Indian Institute of Technology Guwahati
Department of Civil Engineering

CE 602 - Optimization Methods Term Project Report

Algorithms used

1. Ant Colony Optimization
2. Generalized Pattern Search Algorithm

Submitted by

1. Adil Nawab (224104102)
2. Brijesh Kumar Yadav (224104002)
3. Kiran S (226104026)
4. Mrigendra Singh Chauhan (224104003)

CONTENTS

Title **Page No.2**

CHAPTER 1: PROBLEM STATEMENT AND OBJECTIVE FUNCTION FORMULATION

CHAPTER 2: ANT COLONY OPTIMIZATION ALGORITHM

2.1 Introduction to Ant Colony Optimization Algorithm

2.2 Ant colony optimization: A metaheuristic

2.3 Methodology of Ant Colony Optimization

2.3.1. Algorithm of ACO

2.3.2. Pseudocode of ACO

CHAPTER 3: PATTERN SEARCH ALGORITHM

3.1 Introduction

3.2 Methodology

3.2.1. Algorithm for Pattern Search

3.3. Description

3.4. Options For Pattern Search

CHAPTER 4: MATLAB CODE FOR PATTERN SEARCH, RESULTS & INFERENCES

CHAPTER 1

PROBLEM STATEMENT AND OBJECTIVE FUNCTION FORMULATION

1.1.Description of Problem Statement & Objective Function

The reservoir is located on a non-perennial river and supplies water for irrigation purposes during the Rabi season. The crop calendar and cropping pattern for the study area is given in the tables below. The live storage capacity is 70.04 MCM, and dead storage capacity is 10 MCM. The Gross Commanded Area GCA is 14,222 and Culturable Command Area CCA is 9848 hectares. Ultimate Potential 12,507 hectares with an irrigation intensity of 127%.

Table 1: Crop Calendar

Sl. No	CROP	TIME DURATION	DAY
1	Gram	16 Oct - 28 Feb	135
2	Potato	1 Nov – 15 Feb	105
3	Vegetable	1 Nov – 31 Jan	90
4	Wheat	16 Nov – 15 Mar	120
5	Fodder	16 Oct – 15 Feb	120

Table 2: Cropping Pattern

Sl. No.	CROP	AREA
1	Gram	492
2	Potato	590
3	Vegetable	590
4	Wheat	5320
5	Fodder	197
	Total	7189

Table 3: Demand

CROPS\FORTNIGHT	FN1	FN2	FN3	FN4	FN5	FN6	FN7	FN8	FN9	FN10
Gram	0.20	0.19	0.24	0.33	0.42	0.43	0.50	0.47	0.36	
Fodder	0.08	0.08	0.10	0.13	0.16	0.16	0.18	0.18		
Potato		0.26	0.29	0.41	0.55	0.57	0.64	0.54		
Vegetable		0.37	0.39	0.47	0.53	0.50	0.21			
Wheat			1.70	2.64	4.55	5.10	5.97	6.13	4.99	3.75
Total	0.28	0.91	2.71	3.99	6.22	6.74	7.49	7.32	5.35	3.75

The difference between demand and release should be minimized. The variable here is the amount of water released and that should be to be minimized. The demand of each crop in is MCM for every fortnight is given below. The crop area is also for each crop has also been

provided. As water is released from the reservoir to fulfill the demand for the crops at every fortnight, the water level (given by storage) in the reservoir decreases. The inflow is considered to be zero in our case.

The objective function is formulated by the equation:

$$\boldsymbol{min\ f(x) = \sum_{nc=1}^{NC} \sum_{t=1}^{N} ky_t^{nc}\left(R_{t,nc} - D_{t,nc}\right)^2 + \sum_{t=1}^{N}\left(S_t(1 - B * e_t) - S_{t+1}(1 + B * e_t) + I_t - \sum_{nc=1}^{NC} R_{t,nc} - A_o * e_t\right)^2}$$

$R_{t,\,nc}$ = release in the period t = 1, 2..., T for crop nc = 1, 2, ..., NC (MCM)

$D_{t,\,nc}$ = demand to sustain the crop nc = 1, 2, NC in the period t=1, 2..., T (MCM)

ky_t^{nc} = yield response factor for time period t= 1, 2..., T of crop nc = 1, 2..., NC

S_t = Storage at time period t= 1, 2..., T (MCM)

S_{t+1}= Storage at time period t+1 (MCM)

I_t = Inflow at time period t= 1, 2..., T (MCM) (No inflow case considered here)

A_o and B = Regression constant correlating surface area (hectares) and storage value

e_t= Rate of evaporation at each fortnight in (mm)

The releases for irrigation should not exceed the maximum irrigation for all crops at all time steps.

$$0 \leq R_{t,\,nc} \leq D_{t,\,nc}$$

$R_{t,\,nc}$ is release for irrigation in the period t= 1, 2..., T of crop nc = 1,2,3..., NC

$D_{t,\,nc}$ is irrigation demand to sustain the crop nc = 1,2,3..., NC in the period t= 1, 2..., T

The reservoir storage in each fortnight should not be less than the dead storage, and should not be more than the live storage of the reservoir

$S_{min} \leq S_t \leq S_{max}$

S_{max}: Live storage of the reservoir in MCM.

S_{min}: Dead storage of the reservoir in MCM (Assumed to be 10 MCM).

