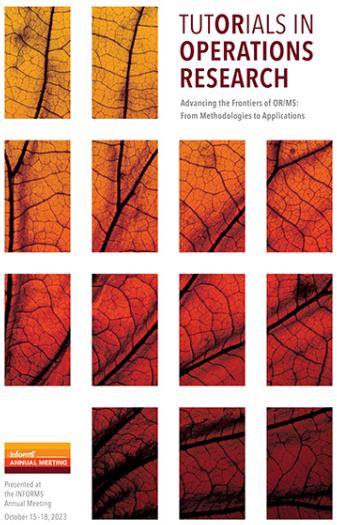


INFORMS TutORials in Operations Research

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Stockpyl: A Python Package for Inventory Optimization and Simulation

https://orcid.org/0000-0002-2227-7030Lawrence V. Snyder

To cite this entry: https://orcid.org/0000-0002-2227-7030Lawrence V. Snyder. Stockpyl: A Python Package for Inventory Optimization and Simulation. *In* INFORMS TutORials in Operations Research. Published online: 13 Oct 2023; 156-197.
https://doi.org/10.1287/educ.2023.0256

https://pubsonline.informs.org/series/educ

TUTORIALS IN OPERATIONS RESEARCH
 | ISBN 979-8-9882856-1-8
https://doi.org/10.1287/educ.2023.0256

Stockpyl: A Python Package for Inventory Optimization and Simulation

Lawrence V. Snyder[a]
[a] Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, Pennsylvania 18015
Contact: larry.snyder@lehigh.edu, https://orcid.org/0000-0002-2227-7030 (LVS)

Abstract Stockpyl is a Python package for inventory optimization and simulation that implements classical single-node inventory models like the economic order quantity, newsvendor, and Wagner-Whitin problems. It also contains algorithms for multiechelon inventory optimization under both stochastic-service model and guaranteed-service model assumptions. It has extensive features for simulating multiechelon inventory systems. In this tutorial, we provide an overview of the optimization features of Stockpyl, including a short primer on the inventory models and algorithms that underlie the modules in Stockpyl. Python code snippets are used throughout, and a Jupyter notebook containing all the code is available on the GitHub repository for the Stockpyl project.

Keywords inventory management • optimization • simulation • Python

1. Introduction

1.1. About Stockpyl

Stockpyl is a Python package for inventory optimization and simulation. It implements classical single-node inventory models like the economic order quantity (EOQ), newsvendor, and Wagner-Whitin problems. It also contains algorithms for multiechelon inventory optimization (MEIO) under both stochastic-service model (SSM) and guaranteed-service model (GSM) assumptions. Moreover, it has extensive features for simulating multiechelon inventory systems.

In this tutorial, we provide an overview of the inventory-optimization features of Stockpyl. Stockpyl also has simulation features, but they are outside the scope of this tutorial and will be discussed in a future document.

Remark 1. The notation and references (equations, sections, examples, etc.) used throughout this tutorial refer to Snyder and Shen [28]. We will use the shorthand FoSCT to refer to this text.

Full documentation for the package (much of which overlaps with this tutorial) can be found at `stockpyl.readthedocs.io/en/latest/`. This tutorial is static and will not be updated over time, so for up-to-date documentation, see the online documentation. Source code for the package can be found at `github.com/LarrySnyder/stockpyl`, where you can also submit bug reports or feature requests (or contribute to the project). The GitHub repo also contains a Jupyter notebook with all the code from this tutorial.

Before we get into the details, let's look at a few examples.

1.2. A Few Quick Examples

Solve the EOQ problem with a fixed cost of 8, a holding cost of 0.225, and a demand rate of 1,300 (example 3.1 in FoSCT):

```
>>> from stockpyl.eoq import economic_order_quantity
>>> Q, cost = economic_order_quantity(fixed_cost=8, holding_cost=0.225,
    demand_rate=1300)
>>> Q
304.0467800264368
>>> cost
68.41052550594829
```

Solve the newsvendor problem with a holding cost of 0.18, a stockout cost of 0.70, and demand that is normally distributed with a mean of 50 and a standard deviation of 8 (example 4.3 in FoSCT):

```
>>> from stockpyl.newsvendor import newsvendor_normal
>>> S, cost = newsvendor_normal(holding_cost=0.18, stockout_cost=0.70,
    demand_mean=50, demand_sd=8)
>>> S
56.60395592743389
>>> cost
1.9976051931766445
```

Remark 2. Most functions in Stockpyl use longer, more descriptive parameter names (`holding_cost`, `fixed_cost`, etc.) rather than the shorter notation assigned to them in textbooks and articles (h, K, etc.).

