ORMM

Release 0.0.1

Ethan Buck

Jun 03, 2021

CONTENTS

Operations Research Models & Methods (ORMM) is inspired by Paul A. Jensen's Excel Add-ins. His Excel packages were last updated in 2011, and while I believe they do still work (for the most part), his work may become outdated in a couple of ways:

- Excel is not as commonly used for OR, except in settings where security is of the utmost concern and/or modern languages like Python, R, Julia, C, C++, MATLAB, AMPL, or other modeling software are not available.
- From what I understand, Microsoft has been trying to phase out VBA and move to Javascript. If this happens, this could significantly impact whether or not his packages will work.
- His website and packages used to be available here, but currently I at least have not been able to load this webpage anymore - I'm not sure if UTexas took it down or not.

This python package aims to accomplish some of the same goals as Paul Jensen's website and add-ins did, mainly to

1. Be an educational tool that shows how abstract models (linear programs, integer programs, nonlinear programs, etc.) can be applied to real-life scenarios to solve complex problems.
2. Help the practitioner by providing modeling frameworks, methods for solving these models, and problem classes so a user can more easily see how they may be able to frame their business problem/objective through the lens of Operations Research.

This repository contains subpackages for grouping the different types of OR Models & Methods. Currently this subpackage list includes

1. `mathprog`: A subpackage for mathematical programs, including linear programs, mixed integer linear programs, nonlinear programs, and stochastic programs. Note for this subpackage that models and methods are not necessarily implemented in their abstract form, like Paul Jensen did - there are many python libraries that accomplish this task far better than I could (Pyomo, PuLP, GLPK to name a few). Thus, this subpackage here is dedicated to providing many problem classes, which show how these can be applied to real-life problems and provide an abstract/concrete model for that particular class of problems. Note that the abstract models can be built upon based on a unique business problem that may have more or fewer constraints, or a more complex objective to maximize/minimize.
2. *markov*: A subpackage for discrete state markov analysis. Currently this only has implementations for discrete time markov processes, but continous time will be added in the near future. This includes the main function *markov_analysis*, which returns a dictionary of the results, as well as a *print_markov* function. The main method requires a transition matrix, but can then run simulations, analyze steady state and transient probabilities, and run cost analyses if additional arguments are passed.

CHAPTER

ONE

INSTALLATION

```
$ pip install ormm
```

CHAPTER

TWO

EXAMPLES

The `mathprog` subpackage has multiple problem classes, as well as functions for printing the solution of a solved concrete model and for returning a pandas dataframe containing information for sensitivity analysis. Following are some examples of a few of these problem classes.

1. Resource Allocation: Optimize using scarce resources for valued activities.

```
from ormm.mathprog import resource_allocation
model = resource_allocation()
```

2. Blending Problem: Optimize the mixing of ingredients to satisfy requirements while minimizing cost.

```
from ormm.mathprog import blending
model = blending()
```

3. Employee Scheduling: Minimize the number of workers hired while meeting the minimum number of workers required for each period.

```
from ormm.mathprog import scheduling
model = scheduling(prob_class="employee")
```

4. Rental Scheduling: Minimize the cost of the plans purchased (which rent units for different amounts of time) while satisfying the number of units needed for each period.

```
from ormm.mathprog import scheduling
model = scheduling(prob_class="rental")
```

For more details on optional parameters and usage, see the *API Library Reference*. For more details on the MathProg problem descriptions, see the *Mathematical Programming*.

CHAPTER

THREE

DEVELOPER ENVIRONMENT

To use the same packages used in development (for creating additions / modifications), you may use the bash command below to install the dev requirements (recommended to do this in your virtualenv). This includes being able to run tests and add to the documentation.

```
$ pip install -e .[dev]
```

3.1 Mathematical Programming

3.1.1 Resource Allocation

The Resource Allocation Problem optimizes using scarce resources for valued activities. The resources are limited by some maximum quantity available, while the activities have some numeric value assigned to each of them. A matrix-like parameter shows all of the resources needed to conduct one unit of that activity (`ResourceNeeds`). This type of problem is seen often, with a few examples being production management and budget allocation.

Definitions

Sets

- `Activities` - Set of activities that are available to produce
  - `a in Activities` or $a \in A$
- `Resources` - Set of resources that are used to conduct activities
  - `r in Resources` or $r \in R$

Parameters

- `Values` - measure of value from conducting one unit of `Activity a`
  - `Values[a] for a in Activities` or $V_a \quad \forall a \in A$
- `ResourceNeeds` - amount of `Resource r` needed for `Activity a`
  - `ResourceNeeds[r, a] for r in Resources for a in Activities` or $N_{r,a} \quad \forall r \in R, a \in A$

Note: To conduct one unit of `Activity a`, you need all resources required. For example, to conduct one unit of `Activity a_1`, you need `sum(ResourceNeeds[r, a_1] for r in Resources)`

- `MaxResource` - maximum amount of units available of `Resource r`
  - `MaxResource[r] for r in Resources` or $M_r \ \forall r \in R$
- `MaxActivity` - maximum amount of demand for `Activity a`
  - `MaxActivity[a] for a in Activities` or $M_a \ \forall a \in A$

Decision Variables

- `NumActivity` - number of units to conduct of `Activity a`
  - `NumActivity[a] for a in Activities` or $X_a \ \forall a \in A$

Objective

Maximize total value of activities being conducted.

$$\text{Max} \sum_{a \in A} V_a X_a$$

Constraints

- An `Activity a` cannot be conducted more than its `MaxActivity`

$$0 \leq X_a \leq M_a \quad \forall a \in A$$

- To conduct 1 unit of an Activity, all `ResourceNeeds` are required. In other words, `sum(ResourceNeeds[r,a] for r in Resources)` must happen per `Activity a` conducted. This is implied by the problem parameters given by the user and the next constraint.
- The amount of resources used for a `Resource r` must not exceed `MaxResource[r]`

$$\sum_{a \in A} N_{r,a} X_a \leq M_r \quad \forall r \in R$$

API Reference

See the corresponding section in the *API Library Reference* to learn more about how to use the API for this problem class.