The first part of the objective function contains the minimization of the difference between release and demand for all crops at all time steps and the second part has the constraints in the form of continuity equation. The constraints have been incorporated into the objective function itself and considered as a heuristic optimization problem. A total of 48 variables are considered in the problem. Variables x1 to x38 indicate the release for each fortnight and variable x39 to x48 indicated the storage in the reservoir at each fortnight after release.

For all the variables associated with the release of water, the lower bound will be zero and the upper bound will be the maximum demand. For the storage variables, the lower limit will be the dead storage in the reservoir and the upper limit will be the live storage.

Release variables for each fortnight										
CROPS\FORTNIGHT	**FN1**	**FN2**	**FN3**	**FN4**	**FN5**	**FN6**	**FN7**	**FN8**	**FN9**	**FN10**
Gram	x1	x2	x3	x4	x5	x6	x7	x8	x9	
Fodder	x10	x11	x12	x13	x14	x15	x16	x17		
Potato		x18	x19	x20	x21	x22	x23	x24		
Vegetable		x25	x26	x27	x28	x29	x30			
Wheat			x31	x32	x33	x34	x35	x36	x37	x38
Storage Variables for storage left at the end of each fortnight										
	x39	x40	x41	x42	x43	x44	x45	x46	x47	x48

Terms, using regression constants for each fortnight			
FORTNIGHT	**(1-B*et)**	**(1+B*et)**	**(A_0) *et**
FN1	0.99325	1.00675014	0.17028
FN2	0.994113	1.00588675	0.1485
FN3	0.994783	1.00521723	0.1316106
FN4	0.994806	1.005194468	0.1310364
FN5	0.994404	1.005595552	0.1411542
FN6	0.99455	1.005449561	0.1374714
FN7	0.993624	1.006375743	0.1608354
FN8	0.993448	1.006552345	0.1652904
FN9	0.993627	1.006373388	0.160776
FN10	0.992151	1.007849	0.198

Ky values						
Fortnight		**Gram**	**Fodder**	**Potato**	**Vegetable**	**Wheat**
FN 1	OCT II	0.2	0.2			
FN 2	NOV I	0.317	0.25	0.45	1.1	
FN 3	NOV II	0.608	0.375	0.43	1.01	0.2
FN 4	DEC I	0.813	0.525	0.38	0.83	0.32
FN 5	DEC II	0.8	0.581	0.393	0.65	0.52
FN 6	JAN I	0.666	0.544	0.515	0.43	0.6
FN 7	JAN II	0.388	0.515	0.585	0.2	0.6
FN 8	FEB I	0.25	0.5	0.388		0.575
FN 9	FEB II	0.2				0.545
FN 10						0.5

CHAPTER 2

ANT COLONY OPTIMIZATION

2.1. Introduction

Ants communicate to one another by laying down pheromones along their trails, so where ants go within and around their ant colony is a stigmaria system. In many ant species, ants walking from or to a food source, deposit on the ground a substance called pheromone. Other ants are able to smell this pheromone, and its presence influences the choice of their path, that is, they tend to follow strong pheromone concentrations. The pheromone deposited on the ground forms a pheromone trail, which allows the ants to find good sources of food that have been previously identified by other ants. Ant behavior is shown in Fig. 1. Using random walks and pheromones within a ground containing one nest and one food source, the ants will leave the nest, find the food and come back to the nest. After some time, the way being used by the ants will converge to the shortest path.

a) Ants in a pheromone trail between nest and food;
b) An obstacle interrupts the trail;
c) Ants find two paths to go around the obstacle;
d) A new pheromone trail is formed along the shorter path.

2.2. Ant colony optimization: A metaheuristic

The term metaheuristic is a combination of two Greek words. Heuristic derives from the verb heuriskein which means “to find”, while the suffix Meta means “beyond, in an upper level”.

The new heuristic has the following desirable characteristics:

• **Versatile:** It can be applied to similar versions of the same problem; for example, there is a straightforward extension from the traveling salesman problem (TSP) to the asymmetric traveling salesman problem (ATSP).

• **Robust:** It can be applied with only minimal changes to other combinatorial optimization problems such as the quadratic assignment problem (QAP) and the job-shop scheduling problem (JSP).

• **Population based approach:** This is interesting because it allows the exploitation of positive feedback as a search mechanism. It also makes the system amenable to parallel implementations.

Ant system is the first member of ACO class of algorithms. This algorithm is inspired by the trail laying and following behavior of natural ants. The essential trait of ACO algorithms is the combination of a priori information about the structure of a promising solution with posterior information about the structure of previously obtained good solutions. The main underlying idea, loosely inspired by the behavior of real ants, is that of a parallel search over several constructive computational threads based on local problem data and on a dynamic

memory structure containing information on the quality of previously obtained result. Extension of ACO algorithms to more complex optimization problems that include:

a) **Dynamic problems:** in which the instance data, such as objective function values, decision parameters, or constraints, may change while solving the problem.
b) Stochastic problems: in which one has only probabilistic information about objective function value(s), decision variable values, or constraint boundaries, due to uncertainty, noise, approximation, or other factors.
c) Multiple objective problems: in which a multiple objective function evaluates competing criteria of solution quality. Active research directions in ACO include also the effective parallelization of ACO algorithms and, on a more theoretical level, the understanding and characterization of the behavior of ACO algorithms while applying it to any proposed system.