Stockpyl can solve the Wagner-Whitin model using dynamic programming (DP):

```
>>> from stockpyl.wagner_whitin import wagner_whitin
>>> T = 4
>>> h = 2
>>> K = 500
>>> d = [90, 120, 80, 70]
>>> Q, cost, theta, s = wagner_whitin(T, h, K, d)
>>> Q # Optimal order quantities
[0, 210, 0, 150, 0]
>>> cost # Optimal cost
1380.0
>>> theta # Cost-to-go function
array([   0., 1380.,  940.,  640.,  500.,    0.])
>>> s # Optimal next period to order in
[0, 3, 5, 5, 5]
```

It can also solve finite-horizon stochastic inventory problems using DP:

```
>>> from stockpyl.finite_horizon import finite_horizon_dp
>>> T = 5
>>> h = 1
>>> p = 20
>>> h_terminal = 1
>>> p_terminal = 20
>>> c = 2
>>> K = 50
>>> mu = 100
>>> sigma = 20
>>> s, S, cost, _, _, _ = finite_horizon_dp(T, h, p, h_terminal, p_terminal,
    c, K, mu, sigma)
>>> s # Reorder points
[0, 110, 110, 110, 110, 111]
>>> S # Order-up-to levels
[0, 133.0, 133.0, 133.0, 133.0, 126.0]
```

Stockpyl includes an implementation of the Clark and Scarf [5] algorithm for stochastic serial systems (or, more precisely, the reworking of it by Chen and Zheng [4]):

```
>>> from stockpyl.supply_chain_network import serial_system
>>> from stockpyl.ssm_serial import optimize_base_stock_levels
>>> # Build network.
>>> network = serial_system(
        num_nodes=3,
        node_order_in_system=[3, 2, 1],
        echelon_holding_cost=[4, 3, 1],
        local_holding_cost=[4, 7, 8],
        shipment_lead_time=[1, 1, 2],
        stockout_cost=40,
        demand_type='N',
        mean=10,
        standard_deviation=2
    )
>>> # Optimize echelon base-stock levels.
>>> S_star, C_star = optimize_base_stock_levels(network=network)
>>> S_star
{3: 44.1689463285519, 2: 34.93248526934437, 1: 25.69602421013684}
>>> C_star
227.15328525645054
```

Stockpyl has extensive features for simulating multiechelon inventory systems. Here, we simulate the same serial system as previously shown, obtaining an average cost per period that is similar to what the theoretical model predicted.

```
>>> from stockpyl.supply_chain_network import
                                        echelon_to_local_base_stock_levels
>>> from stockpyl.sim import simulation
>>> from stockpyl.policy import Policy
>>> # Convert to local base-stock levels and set nodes' inventory policies.
>>> S_star_local = echelon_to_local_base_stock_levels(network, S_star)
>>> for n in network.nodes:
       n.inventory_policy = Policy(type='BS',
           base_stock_level=S_star_local[n.index], node=n)
>>> # Simulate the system.
>>> T = 1000
>>> total_cost = simulation(network=network, num_periods=T, rand_seed=42)
>>> total_cost/T # average total cost per period.
226.16794575837224
```

Stockpyl also implements the dynamic programming algorithm of Graves and Willems [9] for optimizing committed service times (CSTs) in acyclical GSM systems:

```
>>> from stockpyl.gsm_tree import optimize_committed_service_times
>>> from stockpyl.instances import load_instance
>>> # Load a named instance, Example 6.5 from FoSCT.
>>> tree = load_instance("example_6_5")
>>> # Optimize committed service times.
>>> opt_cst, opt_cost = optimize_committed_service_times(tree)
>>> opt_cst
{1: 0, 3: 0, 2: 0, 4: 1}
>>> opt_cost
8.277916867529369
```

Stockpyl is organized into *modules*, each of which contains code for a particular aspect of inventory optimization or simulation.

1.3. Installing Stockpyl

Stockpyl is available on PyPI at pypi.org/project/stockpyl/. To install, use the following:

```
pip install stockpyl
```

Stockpyl requires Python v3.7 or later.

2. Single-Echelon Inventory Optimization

Stockpyl contains code to solve the following types of single-echelon inventory optimization problems:

- The economic order quantity (EOQ) problem (Section 2.1)
- The newsvendor problem (Section 2.2)
- The (r, Q) and (s, S) optimization problems (Sections 2.3 and 2.4)
- The Wagner-Whitin problem (Section 2.6)
- Finite-horizon stochastic problems with or without fixed costs (Section 2.7)
- Plus a number of single-echelon problems with supply uncertainty (Section 3)

We discuss each of these problems in the following sections.