3.1.2 Blending Problem

The Blending Problem optimizes the mixing of ingredients to satisfy restrictions while minimizing cost. The restrictions are that certain properties of using the ingredients must be within their minimum and maximum values allowed. For example, ingredients in food may have nutrional properties, and this problem could force Calcium content of the mixture to be within a lower and upper bound (in terms of proportions). This problem deals with proportions, so it must another constraint to ensure that the sum of the decision variables (proportion of each ingredient to use) equals 1. A matrix-like parameter shows the numeric properties of each of the ingredients (`IngredientProperties`). This type of problem arises often in the food, feed, and oil refinement industries. The diet problem is a well-studied example of an application of this problem class.

Definitions

Sets

- `Ingredients` - Set of ingredients that are available for use
  - `i in Ingredients` or $i \in I$
- `Properties` - Set of properties that exist in the ingredients
  - `p in Properties` or $p \in P$

Parameters

- `Cost` - measure of cost of using one unit of `Ingredient i`
  - `Cost[i] for i in Ingredients` or $C_i \ \forall i \in I$
- `IngredientProperties` - measure of how much `Property p` is in `Ingredient i`
  - `IngredientProperties[i, p] for i in Ingredients for p in Properties` or $N_{i,p} \ \forall i \in I, p \in P$
- `MinProperty` - minimum amount of `Property p` needed in the blend
  - `MinProperty[p] for p in Properties` or $L_p \ \forall p \in P$
- `MaxProperty` - maximum amount of `Property p` allowed in the blend
  - `MaxProperty[p] for p in Properties` or $U_p \ \forall p \in P$

Decision Variables

- `Blend` - proportion of `Ingredient i` to include in the blend.
  - `Blend[i] for i in Ingredients` or $X_i \ \forall i \in I$

Objective

Minimize total cost of the ingredients in the blend.

$$\text{Min} \sum_{i \in I} C_i X_i$$

Constraints

- The Blend must have its Properties within the upper and lower bounds, `MinProperty[p]` and `MaxProperty[p]`.

$$L_p \leq \sum_{i \in I} N_{i,p} X_i \leq U_p \quad \forall p \in P$$

- The Blend decision variables are proportions of the ingredients to include, and thus, the decision variables must add up to 1. Additionally, these decision variables must all be greater than or equal to zero.

$$\sum_{i \in I} X_i = 1$$
$$X_i \geq 0 \ \forall i \in I$$

API Reference

See the corresponding section in the *API Library Reference* to learn more about how to use the API for this problem class.

3.1.3 Employee Scheduling Problem

The Employee Scheduling Problem minimizes the number of workers hired while satisfying the minimum number of workers required for each period. These constraints are commonly called the *covering constraints* in other scheduling problems. ORMM's current implementation of this problem class has limitations. It assumes that the worker works in "Shifts" that are in a row. For example, if the periods were days, and *ShiftLength* was set to 5, then it assumes that when a worker starts working on a Monday, then they will work Monday through Friday, and will be off on the weekend. This is because the decision variables in this implementation are interpreted as the number of workers that start their shift on *Period p*. This simplifies the model, and allows it to be an effective MILP. This type of problem occurs in management decisions where workers' schedules may not be constant, such as nurse scheduling or hourly staff.

Definitions

Sets

- `Periods` - An ordered set of periods when workers are needed
  - `p in Periods` or $p \in P$

Parameters

- `PeriodReqs` - measure of number of workers needed for `Period p`
  - `PeriodReqs[p] for p in Periods` or $R_p \ \forall p \in P$
- `ShiftLength` - measure of how many periods in a row a worker will work
  - `ShiftLength` or L

Decision Variables

- `NumWorkers` - number of workers that work a shift starting on `Period p`
  - `NumWorkers[p] for p in Periods` or $X_p \ \forall p \in P$

Objective

Minimize total number of workers (in the model, hired, etc.).

$$\text{Min} \sum_{p \in P} X_p$$

Constraints

- The number of workers that are working for each period must be greater than or equal to the minimum required - `PeriodReqs[p]`. Obtaining the number of workers that are present in each period requires using both the decision variables (what day a worker starts their shift) as well as the `ShiftLength` parameter. For example, if the `ShiftLength` is 2, and there are 10 workers that start Monday, 15 workers that start Tuesday, and 25 workers that start Wednesday, 40 workers would be present on Wednesday. If we are at the first period given by the data, the model has to go back to the last period given as well - in our example, this would be saying the number of workers present on Sunday (the beginning of the week) is the number of workers that start on Sunday plus the number of workers that start on Saturday (the end of the week). In mathematical terms, this can be represented by

$$\sum_{p-(L-1)}^{p} X_p \leq R_p \quad \forall p \in P$$

where P is a cyclically ordered set (or a cycle) and the start of the sum goes back $L - 1$ terms in that set.

- The decision variables must be greater than or equal to zero and integer.

$$X_p \geq 0, \text{int} \ \forall p \in P$$

API Reference

See the corresponding section in the *API Library Reference* to learn more about how to use the API for this problem class.

3.1.4 Rental Scheduling Problem

The Rental Scheduling Problem minimizes the cost of the plans purchased while satisfying the number of units needed for each period (the covering constraints). The structure of this problem is very similar to the employee problem, except for the addition of these plans, or options. These plans can have different effective number of periods, and different costs as a result. Note that the "plan length" would be pseudo-equivalent to the `ShiftLength` in the employee problem. Additionally, these plans may be restricted to only having certain start periods - for example, there may be a weekend plan that can only start on Saturday (and is effective Saturday and Sunday). This requires connecting the plans to the periods which they can start in, which is done through the `PlanToPeriod` parameter. This type of problem can arise when renting cars or other types of equipment.