Sl. No.	Advantages	Disadvantages
1	Inherent parallelism	Theoretical analysis is difficult
2	There is no central control in the colony	Sequences of random decisions (not independent)
3	Positive Feedback accounts for rapid discovery of good solutions	Probability distribution changes by iteration
4	Efficient for Traveling Salesman Problem and similar problems	Time to convergence uncertain (but convergence is guaranteed)
5	Can be used in dynamic applications (adapts to changes such as new distances, etc.)	Research is experimental rather than theoretical

Successful Applications of ACO Algorithm are shown below

Sl. No.	Problem Type	Problem Name
1	Routing	i. Traveling salesman (TSP) ii. Vehicle routing iii. Sequential ordering iv. Connection-oriented network routing v. Connectionless network routing vi. Optical network routing
2	Assignment	i. Quadratic assignment ii. Course timetabling iii. Graph coloring iv. Frequency Assignment
3	Scheduling	i. Project scheduling ii. Car sequencing
4	Subset	i. Set covering ii. Multiple knapsack iii. Maximum clique
5	Machine learning	i. Classification rules ii. Bayesian networks iii. Neural networks

		iv.	Fuzzy Systems
6	Bioinformatics	i. ii.	Protein folding DNA Sequencing

2.3. Methodology of Ant Colony Optimization

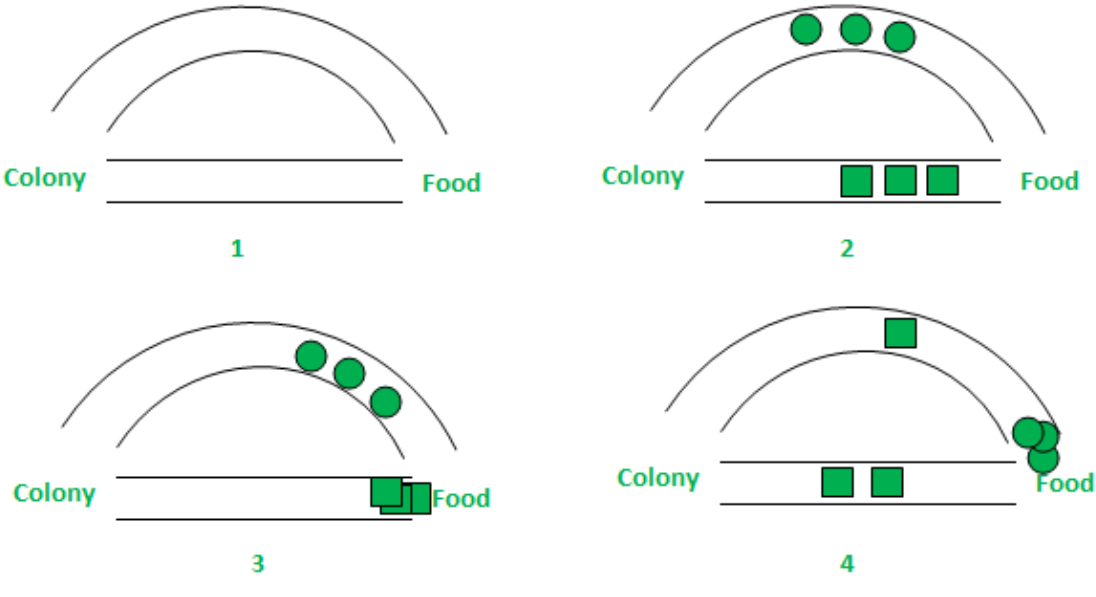

Fig.1. Schematic of Ant Colony Algorithm

In the above figure, for simplicity, only two possible paths have been considered between the food source and the ant nest. The stages can be analyzed as follows:

Stage 1
All ants are in their nest. There is no pheromone content in the environment. (For algorithmic design, residual pheromone amount can be considered without interfering with the probability)

Stage 2
Ants begin their search with equal (0.5 each) probability along each path. Clearly, the curved path is the longer and hence the time taken by ants to reach food source is greater than the other.

Stage 3
The ants through the shorter path reaches food source earlier. Now, evidently, they face with a similar selection dilemma, but this time due to pheromone trail along the shorter path already available, probability of selection is higher.

Stage 4
More ants return via the shorter path and subsequently the pheromone concentrations also increase. Moreover, due to evaporation, the pheromone concentration in the longer path reduces, decreasing the probability of selection of this path in further stages. Therefore, the whole colony gradually uses the shorter path in higher probabilities. So, path optimization is attained.

2.3.1Algorithmic Design

Pertaining to the above behavior of the ants, an algorithmic design can now be developed. For simplicity, a single food source and single ant colony have been considered with just two paths of possible traversal. The whole scenario can be realized through weighted graphs where the ant colony and the food source act as vertices (or nodes); the paths serve as the edges and the pheromone values are the weights associated with the edges.

Let the graph be **G = (V, E)** where V, E are the edges and the vertices of the graph. The vertices according to our consideration are $\mathbf{V_s}$ (Source vertex – ant colony) and $\mathbf{V_d}$ (Destination vertex – Food source), The two edges are $\mathbf{E_1}$ and $\mathbf{E_2}$ with lengths $\mathbf{L_1}$ and $\mathbf{L_2}$ assigned to each. Now, the associated pheromone values (indicative of their strength) can be assumed to be $\mathbf{R_1}$ and $\mathbf{R_2}$ for vertices E_1 and E_2 respectively. Thus, for each ant, the starting probability of selection of path (between E_1 and E_2) can be expressed as follows:

$$P_i = R_i / R_1+R_2;\ i = 1, 2\ldots\ldots$$

Evidently, if $R_1>R_2$, the probability of choosing E_1 is higher and vice-versa. Now, while returning through this shortest path say E_i, the pheromone value is updated for the corresponding path. The updation is done based on the length of the paths as well as the evaporation rate of pheromone. So, the update can be step-wise realized as follows:

i. **In accordance to path length**

$$\mathbf{R_i \leftarrow R_i + K / L_i}$$

In the above updation, i = 1, 2 and 'K' serves as a parameter of the model. Moreover, the update is dependent on the length of the path. Shorter the path, higher the pheromone added.