2.1. EOQ Problem and Variants

The `eoq` module contains code for solving the EOQ problem and some of its variants. The EOQ problem is arguably the oldest and simplest inventory optimization problem, dating back to Harris [15]. The problem assumes that the demand is continuous, deterministic, and constant and that there is a fixed cost to place an order and a holding cost to store inventory. The goal of the problem is to determine the optimal order quantity to use for each order. Many variants of the EOQ have been proposed in the literature, and Stockpyl implements some of the more common ones. The notation for these problems—both the mathematical notation from FoSCT and the Python names in Stockpyl—is summarized in Table 1.

The main function in the module is the `economic_order_quantity()` function, which implements the basic EOQ model (Harris [15]; FoSCT, section 3.2):

$$Q^* = \sqrt{\frac{2K\lambda}{h}}, \tag{1}$$

$$g(Q^*) = \sqrt{2K\lambda h}. \tag{2}$$

Table 1. Notation for `economic_order_quantity()` and related functions.

Math notation (FoSCT)	Python variable name (Stockpyl)	Type	Definition
K	`fixed_cost`	Parameter	Fixed cost per order
h	`holding_cost`	Parameter	Holding cost per item per unit time
λ	`demand_rate`	Parameter	Demand quantity per unit time
Q	`order_quantity`	Decision variable	Order quantity
$g(\cdot)$	`cost`	Objective function	Cost per unit time attained by Q
x	`stockout_fraction`	Decision variable	Stockout fraction[a]
μ	`production_rate`	Parameter	Production quantity per unit time[b]

[a] `economic_order_quantity_with_backorders()`.
[b] `economic_production_quantity()`.

The function has the following signature:

```
economic_order_quantity(fixed_cost, holding_cost, demand_rate,
    order_quantity=None)
```

and returns two parameters: `order_quantity` (the optimal order quantity) and `cost` (the corresponding optimal cost). For example (FoSCT, example 3.1),

```
>>> economic_order_quantity(8, 0.225, 1300)
(304.0467800264368, 68.41052550594829)
```

Alternatively, you can pass an order quantity Q to `economic_order_quantity()` in the optional parameter `order_quantity`, and the function will return the cost of that order quantity (and the order quantity itself), calculated as

$$g(Q) = \frac{K\lambda}{Q} + \frac{hQ}{2}. \tag{3}$$

For example,

```
>>> economic_order_quantity(8, 0.225, 1300, order_quantity=350)
(350, 69.08928571428572)
```

Similar functions solve variants of the EOQ:

- `economic_order_quantity_with_backorders()` implements the economic order quantity problem with backorders (EOQB) (FoSCT, section3.5)
- `economic_production_quantity()` implements the economic production quantity (EPQ) problem (FoSCT, section 3.6)
- `joint_replenishment_problem_silver_heuristic()` implements the heuristic of Silver [24] for the joint replenishment problem (JRP)

2.2. Newsvendor Problem

The newsvendor problem (Arrow et al. [1]; FoSCT, section 4.3.2) is perhaps the simplest stochastic inventory problem. The `newsvendor` module contains code for solving several flavors of this problem. The notation for the basic functions is summarized in Table 2.

Table 2. Notation for `newsvendor_normal()` and related functions.

Math notation (FoSCT)	Python variable name (Stockpyl)	Type	Definition
h	`holding_cost`	Parameter	Holding cost per item per period
p	`stockout_cost`	Parameter	Stockout cost per item per period
μ	`demand_mean`	Parameter	Mean demand per period
σ	`demand_sd`	Parameter	Standard deviation of demand per period[a]
L	`lead_time`	Parameter	Lead time[a]
S	`base_stock_level`	Decision variable	Base-stock level
$g(\cdot)$	`cost`	Objective function	Expected cost per period attained by S
$f(\cdot)$	`demand_pdf`	Parameter	Demand pdf[b]
$f(\cdot)$	`demand_pmf`	Parameter	Demand pmf[c]

[a]Not used in `newsvendor_poisson()`.
[b]`newsvendor_continuous()`.
[c]`newsvendor_discrete()`.

Remark 3. The functions in the `newsvendor` module solve either the single-period newsvendor problem or its infinite-horizon analogue. In the infinite-horizon version, unsold items may be held from one period to the next in the form of inventory (incurring a holding cost), and unmet demands may be held from one period to the next in the form of backorders (incurring a stockout cost). We use the term "newsvendor" to refer to both the single-period and infinite-horizon problems because they are mathematically equivalent. However, some authors prefer not to use the term "newsvendor" to describe the infinite-horizon problem. That problem may alternatively be called the *base-stock optimization problem*, because the firm is following a base-stock policy and the objective is to find the optimal base-stock level.