Definitions

Sets

- `Periods` - An ordered set of periods when the units are needed
  - `p in Periods` or $p \in P$
- `Plans` - Set of plans that are available to rent units
  - `a in Plans` or $a \in A$
- `PlanToPeriod` - A set that describes which periods a plan can start in. This set is 2 dimensions, consisting of tuples (p, a). Note that not all combinations of (p, a) for $p \in P$, $a \in A$ may exist in this set.
  - `(period, plan) in PlanToPeriod` or $(p, a) \in J$ or $j \in J$

Parameters

- `PeriodReqs` - measure of number of units needed for `Period p`
  - `PeriodReqs[p] for p in Periods` or $R_p \quad \forall p \in P$
- `PlanCosts` - measure of the cost of reunting one unit under `Plan a`
  - `PeriodReqs[a] for p in Periods` or $C_a \quad \forall a \in A$
- `PlanLengths` - measure of how many periods in a row a `Plan a` is effective for
  - `PlanLengths[a] for a in Plans` or $L_a \quad \forall a \in A$

Decision Variables

- `NumRent` - number of units that are rented starting on `Period p` and under `Plan a`
  - `NumWorkers[(p,a)] for p in Periods for a in Plans` or $X_{(p,a)} \quad \forall (p, a) \in J$ or $X_j \quad \forall j \in J$

Objective

Minimize cost of the purchased plans. Note that we have to make sure that the combination of `Period p` and `Plan a` exists in `PlanToPeriod`, or $(p, a) \in J$.

$$\text{Min} \sum_{a \in A} C_a \sum_{p \in P \,|\, (p,a) \in J} X_{(p,a)}$$

Constraints

- The covering constraints require that there are enough units available in each `Period p`. To obtain the number of units available in each period, we need to use the decision variables `NumRent[(p,a)]` in combination with the plan lengths `PlanLengths[p]`. The number of units available in each period would be the sum of all of the `NumRent[(p,a)]` that are *effective* during the covering contraint's `Period p`. In other words, we have to look through all of the plans, and see which periods they can start in, and determine whether or not that combination symbol will be effective in the constraint's `Period p` based on the `PlanLengths[p]`. This *effective* condition will be represented by the math symbol f. In mathematical terms, these constraints can be represented by

$$\sum_{j \in J \,|\, f} X_j \geq R_p \quad \forall p \in P$$

where P is a cyclically ordered set (or a cycle).

- The decision variables must be greater than or equal to zero and integer.

$$X_j \geq 0, \text{int } \forall j \in J$$

API Reference

See the corresponding section in the *API Library Reference* to learn more about how to use the API for this problem class.

3.1.5 Aggregate Planning Problem

The Aggregate Planning Problem minimizes the production and holding costs while satisfying the demand of units for each period. The decisions to be made in this problem are how many units to produce in each period. Units that aren't sold (aka are over the demand) in one period can be held into the next, but with some holding cost per unit. While the holding cost is a single parameter, production costs per unit can vary from period to period.

This main constraints are the *conservation of flow* constraints, or that the leftover inventory plus the production minus the extra inventory equals the demand for every period. This type of problem can arise in manufacturing or in sales industries.

Definitions

Sets

- `Periods` - An ordered set of periods when the units are needed
  - `p in Periods` or $p \in P$

Parameters

- `Demand` - measure of number of units needed for `Period p`
  - `Demand[p] for p in Periods` or $D_p \ \ \forall p \in P$
- `Cost` - measure of cost of producing one unit within `Period p`
  - `Cost[p] for p in Periods` or $C_p \ \ \forall p \in P$
- `HoldingCost` - measure of the cost of holding one extra unit from one period to the next
  - `HoldingCost` or h
- `MaxStorage` - maximum number of units that can be held over from one period to the next
  - `MaxStorage` or m
- `InitialInv` - initial number of units in inventory, before the first period begins
  - `InitialInv` or I_I
- `FinalInv` - desired number of units in inventory to end up with, after the last period ends
  - `FinalInv` or I_F

Decision Variables

- `Produce` - number of units to produce in `Period p`
  - `Produce[p] for p in Periods` or $X_p \ \ \forall p \in P$
- `InvLevel` - number of units left in inventory at the end of `Period p`
  - `InvLevel[p] for p in Periods` or $Y_p \ \ \forall p \in P$

Objective

Minimize production cost and holding costs.

$$\text{Min} \sum_{p \in P} C_p X_p + h Y_p$$

Constraints

- The conservation of flow constraints enforce the relationships between the production, inventory levels, and the demand for each period. In mathematical terms, these constraints can be represented by

$$Y_{p-1} + X_p - Y_p = D_p \quad \forall p \in P$$

where Y_{p-1} is defined to be I_I when p is the first period.

- The amount stored at the end of each period cannot be more than the maximum amount allowed, m.

$$Y_p \leq m \quad \forall p \in P$$

- We define constraints to enforce the definition of Y_{p-1} when p is the first period, as well as the last period's inventory level to be I_F.

$$Y_{\min(P)-1} = I_I$$
$$Y_{\max(P)} = I_F$$

- The decision variables must be greater than or equal to zero and integer.

$$X_p,\, Y_p \geq 0, \text{int } \forall p \in P$$

API Reference

See the corresponding section in the *API Library Reference* to learn more about how to use the API for this problem class.

3.2 Markov Analysis

This subpackage provides methods for performing analyis on discrete-state Markovian processes. While some refer to a discrete time Markov process as a Markov Chain, in this package a Markov Chain refers to a discrete state Markov process. This is so that we can refer to them as continuous time or discrete time Markov Chains, since this package currently only considers discrete states.

A Markov process is a stochastic process that holds the Markovian Property, both of which are described in the definitions below.