ii. **In accordance to evaporation rate of pheromone**

$$R_i \leftarrow (1 - v) * R_i$$

The parameter 'v' belongs to interval (0, 1] that regulates the pheromone evaporation. Further, i = 1, 2. At each iteration, all ants are placed at source vertex V_s (ant colony). Subsequently, ants move from V_s to V_d (food source) following step 1. Next, all ants conduct their return trip and reinforce their chosen path based on step 2.

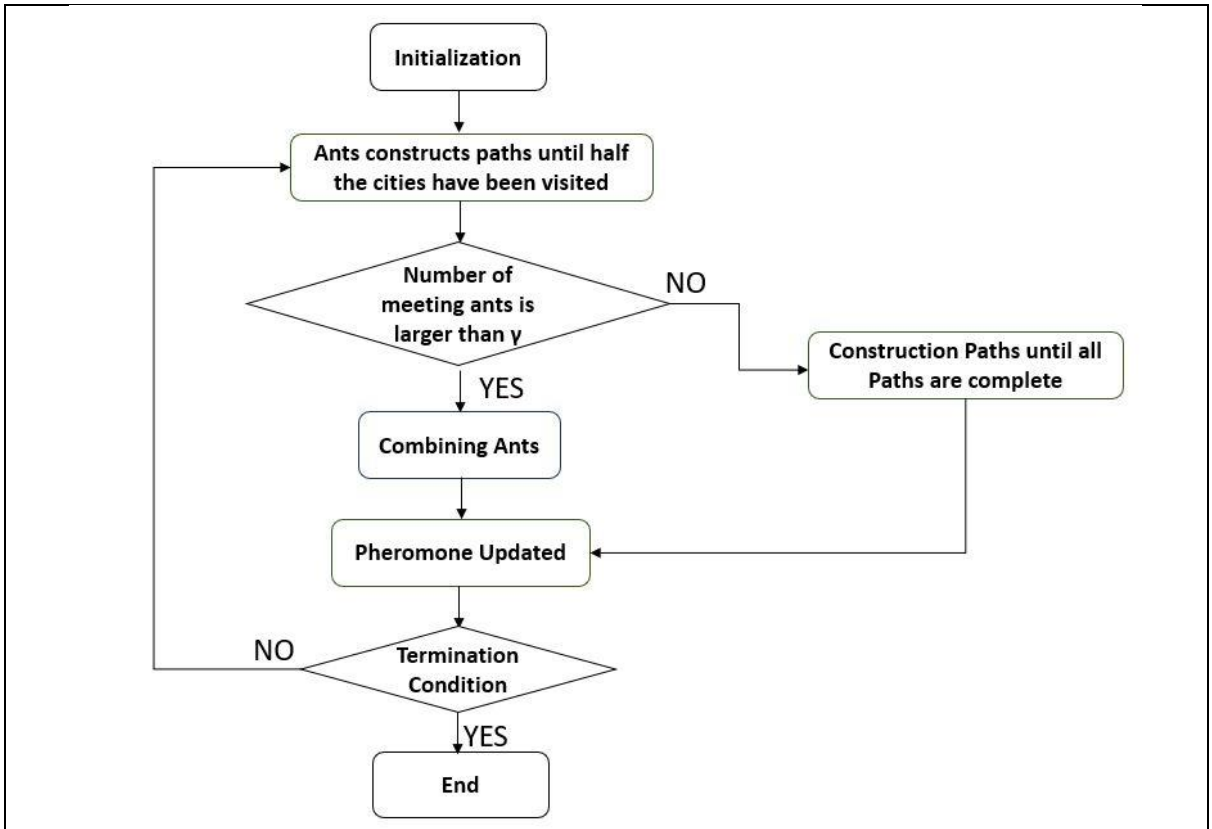

Fig.2. Flowchart of algorithm – Ant Colony Algorithm

2.3.2 Pseudocode

Procedure Ant Colony Optimization

Initialize necessary parameters and pheromone trials;
  while not termination **do**:
    Generate ant population;
    Calculate fitness values associated with each ant;
    Find best solution through selection methods;
    Update pheromone trial;
  end while
end procedure

NOTE:

Due to the unavailability of suitable modules for ACO, we have chosen the Generalized Pattern Search Algorithm for the work. Since the literature regarding ACO was completed, it was included in the report.

CHAPTER 3

PATTERN SEARCH ALGORITHM

3.1. Introduction

Both Evolutionary and Simplex search methods indicated some basic pattern while reaching closer to the optimum solution. However, these two methods or any previous methods did not explicitly exploit the pattern. Simplex search method attempted to exploit the pattern observed within the simplex by either expanding or contracting.

There are a few methods that can use the pattern of the objective function value to reach closer to the optimum solution. These methods are: Hooke-Jeeves pattern search method and Powell's conjugate direction method.

