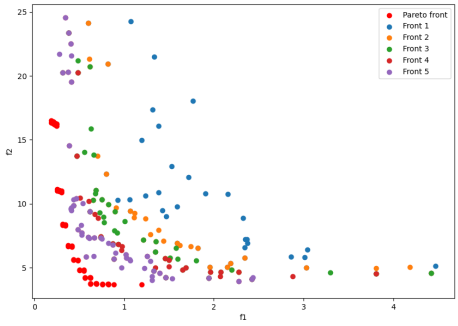
We also consider an additional situation (constrained) where the fitness value of an individual is penalized every time the constraints on the range of the parameters are violated.

Constrained	w1	w2	Fitness Mean	Fitness Std.	Best Fitness
False	1.0	0.0	(0.187, 20.63)	(8.32e-17, 3.249)	(0.187, 17.900)
False	0.0	1.0	(3.305, 3.378)	(0.935, 0.055)	(2.095, 3.317)
False	1.0	1.0	(0.624, 3.730)	(1.11e-16, 0.074)	(0.624, 3.730)
False	1.0	2.0	(0.624, 3.791)	(1.11e-16, 0.054)	(0.624, 3.731)
False	2.0	1.0	(0.624, 3.788)	(1.11e-16, 0.096)	(0.624, -3.731)
True	1.0	0.0	(0.187, 20.631)	(8.32e-17, 3.249)	(0.187, 17.900)
True	0.0	1.0	(3.305, 3.378)	(0.935, 0.055)	(2.095, 3.317)
True	1.0	1.0	(0.624, 3.738)	(1.11e-16, 0.074)	(0.624, 3.730)
True	1.0	2.0	(0.624, 3.791)	(1.11e-16, 0.054)	(0.624, 3.731)
True	2.0	1.0	(0.624, 3.788)	(1.11e-16, 0.096)	(0.624, -3.731)

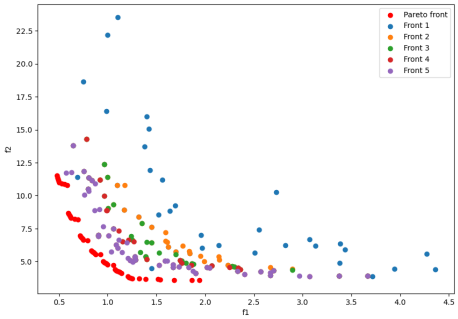
Table 21: Genetic Algorithm results for the Kursawe function with different weights

We can see that the results for the constrained and uncostrained cases are basically the same, which means that the constraints are never violated. We can also observe that the first objective is much simpler to optimize given how low the standard deviation is compared to the second objective.

As for NSGA-II, we can see that the achieved Pareto front has slightly better results in the constrained case specifically in terms of crowding distance and optimization of the second objective. The final population divided into fronts for both scenarios is shown below.



(a) NSGA-II final population



(b) NSGA-II final population (constrained)

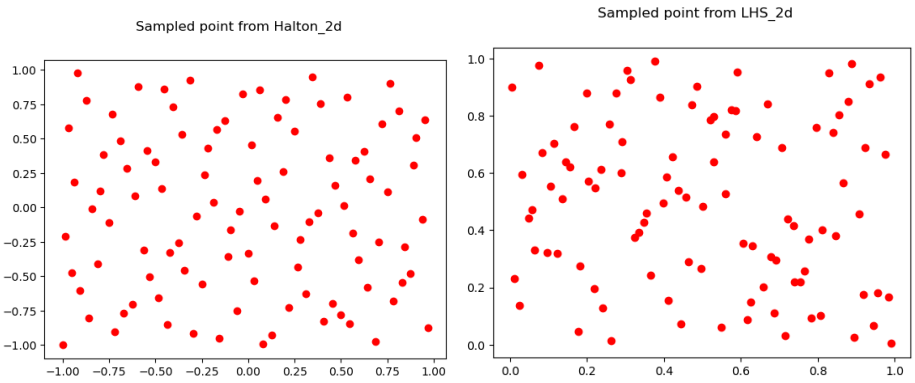
Constrained	Objective	Best Fitness Mean	Best Fitness Std.
False	f1	-0.1874	8.32e-17
False	f2	3.4276	0.1117
True	f1	0.4900	0.0260
True	f2	0.0853	0.0853