1. Stochastic Process: A collection of random variables $\{X_t\}$, where t is a time index that takes values from a given set T ([1], P. 413).
2. Markovian Property: Given that the current state is known, the conditional probability of the next state is independent of the states prior to the current state ([1], P. 413).

These processes can be represented by labels for the possible states (or a state space S) and a transition matrix P. The transition matrix details the probabilities of moving from one state to another - thus, it must be of size $m \times m$, where m is the number of states in the state space S.

There are many ways to analyze a Markovian system, including simulation, transient probabilities, steady state probabilities, and cost analysis. The `analyze_dtmc` and `analyze_ctmc` methods can be passed dictionaries of key word arguments to add these details to the returned analysis.

The set T of periods needs to be finite for simulations or transient probability analyis, but steady state probabilities assess the distribution of the states in the long term. Note that steady state probabilities are the eigenvectors of the transition matrix P corresponding to the eigenvalue 1.

3.2.1 Discrete Time Analysis

In a Discrete Time Markov Chain, the set T is made up of discrete time intervals (ex: period 1, 2, 3, etc.). This works well for systems where events occur in steps or specified intervals of time. These could be months, years, weeks, days, hours, or other time interval choices.

[1] Jensen, P. A., & Bard, J. F. (2002). Operations Research Models and Methods. Wiley.

Usage Example

Consider an ATM located at a bank. Only one person can use the machine at a time, so additional arrivals must wait in a first come first serve queue. Suppose the parking lot is limited in size, and thus the system is limited to a maximum of 5 cars (4 waiting and 1 in service).

The code below will return a dictionary with the results of analyzing this system.

```
>>> from ormm.markov import analyze_dtmc, print_markov
>>> import numpy as np
>>>
>>> # Probability of bulb failing based on age of bulb in months
>>> prob = [0.5, 0.1, 0.1, 0.1, 0.2]
>>> cdf = np.cumsum(prob)
>>> cond_prob = [p / (1 - cdf[ind - 1])
>>>              if ind > 0 else p for ind, p in enumerate(prob)]
>>> cond_prob[-1] = 1  # roundoff error overriding
>>>
>>> # Create state space and transition probability arrays
>>> state_space = [0, 1, 2, 3, 4]
>>> num_states = len(state_space)
>>> num_bulbs = 1_000
>>> transition_matrix = np.zeros(shape=(num_states, num_states))
>>> transition_matrix[:, 0] = cond_prob
>>> for row in range(num_states - 1):
...     transition_matrix[row, row + 1] = 1 - transition_matrix[row, 0]
>>> transition_matrix
array([[0.5       , 0.5       , 0.        , 0.        , 0.        ],
       [0.2       , 0.        , 0.8       , 0.        , 0.        ],
       [0.25      , 0.        , 0.        , 0.75      , 0.        ],
       [0.33333333, 0.        , 0.        , 0.        , 0.66666667],
       [1.        , 0.        , 0.        , 0.        , 0.        ]])
>>>
>>> # Create cost parameters
>>> inspect_cost = 0.10
>>> replace_cost = 2
>>> inspect_vector = [inspect_cost] * num_states
>>> inspect_vector
[0.1, 0.1, 0.1, 0.1, 0.1]
>>> replace_matrix = np.array([[replace_cost] * num_states]
...                           + ([[0] * num_states] * (num_states - 1))).T
>>> replace_matrix
array([[2, 0, 0, 0, 0],
       [2, 0, 0, 0, 0],
       [2, 0, 0, 0, 0],
       [2, 0, 0, 0, 0],
       [2, 0, 0, 0, 0]])
>>>
>>> # Run Markov Analysis
>>> analysis = analyze_dtmc(transition_matrix, state_space,
...                         trans_kwargs={"ts_length": 12,
...                                       "init": [1, 0, 0, 0, 0]},
...                         cost_kwargs={"state": inspect_vector,
...                                      "transition": replace_matrix,
```

(continues on next page)

(continued from previous page)

```
...                                             "num": num_bulbs}
>>> print_markov(analysis)
CDFs:
[[0.5        1.         1.         1.         1.        ]
 [0.2        0.2        1.         1.         1.        ]
 [0.25       0.25       0.25       1.         1.        ]
 [0.33333333 0.33333333 0.33333333 0.33333333 1.        ]
 [1.         1.         1.         1.         1.        ]]

Steady State Probs:
[0.41666667 0.20833333 0.16666667 0.125      0.08333333]

Transient Probabilities (length 12)
Initial Conditions:
[1, 0, 0, 0, 0]
Output:
[[1.         0.         0.         0.         0.        ]
 [0.5        0.5        0.         0.         0.        ]
 [0.35       0.25       0.4        0.         0.        ]
 [0.325      0.175      0.2        0.3        0.        ]
 [0.3475     0.1625     0.14       0.15       0.2       ]
 [0.49125    0.17375    0.13       0.105      0.1       ]
 [0.447875   0.245625   0.139      0.0975     0.07      ]
 [0.4103125  0.2239375  0.1965     0.10425    0.065     ]
 [0.39881875 0.20515625 0.17915    0.147375   0.0695    ]
 [0.40385313 0.19940938 0.164125   0.1343625  0.09825   ]
 [0.42587719 0.20192656 0.1595275  0.12309375 0.089575  ]
 [0.42381203 0.21293859 0.16154125 0.11964563 0.0820625 ]
 [0.41682342 0.21190602 0.17035088 0.12115594 0.07976375]]

Cost kwargs:
{'state': [0.1, 0.1, 0.1, 0.1, 0.1], 'transition': array([[2, 0, 0, 0, 0],
       [2, 0, 0, 0, 0],
       [2, 0, 0, 0, 0],
       [2, 0, 0, 0, 0],
       [2, 0, 0, 0, 0]]), 'num': 1000}
Expected Steady State Cost:
$0.93
Expected Total Steady State Cost: $933.33
Expected Transient Cost:
[1.1        0.8        0.75       0.795      1.0825     0.99575
 0.920625   0.8976375  0.90770625 0.95175438 0.94762406 0.93364684
 0.92705939]
Expected Total Transient Cost: $12,009.30
```

3.2.2 Continuous Time Analysis

In a Continuous Time Markov Chain, the set T is made up of a continuous time interval. The Markovian property is only satisfied with a continuous time interval if all activity durations are exponentially distributed, such as an $M/M/s$ queuing system.