The `newsvendor_normal()` function solves the newsvendor problem with normally distributed demands (FoSCT, section 4.3.2.5). This version of the problem assumes the demand per period is distributed as $N(\mu, \sigma^2)$. If h and p are the per-unit holding and stockout costs (sometimes called overage and underage costs and denoted c_o and c_u), then the optimal base-stock level and cost are given by

$$S^* = \mu + z_\alpha \sigma, \tag{4}$$

$$g(S^*) = (h+p)\phi(z_\alpha)\sigma, \tag{5}$$

where $\alpha = p/(h+p)$, z_α is the αth fractile of the standard normal distribution, and $\phi(\cdot)$ is the probability density function (pdf) of the standard normal distribution. The function has the following signature:

```
newsvendor_normal(holding_cost, stockout_cost, demand_mean, demand_sd,
    lead_time=0, base_stock_level=None)
```

and returns two parameters: `base_stock_level` (the optimal base-stock level) and cost (the corresponding optimal expected cost). For example (FoSCT, example 4.3),

```
>>> newsvendor_normal(0.18, 0.70, 50, 8)
(56.60395592743389, 1.9976051931766445)
```

Alternatively, you can pass a base-stock level S to `newsvendor_normal()` in the optional parameter `base_stock_level`, and the function will return the expected cost of that base-stock level (and the base-stock level itself), calculated as

$$g(S) = h\int_0^y (y-d)f(d)dd + p\int_y^\infty (d-y)f(d)dd = [hz + (h+p)\mathscr{L}(z)]\sigma, \tag{6}$$

where $z = (S-\mu)/\sigma$ and $\mathscr{L}(\cdot)$ is the standard normal loss function. For example,

```
>>> newsvendor_normal(0.18, 0.70, 50, 8, base_stock_level=60)
(60, 2.156131552870387)
```

If you provide this function with a lead time L in the `lead_time` parameter, then the optimal base-stock level is given by (FoSCT, section 4.3.4.1):

$$S^* = (L+1)\mu + \sqrt{L+1}\, z_\alpha \sigma, \tag{7}$$

which reduces to (4) when $L = 0$.

The `newsvendor_poisson()` function solves the problem for Poisson demands. It has a similar signature as `newsvendor_normal()` except that it does not have a parameter for the standard deviation.

Stockpyl provides two functions that allow you to solve newsvendor problems with generic distributions (i.e., not normal or Poisson). The `newsvendor_continuous()` function is for continuous distributions. It allows you to specify the demand pdf in one of two ways:

- As a `scipy.stats.rv_continuous` object, or
- As a function that takes a single argument and returns the pdf of that argument.[1]

The signature of the function is

```
newsvendor_continuous(holding_cost, stockout_cost, demand_distrib=None,
    demand_pdf=None, base_stock_level=None)
```

(Exactly one of `demand_distrib` and `demand_pdf` must be provided.) For example (FoSCT, problem 4.8(b)),

```
>>> from scipy.stats import lognorm
>>> demand_distrib = lognorm(0.3, 0, np.exp(6))
>>> newsvendor_continuous(1, 0.1765, demand_distrib)
(295.6266448071368, 29.44254351324322)
```

For discrete distributions, the `newsvendor_discrete()` function allows you to specify the demand probability mass function (pmf) either

- As a `scipy.stats.rv_discrete` object or
- As a Python dictionary in which the keys are the possible demand values and the values are their probabilities.

The signature is

```
newsvendor_discrete(holding_cost, stockout_cost, demand_distrib=None,
    demand_pmf=None, base_stock_level=None)
```

(Exactly one of `demand_distrib` and `demand_pmf` must be provided.) Here's an example (FoSCT, example 4.7) that specifies the pmf as an `rv_discrete` object:

```
>>> from scipy.stats import poisson
>>> demand_distrib = poisson(6)
>>> newsvendor_discrete(1, 4, demand_distrib)
(8.0, 3.5701069457709416)
```

Another example in which we solve the same problem but passing a dictionary with the same pmf is as follows:

```
>>> from scipy.stats import poisson
>>> d = range(0, 41)
>>> f = [poisson.pmf(d_val, 6) for d_val in d]
>>> demand_pmf = dict(zip(d, f))
>>> newsvendor_discrete(1, 4, demand_pmf=demand_pmf)
(8, 3.570106945770941)
```

The functions discussed thus far use the "implicit" form of the newsvendor problem, in which we provide holding and stockout (overage and underage) costs. Another version of the problem, the "explicit" version, optimizes based on the revenue r per item sold, the cost c per item purchased, and the salvage value v per item left over at the end of the period. For a notation summary, see Table 3. If we let

$$h' = c - v, \tag{8}$$

$$p' = r - c, \tag{9}$$

then the explicit formulation under r, c, and v is equivalent to the implicit formulation under h' and p'.