Table 22: NSGA-II results for the Kursawe function

9 Design of Experiments

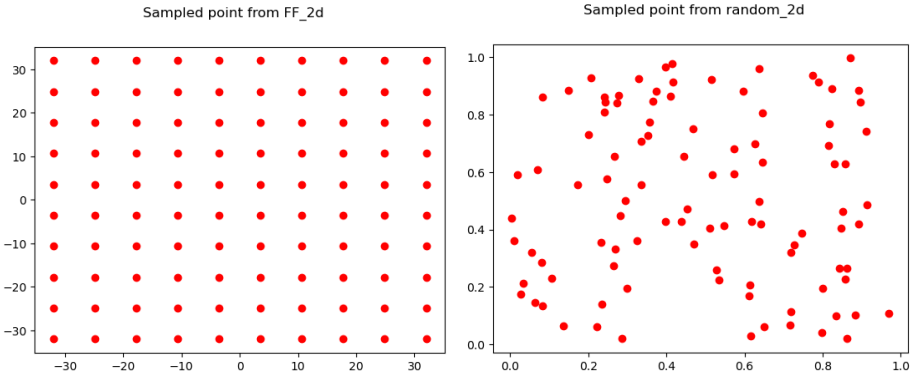
In this laboratory we'll compare different DoE techniques in the context of a simple genetic algorithm. We'll compare the following techniques:

- Halton sequence: a low-discrepancy sequence that is used to generate points in a space-filling manner.
- Latin hypercube sampling: a technique that generates points in a space-filling manner, but with a more random distribution than the Halton sequence.
- Full factorial design: a technique that generates all possible combinations of levels for each factor.
- Random



(a) Halton sequence

(b) Latin hypercube sampling



(c) Full factorial design

(d) Random sampling

Figure 29: Comparison of different DoE techniques.

We can see that the methods don't differ much both in terms of total time and in term of mean and standard deviation of the results. The only exception is the full factorial which in some cases finds significantly better results (e.g. DeJong5) while in others (e.g. Rosenbrock, Rastrigin) performs significantly worse. In the Rastrigin function, for example, full factorial works so well because the point it samples is perfectly the global minimum 30

Function	Method	Mean	Std. Dev.	Time (s)
Ackley	Halton	0.7861	1.9925	14.56
	LHS	0.2753	0.5453	15.38
	Full Factorial	4.44e-16	0.0	16.25
	Random	0.3867	0.88809	16.25
Rastrigin	Halton	0.4721	0.5585	17.79
	LHS	0.6673	0.8143	20.66
	Full Factorial	0.0	0.0	20.80
	Random	0.8337	0.9970	20.94
Rosenbrock	Halton	0.0101	0.0142	17.37
	LHS	0.0519	0.0747	17.83
	Full Factorial	0.1336	0.1468	18.61
	Random	0.0680	0.0948	18.48
DeJong5	Halton	3.0290	2.8565	46.83
	LHS	4.2475	3.3345	48.84
	Full Factorial	0.9980	9.41e-05	47.08
	Random	4.7181	3.6863	48.31

Table 23: Results of the comparison of different DoE techniques.

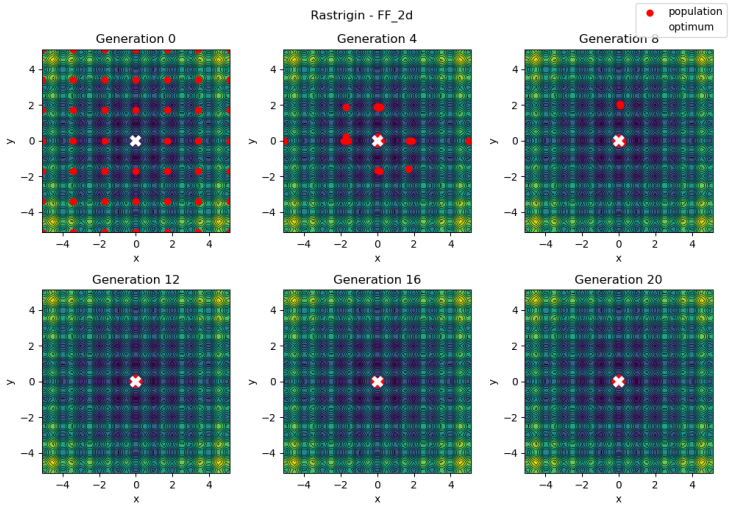


Figure 30: Rastrigin function with full factorial design.