Usage Example

Consider a store that has a lit-up sign for displaying the name of the store to potential customers driving by. Assume that the sign has 1,000 of these special bulbs to make the sign brighter. Your boss may notice the bulbs burning out more than they expected, and wants to know if this was just by chance or if this behavior is expected in the long term future. They also would like an analysis on the expected cost of replacing these light bulbs.

The code below will return a dictionary with the results of this analysis.

```
>>> from ormm.markov import analyze_ctmc
>>> import numpy as np
>>>
>>> # Defining Parameters
>>> arrival_rate = 2  # per minute
>>> service_rate = 2.5  # per minute
>>> states = [0, 1, 2, 3, 4, 5]
>>> num_states = len(states)
>>> rate_matrix = []
>>> for row in states:
>>>     this_row = [arrival_rate if col == row + 1 else 0 for col in states]
>>>     if row >= 1:
>>>         this_row[row - 1] = service_rate
>>>     rate_matrix.append(this_row)
>>> rate_matrix = np.array(rate_matrix)
>>> d = 0.05  # small time interval, a parameter
>>> t = 1  # want to approx. transient probs at this time, a parameter
>>> n = int(t / d)  # number of steps - determines accuracy of approx.
>>> q_init = [1, 0, 0, 0, 0, 0]
>>>
>>> # Run Analysis
>>> analysis = analyze_ctmc(states=states, rate_matrix=rate_matrix,
>>>                         t=t, d=d, init=q_init)
>>> analysis
{'transition_rates': array([2. , 4.5, 4.5, 4.5, 4.5, 2.5]),
 'P': array([[0.9  , 0.1  , 0.   , 0.   , 0.   , 0.   ],
             [0.125, 0.775, 0.1  , 0.   , 0.   , 0.   ],
             [0.   , 0.125, 0.775, 0.1  , 0.   , 0.   ],
             [0.   , 0.   , 0.125, 0.775, 0.1  , 0.   ],
             [0.   , 0.   , 0.   , 0.125, 0.775, 0.1  ],
             [0.   , 0.   , 0.   , 0.   , 0.125, 0.875]]),
 'init': [1, 0, 0, 0, 0, 0],
 'transient': array([0.43253628, 0.29099199, 0.16203962, 0.0745227,
                     0.02883399, 0.01107543]),
 'generator_matrix': array([[ 1. ,  2. ,  0. ,  0. ,  0. ,  0. ],
                            [ 1. , -4.5,  2. ,  0. ,  0. ,  0. ],
                            [ 1. ,  2.5, -4.5,  2. ,  0. ,  0. ],
                            [ 1. ,  0. ,  2.5, -4.5,  2. ,  0. ],
```

(continues on next page)

(continued from previous page)

```
                              [ 1. ,  0. ,  0. ,  2.5, -4.5,  2. ],
                              [ 1. ,  0. ,  0. ,  0. ,  2.5, -2.5]]),
 'steady_state': array([0.2710556 , 0.21684448, 0.17347558, 0.13878047,
                         0.11102437, 0.0888195 ])}
```

3.3 Network Flow

3.3.1 Transportation Problem

The Transportation Problem minimizes the shipping costs while satisfying the demand at each destination. The decision variables are how many units at each source node will be shipped to each destination node. Each source node has a supply, which is the upper limit on how many units can be shipped from that node. Each destination node has a demand, which is the required amount at each destination node.

The main constraints ensure that all supply is used from the source nodes, and that all demand is met at the destination nodes. This type of problem arises often, with notable examples being supply chain management and online order shipments.

Definitions

Sets

- `Sources` - A set of nodes where the units are shipped from
  - `i in Sources` or $i \in I$
- `Destinations` - A set of nodes where the units are shipped to
  - `j in Destinations` or $j \in J$

Parameters

- `Supply` - measure of number of units available at `Source i`
  - `Supply[i] for i in Sources` or $S_i \ \forall i \in I$
- `Demand` - measure of number of units required at `Destination j`
  - `Demand[j] for j in Destinations` or $D_j \ \forall j \in J$
- `ShippingCost` - measure of the cost of shipping one unit from `Source i` to `Destination j`
  - `ShippingCost[i, j] for i in Sources for j in Destinations` or $C_{i,j} \ \forall i \in I, j \in J$

Decision Variables

- `Flow` - number of units to ship from `Source i` to :py:obj`:Destination j`
    - `Flow[i, j] for i in Sources for j in Destinations` or $X_{i,j} \ \forall i \in I, j \in J$

Objective

Minimize shipping costs from sources to destinations.

$$\text{Min} \sum_{i \in I} \sum_{j \in J} C_{i,j} X_{i,j}$$

Constraints

- The total supply must be equal to the total demand. Currently, this constraint must be met by the user changing their data. See the Notes section of the API docs for more details.
- All of the supply at each node must be shipped to the destination nodes.

$$\sum_{j \in J} X_{i,j} = S_i \quad \forall i \in I$$

- All of the demand at each node must be met by the source nodes.

$$\sum_{i \in I} X_{i,j} = D_j \quad \forall j \in J$$

- The decision variables must be greater than or equal to zero and integer.