Table 3. Notation for `newsvendor_normal_explicit()` and `newsvendor_poisson_explicit()` functions.

Math notation (FoSCT)	Python variable name (Stockpyl)	Type	Definition
r	`revenue`	Parameter	Revenue per item sold
c	`purchase_cost`	Parameter	Cost per item purchased
v	`salvage_value`	Parameter	Revenue per item unsold
μ	`demand_mean`	Parameter	Mean demand per period
σ	`demand_sd`	Parameter	Standard deviation of demand per period[a]
L	`lead_time`	Parameter	Lead time
S	`base_stock_level`	Decision variable	Base-stock level
$g(\cdot)$	`cost`	Objective function.	Expected cost per period attained by S

[a]Not used in `newsvendor_poisson_explicit()`.

Stockpyl provides two functions to solve the explicit form of the newsvendor problem: `newsvendor_normal_explicit()`, which has the signature

```
newsvendor_normal_explicit(revenue, purchase_cost, salvage_value,
    demand_mean, demand_sd, holding_cost=0, stockout_cost=0, lead_time=0,
    base_stock_level=None)
```

and `newsvendor_poisson_explicit()`, which has the signature

```
newsvendor_poisson_explicit(revenue, purchase_cost, salvage_value,
    demand_mean, holding_cost=0, stockout_cost=0, lead_time=0,
    base_stock_level=None)
```

(The optional `holding_cost` and `stockout_cost` parameters allow you to specify additional holding and stockout costs, in addition to the costs implied by the revenue, purchase cost, and salvage value.) For example,

```
>>> newsvendor_normal_explicit(1, 0.3, 0.12, 50, 8)
(56.60395592743389, 33.002394806823354)
>>> newsvendor_poisson_explicit(1, 0.3, 0.12, 50)
(56.60395592743389, 33.002394806823354)
```

The `newsvendor` module also has functions related to the so-called "myopic" cost function (Veinott [30]); for further discussion, see the online documentation.

2.3. The (r, Q) Optimization Problem

The `rq` module contains functions for evaluating and optimizing (r, Q) policies. The functions in this module assume a continuous-review system; different functions make different assumptions about the demand distribution. The module offers functions for both exact and approximate optimization methods to find r and Q. The notation for these functions is summarized in Table 4.

Four functions implement heuristic (approximate) methods for finding r and Q. These are discussed in FoSCT, section 5.3. The `r_q_eil_approximation()` function implements the "expected-inventory-level" (EIL) approximation (Hadley and Whitin [14], Whitin [32]), which assumes the demand per unit time is distributed as $N(\lambda, \tau^2)$. We iteratively solve

$$Q = \sqrt{\frac{2\lambda[K + pn(r)]}{h}}, \tag{10}$$

Table 4. Notation for `rq` and `ss` modules.

Math notation (FoSCT)	Python variable name (Stockpyl)	Type	Definition
h	`holding_cost`	Parameter	Holding cost per item per unit time
p	`stockout_cost`	Parameter	Stockout cost per item per unit time
K	`fixed_cost`	Parameter	Fixed cost per order
Λ	`demand_mean`	Parameter	Mean demand per unit time
T	`demand_sd`	Parameter	Standard deviation of demand per unit time[a]
L	`lead_time`	Parameter	Lead time
R	`reorder_point`	Decision variable	Reorder point
Q	`order_quantity`	Decision variable	Order quantity
$g(\cdot,\cdot)$	`cost`	Objective function	Expected cost per period attained by r and Q

[a]Not used in `r_q_poisson_cost()`, `r_q_poisson_exact()`, `s_s_cost_discrete()`, and `s_s_discrete_exact()`.

$$r = F^{-1}\left(1 - \frac{Qh}{p\lambda}\right). \tag{11}$$

The expected cost is given by

$$g(r, Q) = h\left(r - \lambda L + \frac{Q}{2}\right) + \frac{K\lambda}{Q} + \frac{p\lambda n(r)}{Q}. \tag{12}$$

The function has the following signature:

```
r_q_eil_approximation(holding_cost, stockout_cost, fixed_cost, demand_mean,
    demand_sd, lead_time, tol=1e-6)
```

and returns three parameters: `reorder_point` (the approximate reorder point), `order_quantity` (the approximate order quantity), and `cost` (the corresponding expected cost). The optional parameter `tol` allows you to specify the tolerance for the iterative method.

For example (FoSCT, example 5.2):

```
>>> r_q_eil_approximation(0.225, 7.5, 8, 1300, 150, 1/12)
(213.97044213580244, 318.5901810768729, 95.45114022285196)
```

The other three functions for (r, Q) approximations work similarly:

- `r_q_eoqb_approximation()` implements the "economic order quantity with backorders" (EOQB) approximation [34], in which

$$Q = \sqrt{\frac{2K\lambda(h+p)}{hp}}, \tag{13}$$

and r is set such that

$$g(r) = g(r + Q), \tag{14}$$

where $g(\cdot)$ is the newsvendor cost function (6).