Pattern search (also known as direct search, derivative-free search, or black-box search) is a family of numerical optimization methods that does not require a gradient. As a result, it can be used on functions that are not continuous or differentiable. One such pattern search method is "convergence" (see below), which is based on the theory of positive bases. Optimization attempts to find the best match (the solution that has the lowest error value) in a multidimensional analysis space of possibilities. It uses a combination of exploratory moves and heuristic pattern moves. The exploratory moves help to create the search direction along which solutions can be explored within the search space. The exploratory moves are made in the vicinity of the current solution to identify two best points. These two points are used in the pattern moves. For a problem 'n' design or decision variables 'n' linearly independent search directions are required

Main features of algorithm:

- Derivative free, robust, simple, accurate, exhibits fastest convergence
- Operation requires tuning of few parameters and it is much simpler other soft computing techniques
- Used for non-linear, discontinuous and non-differentiable objective functions.

3.2. Methodology

Search phase
Evaluates the objective function at a finite set of mesh points and finds out the optimum solution. Current best point values are replaced by the new best value if optimum solution is achieved, else the process is repeated until the optimum solution is achieved.

Poll phase
When optimum solution not achieved. Moves to poll phase. Mesh points are updated with some new values. Objective function is evaluated based on new mesh points.

3.2.1. Algorithm For Pattern Search

Step 1

Set j = 0, and let Xj = (Xj,1, Xj, 2…, Xj, i…., Xj, n) ^T represent the start point, s_i step size,
$\alpha > 1$ step reduction factor and Ɛ Sufficiently small value for termination criteria.

Step 2

Perform exploratory move with Xj as the base point and step size s_i.
If exploratory move is a success, set Xj+1 = Xopt and go to Step 3;
Else if exploratory move is failure and $||s|| > Ɛ$, set $s_i = s_i/2$ for i = 1,2,
Repeat Step 2;

Else if
exploratory move is failure and $||s|| < Ɛ$, Terminate and report Xj as optimum solution and f(Xj) as optimum function value.

Step 3

Set j = j + 1 and perform pattern search in the direction of exploratory move, i.e., Xopt = Xj +λ° (Xj - Xj-1), such that f (X) is optimum. Set Xj = Xopt and go to Step 2.

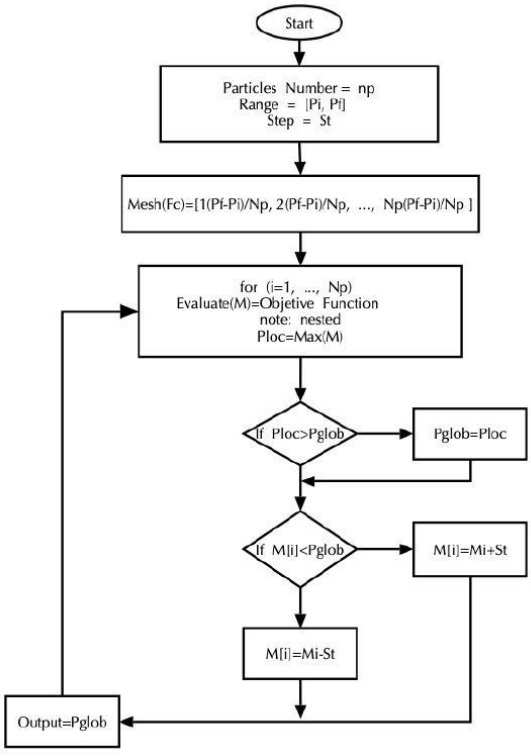

Fig.3. Flowchart of Algorithm

3.3.Description

x = patternsearch(fun,x0) finds a local minimum, x, to the function handle fun that computes the values of the objective function. x0 is a real vector specifying an initial point for the pattern search algorithm.

Note

Passing Extra Parameters explains how to pass extra parameters to the objective function and nonlinear constraint functions, if necessary.

x = patternsearch(fun,x0,A,b) minimizes fun subject to the linear inequalities A*x ≤ b. See Linear Inequality Constraints.

x = patternsearch(fun,x0,A,b,Aeq,beq) minimizes fun subject to the linear equalities Aeq*x = beq and A*x ≤ b. If no linear inequalities exist, set A = [] and b = [].

x = patternsearch(fun,x0,A,b,Aeq,beq,lb,ub) defines a set of lower and upper bounds on the design variables in x, so that the solution is always in the range lb ≤ x ≤ ub. If no linear equalities exist, set Aeq = [] and beq = []. If x(i) has no lower bound, set lb(i) = -Inf. If x(i) has no upper bound, set ub(i) = Inf.

x = patternsearch(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon) subjects the minimization to the nonlinear inequalities c(x) or equalities ceq(x) defined in nonlcon. patternsearch optimizes fun such that c(x) ≤ 0 and ceq(x) = 0. If no bounds exist, set lb = [], ub = [], or both.

x = patternsearch(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options) minimizes fun with the optimization options specified in options. Use optimoptions to set these options. If there are no nonlinear inequality or equality constraints, set nonlcon = [].

x = patternsearch(problem) finds the minimum for problem, a structure described in problem.

[x,fval] = patternsearch(__), for any syntax, returns the value of the objective function fun at the solution x.

[x,fval,exitflag,output] = patternsearch(__) additionally returns exitflag, a value that describes the exit condition of patternsearch, and a structure output with information about the optimization process.