$$X_{i,j} \geq 0, \text{int} \ \forall i \in I, j \in J$$

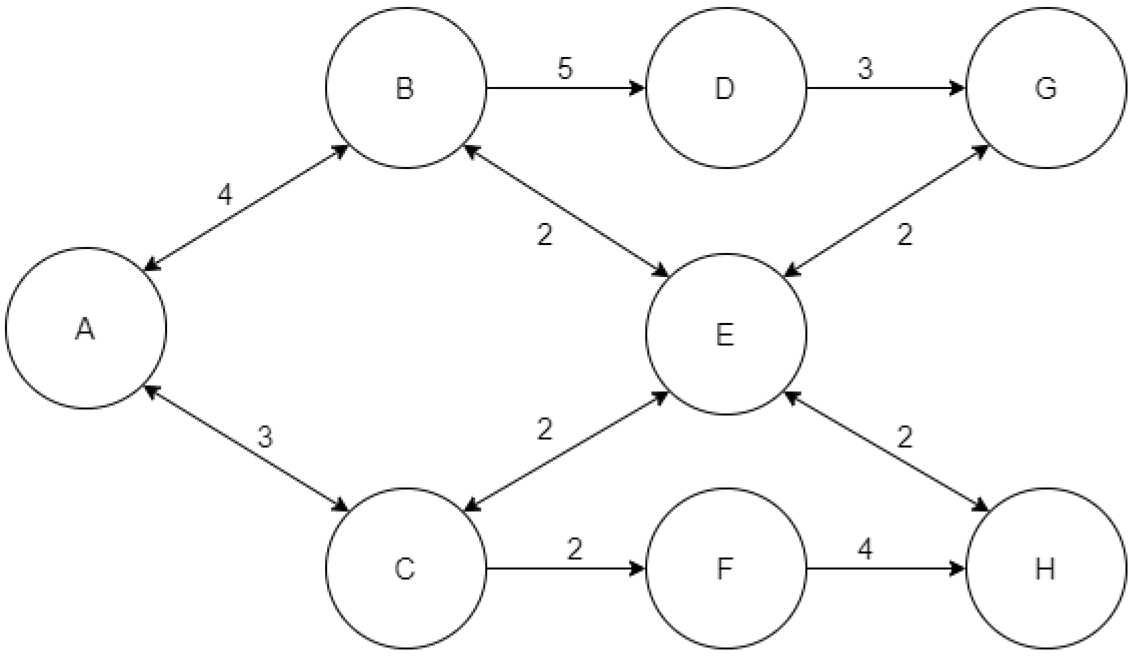
API Reference

See the corresponding section in the *API Library Reference* to learn more about how to use the API for this problem class.

3.3.2 Shortest Path Tree Problem

The Shortest Path Problem minimizes the weighted distance from one source node to one destination node. This module's implementation is actually the Shortest Path Tree Problem, which solves the Shortest Path Problem for every possible destination node from a single source node.

An example network is shown below. The arc lengths are shown as well, which can be interpreted as the cost of traveling an arc. Note that this graph is fully connected - there must be a possible path to every other node from the source node given.



Algorithm

Dijkstra's algorithm finds an optimal solution to the shortest path tree problem as long as all of the weighted distances (costs of traveling an arc) are nonnegative. This explanation of the algorithm is from Jensen and Bard's *Operations Research Models and Methods* ([1], P. 195), with some differences in notation and explanation.

Initially, let the set $\boldsymbol{S} = \{s\}$ and let $\alpha_s = 0$.

Repeat until $\boldsymbol{S}$ is the set of all nodes reachable by the source node:

> Find an arc $k(i, j)$ that passes from a solved node to an unsolved node such that:
>
> $$k(i,j) = \text{argmin}\{\alpha_{i'} + c_{k'} : k^{'}(i^{'}, j^{'}) \in \boldsymbol{A}, i^{'} \in \boldsymbol{S}, j^{'} \in \bar{\boldsymbol{S}}\}$$
>
> where $\bar{\boldsymbol{S}}$ is the complement of the set $boldsymbolS$. With the optimal arc $k(i, j)$ from above, add node j and arc $k(i, j)$ to the tree. Add node j to the solved set $\boldsymbol{S}$. Let $\alpha_j = \alpha_i + c_k$, where $k \equiv k(i, j)$.

API Reference

See the corresponding section in the *API Library Reference* to learn more about how to use the API for this problem class.

[1] Jensen, P. A., & Bard, J. F. (2002). Operations Research Models and Methods. Wiley.

3.4 API Library Reference

The current models are implemented under `ormm.mathprog`, `ormm.network`, and `ormm.markov`. MathProg contains factory methods to implement problem classes and other useful functions for solution analysis. Markov contains functions to perform markov analysis on discrete state and time processes, as well as printing those results nicely. Network contains models and methods for the transportation problem and the shortest path tree problem.

3.4.1 ORMM MathProg

blending([linear])	Factory method returning Pyomo Abstract/Concrete Model for the Blending Problem
resource_allocation([linear, mult_res, . . .])	Factory method returning Pyomo Abstract/Concrete Model for the Resource Allocation Problem.
scheduling(prob_class[, linear])	Calls factory methods for different scheduling problems.
print_sol(instance[, money_obj])	Print the solution to the solved *instance*.
sensitivity_analysis(instance)	Return dataframe containing sensitivity analysis

`ormm.mathprog.`**`blending`**(*linear=True*, ***kwargs*)

Factory method returning Pyomo Abstract/Concrete Model for the Blending Problem

Parameters

- **linear** (`bool`, optional) – Determines whether decision variables will be Reals (True) or Integer (False).
- ****kwargs** – Passed into Pyomo Abstract Model's *create_instance* to return Pyomo Concrete Model instead.