- `r_q_eoqss_approximation()` implements the "EOQ plus safety stock" (EOQSS) approximation, in which

$$Q = \sqrt{\frac{2K\lambda}{h}}, \tag{15}$$

$$r = \mu + z_\alpha \sigma, \tag{16}$$

where $\alpha = p/(h+p)$ and $\mu \equiv \lambda L$ and $\sigma \equiv \tau\sqrt{L}$ are the mean and standard deviation of the *lead-time demand*, respectively.

- `r_q_loss_function_approximation()` implements the "loss function" approximation (Hadley and Whitin [14]), in which we iteratively solve

$$Q = \sqrt{\frac{2[K\lambda + (h+p)n^{(2)}(r)]}{h}}, \tag{17}$$

$$n(r) = \frac{hQ}{h+p}, \tag{18}$$

where $n(\cdot)$ and $n^{(2)}(\cdot)$ are the loss function and second-order loss function for the lead-time demand distribution.

These functions all assume that the demand is normally distributed. They do not return the expected cost of the solution found; to calculate the expected cost, use the `r_q_cost()` function (discussed in the next paragraph). The `r_q_eoqb_approximation()` and `r_q_eoqss_approximation()` functions do not have a `tol` parameter because they are not iterative methods.

The `r_q_cost()` function calculates the *exact* expected cost of given values of r and Q, assuming the demand is normally distributed. The cost function is given by

$$g(r, Q) = \frac{K\lambda + \int_r^{r+Q} g(y)dy}{Q}, \tag{19}$$

where $g(\cdot)$ is the newsvendor cost function (Hadley and Whitin [14]). (The functions described previously attempt to optimize (19) approximately.)

For a given value of Q, the optimal r satisfies (14) (Zheng [34]). The `r_q_optimal_r_for_q()` function finds such an r iteratively using bisection search.

Finally, two functions address (r, Q) problems with Poisson demands. The function `r_q_cost_poisson()` calculates the discrete analogue of (19), namely,

$$g(r, Q) = \frac{K\lambda + \sum_{y=r+1}^{r+Q} g(y)}{Q}. \tag{20}$$

The `r_q_poisson_exact()` function determines the exact r and Q that minimize (20) using the algorithm by Federgruen and Zheng [7]. For example (FoSCT, example 5.8),

```
>>> r_q_poisson_exact(20, 150, 100, 1.5, 2)
(3, 5, 107.92358063314975)
```

2.4. The (s, S) Optimization Problem

The `ss` module contains functions for evaluating and optimizing (s, S) policies in a periodic-review system. The notation is mostly the same as in Table 4. The `s_s_cost_discrete()` function calculates the exact expected cost of an (s, S) policy with given parameters s and S under a discrete distribution using the method by [35, 36]. Its signature is

```
s_s_cost_discrete(reorder_point, order_up_to_level, holding_cost,
    stockout_cost, fixed_cost, use_poisson, demand_mean=None, demand_hi=None,
    demand_pmf=None)
```

The distribution can be Poisson (in which case, set `use_poisson=True` and provide the mean of the distribution in the argument `demand_mean`) or an arbitrary discrete distribution. In the latter case, the distribution is assumed to be on the integers $0, \ldots,$ `demand_hi`; set `use_poisson=False` and provide the pmf of the distribution as a list of probabilities in `demand_pmf`.

For example (FoSCT, example 4.7),

```
>>> s_s_cost_discrete(4, 10, 1, 4, 5, use_poisson=True, demand_mean=6)
8.034111561471642
```

In the next example, we assume that the demand has a geometric distribution with the same mean, 6, as in the previous example, truncating the support at 20.

```
>>> from scipy.stats import geom
>>> demand_prob = 1/6
>>> demand_hi = 20
>>> demand_pmf = [geom.pmf(x, demand_prob) for x in range(demand_hi + 1)]
>>> s_s_cost_discrete(4, 10, 1, 4, 5, use_poisson=False, demand_hi=demand_hi,
    demand_pmf=demand_pmf)
9.891030991618022
```

The `s_s_discrete_exact` function finds the exact optimal s and S for a policy under discrete (Poisson or arbitrary) demands using the algorithm by Zheng and Federgruen [35]; that is, it optimizes the cost calculated by `s_s_cost_discrete()`. Its signature is similar, except that `reorder_point` and `order_up_to_level` are outputs rather than inputs. In the following example, we optimize the instance from FoSCT (example 4.7):

```
>>> s_s_discrete_exact(1, 4, 5, True, 6)
(4.0, 10.0, 8.034111561471642)
```

Finally, the `s_s_power_approximation()` function finds heuristic values for s and S using the "power approximation" by Ehrhardt and Mosier [6], which assumes the demands are normally distributed.