3.4. Options for pattern search

Option	Description	Values
Algorithm	Algorithm used by patternsearch. The Algorithm setting affects the available options. For algorithm details, see How Pattern Search Polling Works and Nonuniform Pattern Search (NUPS) Algorithm.	{"classic"} | "nups" | "nups-gps" | "nups-mads"
Cache	With Cache set to "on", patternsearch keeps a history of the mesh points it polls. At subsequent iterations, patternsearch does not poll points close to those already polled. Use this option if patternsearch runs slowly while computing the objective function. If the objective function is stochastic, do not use this option. **Note** Cache does not work when you run the solver in parallel.	"on" | {"off"}

Option	Description	Values
CacheSize	Size of the history.	Positive scalar | {1e4}
CacheTol	Largest distance from the current mesh point to any point in the history in order for patternsearch to avoid polling the current point. Use if the Cache option is set to "on".	Positive scalar | {eps}
ConstraintTolerance	Tolerance on constraints. For an options structure, use TolCon.	Positive scalar | {1e-6}
Display	Level of display, meaning how much information patternsearch returns to the Command Line during the solution process.	"off" | "iter" | "diagnose" | {"final"}
FunctionTolerance	Tolerance on the function. Iterations stop if the change in function value is less than FunctionTolerance and the mesh size is less than StepTolerance. This option does not apply to MADS (mesh adaptive direct search) polling. For an options structure, use TolFun.	Positive scalar | {1e-6}
InitialMeshSize	Initial mesh size for the algorithm. See How Pattern Search Polling Works.	Positive scalar | {1.0}
InitialPenalty	Initial value of the penalty parameter. See Nonlinear Constraint Solver Algorithm.	Positive scalar | {10}
MaxFunctionEvaluations	Maximum number of objective function evaluations. For an options structure, use MaxFunEvals.	Positive integer | {"2000*numberOfVariables"}, where numberOfVariables is the number of problem variables
MaxIterations	Maximum number of iterations. For an options structure, use MaxIter.	Positive integer | {"100*numberOfVariables"}, where numberOfVariables is the number of problem variables
MaxMeshSize	Maximum mesh size used in a poll or search step. See How Pattern Search Polling Works.	Positive scalar | {Inf}
MaxTime	Total time (in seconds) allowed for optimization. For an options structure, use TimeLimit.	Positive scalar | {Inf}
MeshContractionFactor	Mesh contraction factor for an unsuccessful iteration. This option applies only when Algorithm is "classic". For an options structure, use MeshContraction.	Positive scalar | {0.5}
MeshExpansionFactor	Mesh expansion factor for a successful iteration. This option applies only when Algorithm is "classic". For an options structure, use MeshExpansion.	Positive scalar | {2.0}
MeshRotate	Flag to rotate the pattern before declaring a point to be optimum. See Mesh Options. This option applies only when Algorithm is "classic".	"off" | {"on"}
MeshTolerance	Tolerance on the mesh size. For an options structure, use TolMesh.	Positive scalar | {1e-6}

Option	Description	Values
OutputFcn	Function called by an optimization function at each iteration. Specify as a function handle or a cell array of function handles. For an options structure, use OutputFcns.	Function handle or cell array of function handles | {[]}
PenaltyFactor	Penalty update parameter. See Nonlinear Constraint Solver Algorithm.	Positive scalar | {100}
PlotFcn	Plots of output from the pattern search. Specify as the name of a built-in plot function, a function handle, or a cell array of names of built-in plot functions or function handles. For an options structure, use PlotFcns.	{[]} | "psplotbestf" | "psplotfuncount" | "psplotmeshsize" | "psplotbestx" | "psplotmaxconstr" | custom plot function
PlotInterval	Number of iterations for plots. 1 means plot every iteration, 2 means plot every other iteration, and so on.	positive integer | {1}
PollMethod	Polling strategy used in the pattern search. This option applies only when Algorithm is "classic". **Note** You cannot use MADS polling when the problem has linear equality constraints.	{"GPSPositiveBasis2N"} | "GPSPositiveBasisNp1" | "GSSPositiveBasis2N" | "GSSPositiveBasisNp1" | "MADSPositiveBasis2N" | "MADSPositiveBasisNp1"
PollOrderAlgorithm	Order of the poll directions in the pattern search. This option applies only when Algorithm is "classic". For an options structure, use PollingOrder.	"Random" | "Success" | {"Consecutive"}
ScaleMesh	Automatic scaling of variables. For an options structure, use ScaleMesh = "on" or "off".	{true}| false
SearchFcn	Type of search used in the pattern search. Specify as a name or a function handle. For an options structure, use SearchMethod.	"GPSPositiveBasis2N" | "GPSPositiveBasisNp1" | "GSSPositiveBasis2N" | "GSSPositiveBasisNp1" | "MADSPositiveBasis2N" | "MADSPositiveBasisNp1" | "searchga" | "searchlhs" | "searchneldermead" | "rbfsurrogate" | {[]} | custom search function
StepTolerance	Tolerance on the variable. Iterations stop if both the change in position and the mesh size are less than StepTolerance. This option does not apply to MADS polling. For an options structure, use TolX.	Positive scalar | {1e-6}
TolBind	Binding tolerance. See Constraint Parameters.	Positive scalar | {1e-3}
UseCompletePoll	Flag to complete the poll around the current point. See How Pattern Search Polling Works. This option applies only when Algorithm is "classic".	true | {false}