10 Linear Programming

10.1 Exercise 1

A company makes two products (X and Y) using two machines (A and B). Each unit of X that is produced requires 50 minutes processing time on machine A and 30 minutes processing time on machine B. Each unit of Y that is produced requires 24 minutes processing time on machine A and 33 minutes processing time on machine B. At the start of the current week there are 30 units of X and 90 units of Y in stock. Available processing time on machine A is forecast to be 40 hours and on machine B is forecast to be 35 hours. The demand for X in the current week is forecast to be 75 units and for Y is forecast to be 95 units. Company policy is to maximise the combined sum of the units of X and the units of Y in stock at the end of the week.

$$\begin{aligned} \text{Maximize} \quad & X + Y \\ \text{Subject to} \quad & 50X + 24Y \leq 40 \times 60 \\ & 30X + 33Y \leq 35 \times 60 \\ & X + 30 \geq 75 \\ & Y + 90 \geq 95 \\ & X, Y \geq 0 \end{aligned} \tag{2}$$

Rewrited in slack form and simplified this is equivalent to:

$$\begin{aligned} \text{Minimize} \quad & -X - Y \\ \text{Subject to} \quad & 50X + 24Y + S_1 = 2400 \\ & 30X + 33Y + S_2 = 2100 \\ & -X + S_3 = -45 \\ & -Y + S_4 = -5 \\ & X, Y, S_1, S_2, S_3, S_4 \geq 0 \end{aligned} \tag{3}$$

The simplex method was used to solve this problem and the result was $X = 45$, $Y = 6.25$.

10.2 Exercise 2

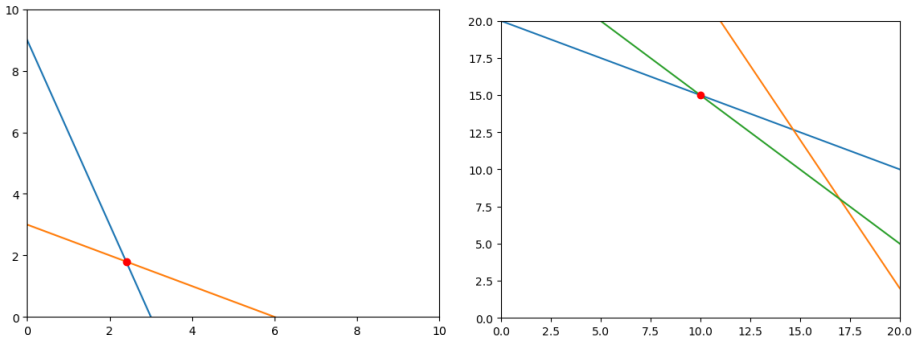
A factory manufactures chairs and tables, each requiring the use of three operations: Cutting, Assembly, and Finishing. The first operation can be used at most 40 hours; the second at most 42 hours; and the third at most 25 hours. A chair requires 1 hour of cutting, 2 hours of assembly, and 1 hour of finishing; a table needs 2 hours of cutting, 1 hour of assembly, and 1 hour of finishing. If the profit is 20 per unit for a chair and 30 for a table, how many units of each should be manufactured to maximize profit?

$$\begin{aligned} \text{Maximize} \quad & 20X_1 + 30X_2 \\ \text{Subject to} \quad & X_1 + 2X_2 \leq 40 \\ & 2X_1 + X_2 \leq 42 \\ & X_1 + X_2 \leq 25 \\ & X, Y \geq 0 \end{aligned} \tag{4}$$

Rewrited in slack form and simplified this is equivalent to:

$$\begin{aligned} \text{Minimize} \quad & -20X_1 - 30X_2 \\ \text{Subject to} \quad & X_1 + 2X_2 + S_1 = 40 \\ & 2X_1 + X_2 + S_2 = 42 \\ & X_1 + X_2 + S_3 = 25 \\ & X_1, X_2, S_1, S_2, S_3 \geq 0 \end{aligned} \tag{5}$$

The simplex method was used to solve this problem and the result was $X_1 = 10$, $X_2 = 15$.