2.5. Specification of Time-Based Quantities

Before moving on, we take a quick detour to discuss how time-based input and output parameters (i.e., parameters with a subscript t) are represented in Stockpyl. These are relevant in Section 2.6 and subsequent sections.

Let T be the number of time periods. In most cases, if a parameter may differ from one period to the next, Stockpyl allows you to specify it in one of three ways:

- As a singleton, in which case the parameter is assumed to equal that value in every time period.
- As a list with T elements, in which case the list is assumed to contain values for periods $1, \ldots, T$ in elements $0, \ldots, T-1$.
- As a list with $T+1$ elements, in which case the list is assumed to contain values for periods $1, \ldots, T$ in elements $1, \ldots, T$, and the zeroth element is ignored.

For example, if a given problem instance has three time periods and a parameter `holding_cost` equals one in every time period, the following are equivalent:

- `holding_cost=1`
- `holding_cost=[1, 1, 1]`
- `holding_cost=[0, 1, 1, 1]`

Similarly, if a parameter `demand` equals 10, 30, and 25 in periods 1–3, respectively, then the following are equivalent:

- `demand=[10, 30, 25]`
- `demand=[0, 10, 30, 25]`

In most or all functions, you may mix input types, specifying some as singletons and some as lists. Time-based output parameters returned by Stockpyl functions are always given as lists with $T+1$ elements.

2.6. Wagner-Whitin Problem

The Wagner-Whitin problem[2] is a periodic-review, finite-horizon, deterministic inventory model that makes decisions about which periods to order in and how much to order in each of those periods. Like the EOQ model, it assumes deterministic demand, but unlike the EOQ, it is a periodic-review model, and it allows the demand to vary from one period to another.

The `wagner_whitin` module solves the problem using the classic DP algorithm by Wagner and Whitin [31]. The module contains a single function, `wagner_whitin()`, which implements the following DP recursion (FoSCT, section 3.7.3):

$$\theta_t = \min_{t<s\leq T+1} \left\{ K + h\sum_{i=t}^{s-1}(i-t)d_i + \theta_s \right\}, \tag{21}$$

where θ_t is the optimal cost in periods $t, t+1, \ldots, T$ if we place an order in period t and act optimally thereafter; $\theta_{T+1} \equiv 0$; and the rest of the notation is summarized in Table 5.
The function has the signature

```
wagner_whitin(num_periods, holding_cost, fixed_cost, demand, purchase_cost=0)
```

and returns four parameters: `order_quantities` (Q), `cost` (θ_1), `costs_to_go` (θ), and `next_order_periods` (s). The optional input parameter `purchase_cost` allows you to specify a purchase cost in each time period. With the exception of `num_periods`, the input parameters may vary by time period (see Section 2.5). The output arrays are one-indexed; that is, they have length $T+1$, where element t gives the value for period t, and element 0 is ignored.

For example (FoSCT, example 3.9),

```
>>> Q, cost, theta, s = wagner_whitin(4, 2, 500, [90, 120, 80, 70])
>>> Q
[0, 210, 0, 150, 0]
>>> cost
1380.0
>>> theta
array([   0., 1380.,  940.,  640.,  500.,    0.])
>>> s
[0, 3, 5, 5, 5]
```

In other words, the optimal solution says that we should order in periods 1 and 3, with order quantities 210 and 150 and a total cost of 1,380.

Table 5. Notation for `wagner_whitin` module.

Math notation (FoSCT)	Python variable name (Stockpyl)	Type	Definition
h	`holding_cost`	Parameter	Holding cost per item per period
K	`fixed_cost`	Parameter	Fixed cost per order
c	`purchase_cost`	Parameter	Purchase cost per item
d	`demand`	Parameter	Demand in each period
T	`num_periods`	Parameter	Number of time periods
Q	`order_quantities`	Decision variable	Order quantity in each period
θ_1	`cost`	Objective function	Optimal cost for entire horizon
θ	`costs_to_go`	Decision variable	"Cost to go" in each period
s	`next_order_periods`	Decision variable	Next period to order in if we order in t