Notes

The Blending Problem optimizes the mixing of ingredients to satisfy restrictions while minimizing cost.

$$\begin{aligned} \text{Min} \sum_{i \in I} C_i X_i \\ \text{s.t.} \;\; L_p \leq \sum_{i \in I} N_{i,p} X_i \leq U_p \quad \forall p \in P \\ \sum_{i \in I} X_i = 1 \\ X_i \geq 0 \quad \forall i \in I \end{aligned}$$

Examples

Creating a concrete model from data params & solving instance.

```
>>> import pyomo.environ as pyo
>>> instance = blending("my_params.dat") # AMPL data file
>>> opt = pyo.SolverFactory("glpk")
>>> results = opt.solve(instance)
```

`ormm.mathprog.`**`resource_allocation`**(*linear=True*, *mult_res=False*, *max_activity=True*, ***kwargs*)

Factory method returning Pyomo Abstract/Concrete Model for the Resource Allocation Problem.

Parameters

- **linear** (bool, optional) – Determines whether decision variables will be Reals (True) or Integer (False).
- **mult_res** (bool, optional) – Determines whether there are multiple of each resource or not.
- **max_activity** (bool, optional) – Determines whether there is an upper limit on the decision variables.
- ****kwargs** – Passed into Pyomo Abstract Model's *create_instance* to return Pyomo Concrete Model instead.

Notes

The Resource Allocation Problem optimizes using scarce resources for valued activities.

$$\begin{aligned}\text{Max} \sum_{a\in A} V_a X_a & \\ \text{s.t.} \sum_{a\in A} N_{r,a} X_a \leq M_r & \quad \forall r \in R \\ 0 \leq X_a \leq M_a & \quad \forall a \in A\end{aligned}$$

Examples

Creating a concrete model from data params & solving instance.

```
>>> import pyomo.environ as pyo
>>> instance = resource_allocation("my_params.dat) # AMPL data file
>>> opt = pyo.SolverFactory("glpk")
>>> results = opt.solve(instance)
```

ormm.mathprog.**scheduling**(*prob_class*, *linear=False*, ***kwargs*)

Calls factory methods for different scheduling problems.

The *prob_class* parameter allows the user to choose from different types of problem classes, which in turn have different model structures.

The valid choices are:

- employee: A simple employee scheduling problem to minimize the number of workers employed to meet period requirements. Currently assumes that a worker works their periods in a row (determined by *Shift-Lenth* parameter).
- rental: A type of scheduling problem where there are different plans (with different durations & costs), and the goal is to minimize the total cost of the plans purchased while meeting the period requirements (covering constraints).
- **py:obj`agg_planning`** A planning problem where the decision variables

  are how much to produce during each period, to minimize production and holding costs while satisfying demand.

More details on these model classes can be found in the Notes section here, as well as the corresponding section of the *Mathematical Programming*.

Parameters

- **prob_class** (`str`, optional) – Choice of "employee", "rental", or "agg_planning" to return different scheduling models.
- **linear** (`bool`, optional) – Determines whether decision variables will be Reals (True) or Integer (False).
- ****kwargs** – Passed into Pyomo Abstract Model's *create_instance* to return Pyomo Concrete Model instead.

Raises `TypeError` – Raised if invalid argument value is given for *prob_class*.

Notes

The employee model minimizes workers hired with covering constraints:

$$\begin{aligned} \text{Min} \sum_{p \in P} X_p & \\ \text{s.t.} \sum_{p-(L-1)}^{p} X_p \geq R_p & \quad \forall p \in P \\ X_p \geq 0, \text{int} & \quad \forall p \in P \end{aligned}$$

The rental model minimizes cost of plans purchased with covering constraints:

$$\begin{aligned} \text{Min} \sum_{a \in A} C_a \sum_{p \in P \,|\, (p,a) \in J} X_{(p,a)} & \\ \text{s.t.} \sum_{j \in J \,|\, f} X_j \geq R_p & \quad \forall p \in P \\ X_j \geq 0, \text{int} & \quad \forall j \in J \end{aligned}$$

The aggregate planning model minimizes production & holding costs while meeting demand over a number of periods:

$$\begin{aligned} \text{Min} \sum_{p \in P} C_p X_p + h Y_p & \\ \text{s.t.} \;\; Y_{p-1} + X_p - Y_p = D_p & \qquad \forall p \in P \\ Y_p \leq m & \qquad \forall p \in P \\ Y_{\min(P)-1} = I_I & \\ Y_{\max(P)} = I_F & \\ X_p,\, Y_p \geq 0, \text{int} & \quad \forall p \in P \end{aligned}$$

`ormm.mathprog.`**`print_sol`**`(`*instance*, *money_obj=False*`)`

Print the solution to the solved *instance*.

Parameters

- **instance** (`pyomo.environ.ConcreteModel`) – A solved model to retrieve objective & variable values.
- **money_obj** (*bool*) – Whether or not the objective is a monetary value (adds $ if True).

Notes

Assumes the objective is retrievable by `instance.OBJ()`

`ormm.mathprog.`**`sensitivity_analysis`**`(`*instance*`)`

Return dataframe containing sensitivity analysis

Parameters **`instance`** (`pyomo.environ.ConcreteModel`) – A solved model to retrieve dual suffix.

Returns Dataframe with sensitivity analysis information.

Return type pandas.DataFrame

Notes

Assumes the dual suffix is retrievable by `instance.dual`.

3.4.2 ORMM Markov

`analyze_dtmc`(P[, states, sim_kwargs, …])	Perform Markov Analysis of discrete time discrete state markov chain (DTMC) process.
`analyze_ctmc`(states, rate_matrix[, t, d, n, …])	Perform Markov Analysis of continuous time discrete state markov chain (CTMC) process.
`print_markov`(analysis[, mtype])	Print results of markov analysis.

`ormm.markov.`**`analyze_dtmc`**`(`*P*, *states=None*, *sim_kwargs=None*, *trans_kwargs=None*, *cost_kwargs=None*`)`

Perform Markov Analysis of discrete time discrete state markov chain (DTMC) process.