(a) Feasible region for exercise 1 (b) Feasible region for exercise 2

Figure 31: Feasible regions

10.3 Exercise 3

A mutual fund has $ 100,000 to be invested over a three year horizon. Three investment options are available:

1. *Annuity*: the fund can pay a same amount of new capital at the beginning of each of three years and receive a payoff of 130% of total capital invested at the end of the third year. Once the mutual fund decides to invest in this annuity, it has to keep investing in all subsequent years in the three year horizon.

2. *Bank account*: the fund can deposit any amount into a bank at the beginning of each year and receive its capital plus 6% interest at the end of

that year. In addition, the mutual fund is permitted to borrow no more than \$20,000 at the beginning of each year and is asked to pay back the amount borrowed plus 6% interest at the end of the year. The mutual fund can choose whether to deposit or borrow at the beginning of each year.

3. *Corporate bond*: At the beginning of the second year, a corporate bond becomes available. The fund can buy an amount that is no more than \$50,000 of this bond at the beginning of the second year and at the end of the third year receive a payout of 130% of the amount invested in the bond.

The mutual fund's objective is to maximize total payout that it owns at the end of the third year.

$$\begin{array}{lll}
\max & 3.9X_1 + 1.06X_4 + 1.3X_5 & \text{3*1.3 annuity + 1.06 deposit Y3 + 1.3 bond} \\
\text{s.t.} & -X_1 - X_2 = -100K & \text{spend at most 100K in annuity and deposit} \\
& -X_1 + 1.06X_2 - X_3 - X_5 = 0 & \text{spend for annuity, bond and deposit, get capital Y1} \\
& -X_1 + 1.06X_3 - X_4 = 0 & \text{spend for annuity and deposit, get capital Y2} \\
& X_2, X_3, X_4 \geq -20K & \text{at most 20K deposit in bank} \\
& X_5 \leq 50K & \text{at most 50K for bond} \\
& X_1, X_5 \geq 0 &
\end{array} \tag{6}$$

where X_1 is the total amount to put in the annuity, X_2, X_3, X_4 are the bank deposit balances at the beginning of the three year and X_5 is the amount invested in the corporate bond.

The simplex method was used and the result is

$$X_1 = 24930, X_2 = 75070, X_3 = 4649, X_4 = -20000, X_5 = 50000 \tag{7}$$

11 Integer and Dynamic Programming

In this lab we solve combinational optimization problems using integer and dynamic programming.

11.1 N Queens

The N queens puzzle is the problem of placing N chess queens on an $N \times N (N \geq 4)$ chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal.

The constraints for the rows and columns are trivial to implement. The constraint for the diagonals can be implemented by checking if the sum fo the indexes between the row and column of two queens is the same (main diagonal) or the difference is the same (secondary diagonal). Below an example of the N queens problem for $N = 10$ is shown.

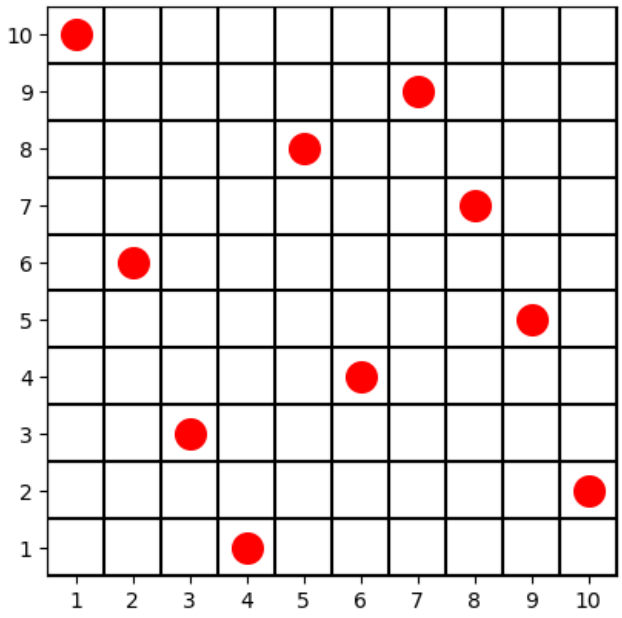


Figure 32: N queens problem for $N = 10$

11.2 TSP

The goal of the TSP is to find the shortest Hamiltonian cycle (a cycle that visits each node only once) on a graph of N nodes. The formulation for integer

programming is as follows:

$$
\begin{aligned}
&\min \sum_{i=1}^{N} \sum_{j=1}^{N} d_{ij} x_{ij} \\
&\text{s.t.} \sum_{j=1}^{N} x_{ij} = 1, \quad i = 1, \ldots, N && \text{one predecessor} \\
&\sum_{i=1}^{N} x_{ij} = 1, \quad j = 1, \ldots, N && \text{one successor} \\
&u_1 = 1 && \text{starting node} \\
&u_i - u_j + 1 \leq (N-1)(1 - x_{ij}), \quad i, j = 2, \ldots, N && \text{subtour elimination}
\end{aligned}
$$

where d_{ij} is the distance between node i and j, x_{ij} is a binary variable that is 1 if the path goes from node i to node j, and u_i is a auxiliary variable that represents the position of node i in the cycle.

Below an example of the TSP problem for $N = 10$ is shown.

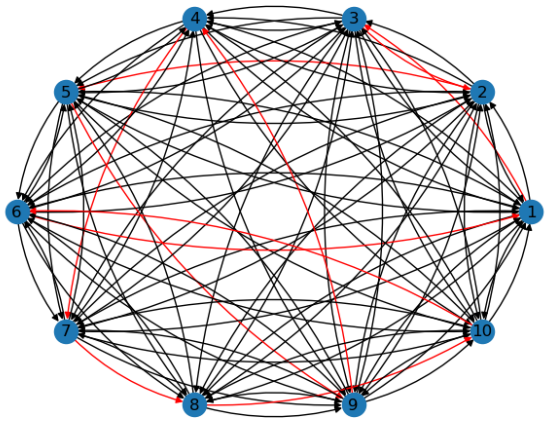


Figure 33: TSP problem for $N = 10$

11.3 Knapsack

The goal of the knapsack problem is to maximize the value of the items in the knapsack without exceeding the volume capacity. We'll solve it using dynamic programming and tabulation.

```
def knapsack(W, wt, val, n):
    DP = [[0 for _ in range(W+1)] for _ in range(n+1)]
    for i in range(n+1):
```

```
    for w in range(W+1):
        if i == 0 or w == 0: # base case
            DP[i][w] = 0
        elif wt[i-1] <= w:
            # decide if we take the item or not
            DP[i][w] = max(val[i-1] + DP[i-1][w-wt[i
                -1]], DP[i-1][w])
        else:
            # we can't take the item
            DP[i][w] = DP[i-1][w]
return DP[n][W] # return the maximum value
```

Example for the knapsack problem Given the following items:

Item	Volume	Value	
01	apple	2	1
02	pear	2	2
03	banana	2	2
04	watermelon	5	10
05	orange	2	3
06	avocado	2	3
07	bluberry	1	3
08	coconut	3	4
09	cherry	1	2
10	apricot	1	1

We compare the results for different knapsack capacities and we can see that the algorithm works as expected, achieving the maximum capacity for all cases except for $C = 20$.

Capacity	Achieved Capacity	Value	Items in the knapsack
5	5	9	05 06 07
10	10	16	05 06 07 08 09 10
15	15	20	01 02 03 05 06 07 08 09
20	16	21	01 02 03 05 06 07 08 09 10
25	25	25	01 02 03 04 05 06 07 08 09

12 Robust Optimization

12.1 Robust KnapSack Problem

In this problem we'll solve the classic Knapsack problem with an added uncertainty on the volumes of the items

Finite uncertainty set In general for finite sets, the only one that leads to a viable result is the one where the uncertainty budget is 0.

Ellipsoidal uncertainty set We can see that the probability of violation decreases as the uncertainty budget / ellipsiod radius increases. This is expected since the larger the uncertainty set, the more likely it is that the solution is feasible.