Option	Description	Values
	Note For the "classic" algorithm, you must set UseCompletePoll to true for vectorized or parallel polling. Similarly, set UseCompleteSearch to true for vectorized or parallel searching. Beginning in R2019a, when you set the UseParallel option to true, patternsearch internally overrides the UseCompletePoll setting to true so that the function polls in parallel. For an options structure, use CompletePoll = "on" or "off".	
UseCompleteSearch	Flag to complete the search around the current point when the search method is a poll method. See Searching and Polling. This option applies only when Algorithm is "classic". **Note** For the "classic" algorithm, you must set UseCompleteSearch to true for vectorized or parallel searching. For an options structure, use CompleteSearch = "on" or "off".	true | {false}
UseParallel	Flag to compute objective and nonlinear constraint functions in parallel. See Vectorized and Parallel Options and How to Use Parallel Processing in Global Optimization Toolbox. **Note** For the "classic" algorithm, you must set UseCompletePoll to true for vectorized or parallel polling. Similarly, set UseCompleteSearch to true for vectorized or parallel searching. Beginning in R2019a, when you set the UseParallel option to true, patternsearch internally overrides the UseCompletePoll setting to true so that the function polls in parallel. **Note** Cache does not work when you run the solver in parallel.	true | {false}
UseVectorized	Specifies whether functions are vectorized. See Vectorized and Parallel Options and Vectorize the Objective and Constraint Functions. **Note** For the "classic" algorithm, you must set UseCompletePoll to true for vectorized or parallel polling. Similarly,	true | {false}

Option	Description	Values
	set UseCompleteSearch to true for vectorized or parallel searching. For an options structure, use Vectorized = "on" or "off".	

CHAPTER 4
MATLAB Code:

```
%% Pattern Search with Bounds

% Find the minimum of a function that has only bound constraints.

%%
% Create the following n-variable objective function.
% On your MATLAB(R)path, save the following code to a file named |psobj.m|.
% <include>psobj.m</include>
% Set the objective function to |@psobj|.
fun = @psobj;
function y = psobj(x)
y = 0.2*(x(1)-0.202554432)^2+0.317*(x(2)-0.194338426)^2 ...
+0.608*(x(3)-0.235463328)^2+0.813*(x(4)-0.328210445)^2+0.8*(x(5)-0.42089616)^2+ ...
0.666*(x(6)-0.426311309)^2+0.388*(x(7)-0.498949402)^2+0.25*(x(8)-0.47117982)^2+ ...
0.2*(x(9)-0.357635981)^2+0.2*(x(10)-0.081104112)^2+0.25*(x(11)-0.082898831)^2+ ...
0.375*(x(12)-0.100664661)^2+0.525*(x(13)-0.132493187)^2+0.581*(x(14)-0.160103082)^2+ ...
0.544*(x(15)-0.155925894)^2+0.515*(x(16)-0.181293092)^2+0.5*(x(17)-0.181312299)^2+ ...
0.45*(x(18)-0.2648274)^2+0.43*(x(19)-0.286270603)^2+0.38*(x(20)-0.411937566)^2+ ...
0.393*(x(21)-0.553101783)^2+0.515*(x(22)-0.56529906)^2+0.585*(x(23)-0.64004439)^2+ ...
0.388*(x(24)-0.53932254)^2+1.1*(x(25)-0.37075836)^2+1.01*(x(26)-0.392957346)^2+ ...
0.83*(x(27)-0.473239944)^2+0.65*(x(28)-0.52996986)^2+0.43*(x(29)-0.499773526)^2+ ...
0.2*(x(30)-0.206623145)^2+0.2*(x(31)-1.69737792)^2+0.32*(x(32)-2.640582)^2+ ...
0.52*(x(33)-4.5511536)^2+0.6*(x(34)-5.09727288)^2+0.6*(x(35)-5.96578416)^2+ ...
0.575*(x(36)-6.12876768)^2+0.545*(x(37)-4.9894152)^2+0.5*(x(38)-3.75022494)^2+ ...
((70.04*0.99325)-(x(39)*1.00675014)-((x(1)+x(10))-0.17028))^2 +...
((x(39)*0.994113)-(x(40)*1.00588675)-((x(2)+x(11)+x(18)+x(25))-0.1485))^2 + ...
((x(40)*0.994783)-(x(41)*1.00521723)-((x(3)+x(12)+x(19)+x(26)+x(31))-0.1316106))^2 + ...
((x(41)*0.994806)-(x(42)*1.005194468)-((x(4)+x(13)+x(20)+x(27)+x(32))-0.1310364))^2 + ...
((x(42)*0.994404)-(x(43)*1.005595552)-((x(5)+x(14)+x(21)+x(28)+x(33))-0.1411542))^2 + ...
((x(43)*0.99455)-(x(44)*1.005449561)-((x(6)+x(15)+x(22)+x(29)+x(34))-0.1374714))^2 + ...
((x(44)*0.993624)-(x(45)*1.006375743)-((x(7)+x(16)+x(23)+x(30)+x(35))-0.1608354))^2 + ...
((x(45)*0.993448)-(x(46)*1.006552345)-((x(8)+x(17)+x(24)+x(36))-0.1652904))^2 + ...
((x(46)*0.993627)-(x(47)*1.006373388)-((x(9)+x(37))-0.160776))^2 + ...
((x(47)*0.992151)-(x(48)*1.007849)-((x(38))-0.198))^2;
lb =
[[0.202554432,0.194338426,0.235463328,0.328210445,0.42089616,0.426311309,0.498949402,0.471
17982, 0.357635981,0.081104112, 0.082898831, 0.100664661, 0.132493187,0.160103082,
0.155925894,0.181293092, 0.181312299,
0.2648274,0.286270603,0.411937566,0.553101783,0.56529906,0.64004439,0.53932254,0.37075836,
0.392957346,0.473239944,0.52996986,0.499773526,0.206623145,1.69737792, 2.640582,
4.5511536,5.09727288, 5.96578416, 6.12876768, 4.9894152, 3.75022494,0.283658544,
```