Parameters

- **`P`** (*`array-like`*) – The transition matrix. Must be of shape n x n.
- **`states`** (*`array-like`*) – Array_like of length n containing the values associated with the states, which must be homogeneous in type. If None, the values default to integers 0 through n-1.
- **`sim_kwargs`** (*`dict`*) – Dictionary of key word arguments to be passed to the simulation of the markov process. If None, then no simulation will be performed. These include ts_length (length of each simulation) and init (Initial state values).
- **`trans_kwargs`** (*`dict`*) – Dictionary of options for the transient probability analysis (tsa). If None is passed instead of dict, no tsa will be done. ts_length is the number of time periods to analyze, while init is the initial state probability vector.
- **`cost_kwargs`** (*`dict`*) – Dictionary of cost parameters for cost analysis. If None, then no cost analysis will be performed. These include state (vector of costs of being in each state), transition (matrix of costs of transitioning from one state to another), and num (number of these processes - total cost multiplied by this, default 1).

Returns **analysis** – Dictionary with results of markov analysis

Return type dict

Raises **`ValueError`** – If sim_kwargs, trans_kwargs, or cost_kwargs is given, but their required arguments are not passed. These are described in the Notes section.

Notes

The required arguments if the kwargs are passed are:

- sim_kwargs: *ts_length* is required, the length of the sim.
- trans_kwargs: *ts_length* is required, the number of periods to analyze
- cost_kwargs: *state* and *transition* are required, which are the costs of being in any state and the costs of transitioning from one state to another.

`ormm.markov.`**`analyze_ctmc`**(*states*, *rate_matrix*, *t=None*, *d=None*, *n=None*, *init=None*)

Perform Markov Analysis of continuous time discrete state markov chain (CTMC) process.

Parameters

- **`states`** (*`array-like`*) – Vector-like of length M containing the values associated with the states, which must be homogeneous in type. If None, the values default to integers 0 through M-1.
- **`rate_matrix`** (*`array-like`*) – matrix of size MxM detailing the stationary probabilities of moving from one state to another.
- **`t`** (*`int`*) – Integer for the end time period for the transient probability analysis. If this is given, then either d or n must be given, and init must be given as well.
- **`d`** (*`float`*) – Float for the small delta (amount of time per step) for numerically solving the transient probabilities. Either this or n (number of steps) must be given (one can be inferred from the other and 't').
- **`n`** (*`int`*) – Integer of the number of steps to take for numerically solving the transient probabilities. Either this or 'd' must be given. If both are given and they do not coincide with t (d*n = t), an error will be raised.
- **`init`** (*`array-like`*) – Vector-like of length n containing the initial state values for the transient probability analyis. This must be given if 't', 'd', or 'n' is given.

Returns **analysis** – Dictionary with results of markov analysis

Return type dict

Raises **`ValueError`** – If some but not all of the required transient probability analysis arguments are given, or if t, d, and n are given, but their values are invalid (t = n * d).

`ormm.markov.`**`print_markov`**(*analysis*, *mtype='dtmc'*)

Print results of markov analysis.

Parameters

- **`analysis`** (*`dict`*) – dictionary returned from markov_analysis() containing cdfs, steady state probs, etc.
- **`mtype`** (*`str`*) – Type of markov analysis that is being passed. Must be 'dtmc' or 'ctmc'.

Raises **`ValueError`** – If invalid value is passed for 'mtype'.

3.4.3 ORMM Network

`transportation_model`(**kwargs)	Factory method for the balanced transportation problem.
`Graph`([arcs, costs, nodes])	Fully connected network of nodes via arcs with costs.

`ormm.network.`**`transportation_model`**`(`***kwargs*`)`

Factory method for the balanced transportation problem.

By balanced, we mean that this implementation currently requires the data to have the same number of source nodes as destination nodes. Your data can be easily changed to meet this requirement; see the notes section.

This network flow problem has a set of source nodes and destination nodes, with shipping costs between each of them. There are demands at the destinations, and supply limits at the sources. The objective is to minimize the shipping costs while meeting the demands.

Parameters **`**kwargs`** – Passed into Pyomo Abstract Model's *create_instance* to return Pyomo Concrete Model instead.

Returns Abstract Model with the sets, parameters, decision variables, objective, and constraints for the transportation problem. Returns a Concrete Model instead if any kwargs passed.

Return type pyomo.environ.AbstractModel or pyomo.environ.ConcreteModel

Notes

This is a bipartite network with m supply nodes and n destination nodes. If not possible to ship from i to j, a large cost M should be passed.

Assumes the feasibility property holds (total supply equals total demand) - then becomes balanced TP. You can modify data so this requirement is satisfied.

Let δ be the excess amount (positive). Add a dummy source to the data with index m + 1 if demand > supply.

$s_{m+1} = \delta \; ; c_{m+1,j} = 0 \quad \forall j \in J$

Add a dummy demand to the data with index n + 1 if supply > demand.

$d_{n+1} = \delta \; ; c_{i,n+1} = 0 \quad \forall i \in I$

$$\begin{aligned}
\text{Min} \sum_{i \in I} \sum_{j \in J} C_{i,j} X_{i,j} \\
\text{s.t.} \sum_{j \in J} X_{i,j} = S_i \quad \forall i \in I \\
\sum_{i \in I} X_{i,j} = D_j \quad \forall j \in J \\
X_{i,j} \geq 0, \text{int} \quad \forall i \in I, j \in J
\end{aligned}$$

class `ormm.network.`**`Graph`**`(`*arcs=None*, *costs=None*, *nodes=None*`)`

Fully connected network of nodes via arcs with costs. This class can be used to create graphs (networks), and then solve different network flow models on these graphs such as the transportation model and shortest path model.

Parameters

- **`arcs`** (`dict`, optional) – All possible paths (arcs) from each node. e.g. {'A': ['B', 'C', 'D', 'E'], . . . }

- **costs** (dict, optional) – The cost of traveling from one node to another. e.g. {('A', 'B'): 2, ('A', 'C'): 5, . . . }
- **nodes** (set, optional) – A set of all unique nodes in the graph. e.g. {"A", "B", "C", . . . }

INDEX