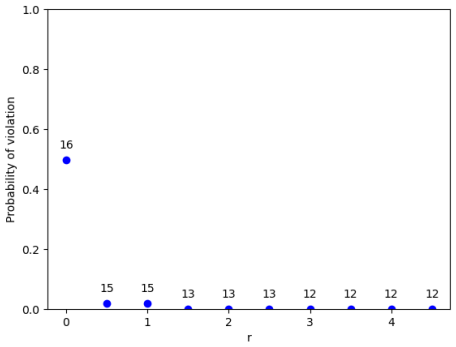
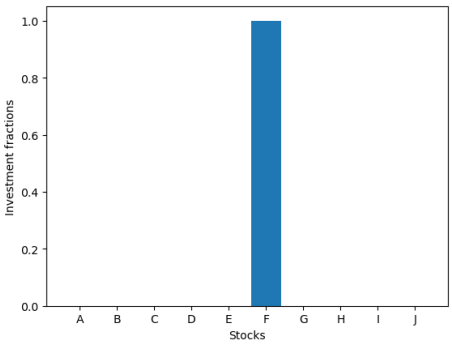


Figure 34: Probability of violation for different uncertainty budgets r

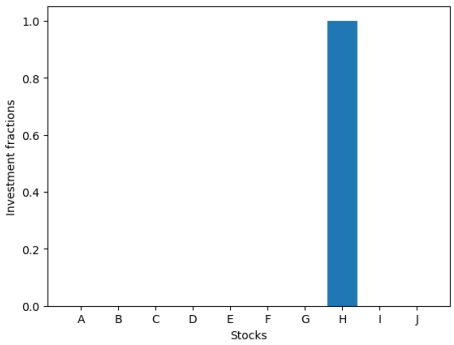
12.2 Robust Portfolio Optimization

In this problem we have a set of fictionary stocks, each one with a different mean and deviation on the return. We want to robustly maximize the return with a box uncertainty set.

Box uncertainty set The box uncertainty set is defined as a lower and upper bound. For almost all possible boxes, the algorithm always return stock F which is the one with the highest return. The only exception is for negative boxes, where the algorithm return stock H which is the one with the lowest deviation.

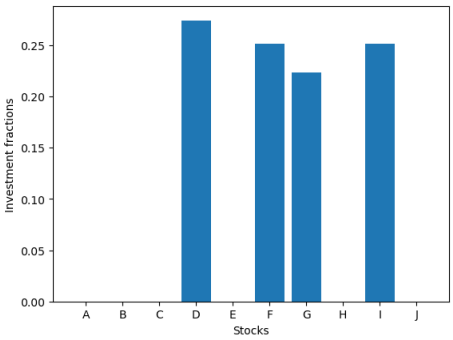


(a) Positive box uncertainty set

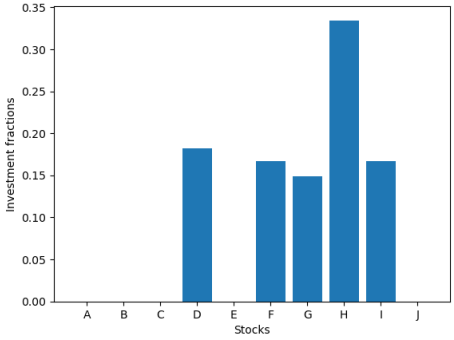


(b) Negative uncertainty set

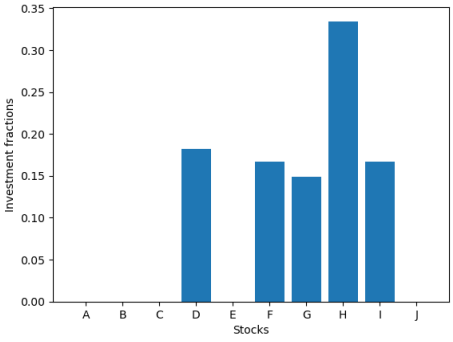
Norm uncertainty set We can see how the return decreases as the upper bound increases. This is expected since the algorithm is trying to maximize the return while keeping the uncertainty low.



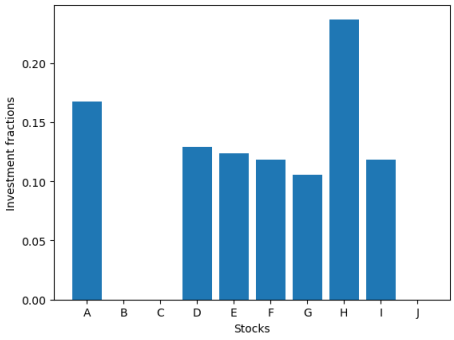
(a) Upper bound = 0.5



(b) Upper bound = 1.0



(c) Upper bound = 1.5



(d) Upper bound = 2.0

Upper bound	Objective value
0.5	1.0196
1.0	0.9963
1.5	0.9763
2.0	0.9602

Table 24: Results for different upper bounds