0.912823017, 2.712733858, 3.986463142,6.215224485, 6.744582668, 7.492694188, 7.320582339, 5.347051181, 3.75022494]]

```
ub =
[70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,7
0,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70]
A = [];
b = [];
Aeq = [];
beq = [];
x0 =
[[10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,
10,10,10,10,10,10,10,10,10,10,10,10,10,0,10,10]]
x = patternsearch (fun, x0, A, b, Aeq, beq, lb, ub)
optsc = optimoptions ("patternsearch", Algorithm="classic");
[sol, fval, eflag, output] = patternsearch (fun, x0, [], [], [], [], lb, ub, [], optsc)
options = optimoptions ("patternsearch", PlotFcn="psplotbestf");
```

***NOTE:** *The code was executed in MATLAB online due to issues in the system*

Results

The problem was solved using the pattern search tool in MATLAB.

1) When storage level is 70.04 MCM and no dead storage constraint was given

Columns 1 through 16 (x1 to x 16 L to R)
0.2188 0.3125 0.3750 0.5938 0.8125 1.0312 1.3750 1.3125 0.3750 0.1406 0.2109 0.3125
0.5625 0.7031 0.8750 0.8281

Columns 17 through 32 (x17 to x 32 L to R)
0.5781 0.3125 0.5000 0.9688 1.4375 1.3125 1.1562 1.0312 0.3906 0.5000 0.6875 1.0625
1.3750 1.8438 2.0625 3.2500

Columns 33 through 48 (x33 to x 48 L to R)
5.2500 5.7500 6.5000 6.3750 5.0000 3.8750 68.9062 67.0000 62.6250 55.8750 45.8750
34.8750 22.6250 13.0706 7.7500 4.0000
fval = 5.1529

2) When storage level is 65.04 MCM and no dead storage constraint was given

Columns 1 through 16 (x1 to x 16 L to R)
0.2500 0.2969 0.3438 0.4531 0.6562 0.8438 1.2188 1.1250 0.5625 0.1250 0.1797 0.2656
0.3438 0.4531 0.6562 0.7188

Columns 17 through 32 (x17 to x 32 L to R)
0.5156 0.3125 0.4219 0.7344 1.0312 1.1250 1.1250 0.9375 0.3906 0.4531 0.6250 0.8438
1.1875 1.5312 1.9375 3.0000

Columns 33 through 48 (x33 to x 48 L to R)
4.9375 5.6250 6.4375 6.3750 5.0000 3.8125 63.9531 62.1562 58.1875 52.5000 44.0000
34.0000 22.5000 13.3206 7.7500 4.0625
fval = 2.7528

3) When storage level is 48.04 MCM and no dead storage constraint was given

Columns 1 through 16 (x1 to x 16 L to R)

0.2026 0.1944 0.2355 0.3282 0.4209 0.4263 0.4990 0.4712 0.3577 0.0811 0.0829 0.1007
0.1325 0.1601 0.1559 0.1813

Columns 17 through 32 (x17 to x 32 L to R)
0.1813 0.2648 0.2863 0.4119 0.5531 0.5653 0.6401 0.5394 0.3708 0.3930 0.4733 0.5300
0.4998 0.2066 1.6974 2.6406

Columns 33 through 48.04 (x33 to x 48 L to R)
4.5513 5.0973 5.9658 6.1288 4.9895 3.7502 47.5164 46.4368 43.6263 39.5815 33.3447
26.6588 19.2852 12.1781 7.1265 3.7502
fval = 0.6132

4) When storage level is 40.04 MCM and no dead storage constraint was given

Columns 1 through 16(x1 to x16 L to R)
0.2026 0.1943 0.2355 0.3282 0.4209 0.4263 0.4990 0.4712 0.3576 0.0811 0.0829 0.1007
0.1325 0.1601 0.1559 0.1813

Columns 17 through 32 (x17 to x32 L to R)
0.1813 0.2648 0.2863 0.4119 0.5531 0.5653 0.6400 0.5393 0.3708 0.3930 0.4733 0.5300
0.4998 0.2066 1.6974 2.6406

Columns 33 through 48 (x33 to x 48 L to R)
4.5512 5.0973 5.9658 6.1288 4.9894 3.7502 40.3331 40.0567 38.0399 34.7881 29.3477
23.4564 16.8828 10.5763 6.3245 3.7502
fval =10.0191

Inferences

1. The problem was solved using the classic pattern search algorithm with bounds. The problem was an unconstrained problem. After solving the given problem, the objective function values for various cases of storages are shown in the results above.

2. The demands were kept as lower bound constraints so that once the storage level reaches below the demand level, the minimization of the objective function will not occur and the function value will start to increase.

3. This can also be seen in result 4 where the function value starts increasing once the storage falls below the demand (40.04 MCM)

4. For the other three cases when the storage is above the total demand, the minimization of the objective function takes place which can be seen in results 1 (70.04 MCM), 2 (65.04 MCM) and 3 (48.04 MCM)

5. We can notice that the demand ceases to be satisfied when the initial storage drops below 44 MCM.