2.7. Stochastic Finite-Horizon Problems

The `finite_horizon` module contains code for solving finite-horizon, stochastic inventory optimization problems, with or without fixed costs, under normally distributed demands, using DP. The main recursion is (FoSCT, sections 4.3.3 and 4.4.3)

$$\theta_t(x) = \min_{y \geq x} \{K\delta(y-x) + c(y-x) + g(y) + \gamma \mathbb{E}_D[\theta_{t+1}(y-D)]\}, \tag{22}$$

where $\delta(z) = 1$ if $z > 0$ and zero otherwise, and $g(y)$ is the newsvendor expected cost function (6). Alternatively, we can write

$$\theta_t(x) = \min_{y \geq x} \{H_t(y) + K\delta(y-x) - cx\}, \tag{23}$$

where

$$H_t(y) = cy + g(y) + \gamma \mathbb{E}_D[\theta_{t+1}(y-D)]. \tag{24}$$

The recursion assumes that $\theta_{T+1}(x)$ is given by a terminal cost function of the form

$$\theta_{T+1}(x) = h_{T+1}x^+ + p_{T+1}x^-, \tag{25}$$

where h_{T+1} and p_{T+1} are the terminal holding and stockout costs, respectively, and

$$x^+ \equiv \max\{x, 0\}$$
$$x^- \equiv \max\{-x, 0\}.$$

The rest of the notation is summarized in Table 6.

Table 6. Notation for `finite_horizon` module.

Math notation (FoSCT)	Python variable name (Stockpyl)	Type	Definition
h	`holding_cost`	Parameter	Holding cost per item per period
p	`stockout_cost`	Parameter	Stockout cost per item per period
h_{T+1}	`terminal_holding_cost`	Parameter	Holding cost per item at end of horizon
p_{T+1}	`terminal_stockout_cost`	Parameter	Stockout cost per item at end of horizon
c	`purchase_cost`	Parameter	Purchase cost per item
K	`fixed_cost`	Parameter	Fixed cost per order
μ	`demand_mean`	Parameter	Mean demand per period
σ	`demand_sd`	Parameter	Standard deviation of demand per period
T	`num_periods`	Parameter	Number of time periods
γ	`discount_factor`	Parameter	Discount factor
x_1	`initial_inventory_level`	Parameter	Initial inventory level
s	`reorder_points`	Decision variable	Reorder point in each period
S	`order_up_to_levels`	Decision variable	Order-up-to level in each period
$\theta_1(x_1)$	`total_cost`	Objective function	Optimal total expected cost for entire horizon
$\theta_t(x)$	`cost_matrix`	Decision variable	Matrix of DP costs: `cost_matrix[t, x]` = $\theta_t(x)$
$y_t(x)$	`oul_matrix`	Decision variable	Matrix of order-up-to levels: `oul_matrix[t, x]` = $y_t(x)$

The main function in the module is `finite_horizon_dp()`, which has the signature

```
finite_horizon_dp(num_periods, holding_cost, stockout_cost,
    terminal_holding_cost, terminal_stockout_cost, purchase_cost, fixed_cost,
    demand_mean, demand_sd, discount_factor=1.0, initial_inventory_level=0.0,
    trunc_tol=0.02, d_spread=4, s_spread=5)
```

and returns six parameters: `reorder_points` (s), `order_up_to_levels` (S), `total_cost` ($\theta_1(x_1)$), `cost_matrix` ($\theta_t(x)$), `oul_matrix` ($y_t(x)$), and `x_range`. Most of the input parameters may vary by time period (see Section 2.5). The output arrays are one-indexed; that is, they have length $T+1$, where element t gives the value for period t, and element 0 is ignored.

If $K=0$ in each time period, then a base-stock policy is optimal and the output parameters will reflect that; that is, they will have the form

$$y_t(x) = \begin{cases} S_t, & \text{if } x \le S_t \\ 0, & \text{if } x > S_t, \end{cases}$$

and `reorder_points[t]` will equal `order_up_to_levels[t]` for all periods t. If $K \neq 0$, then an (s, S) policy is optimal and the output parameters will reflect that; that is, they will have the following form:

$$y_t(x) = \begin{cases} S_t, & \text{if } x \le s_t \\ 0, & \text{if } x > s_t. \end{cases}$$

The following example solves an instance in which $T=5$; in each time period $h=1$, $p=20$, $c=2$, $K=50$, $\mu=100$, and $\sigma=20$; and the terminal costs are $h_6 = h = 1$ and $p_6 = p = 20$.

```
>>> s, S, cost, _, _, _ = finite_horizon_dp(5, 1, 20, 1, 20, 2, 50, 100, 20)
>>> s
[0, 110, 110, 110, 110, 111]
>>> S
[0, 133.0, 133.0, 133.0, 133.0, 126.0]
>>> cost
1558.6946467384012
```

In other words, $s_t = 110, 110, 110, 110, 111$ and $S_t = 133, 133, 133, 133, 126$ for $t = 1, \ldots, 5$, respectively, and the optimal total expected cost over the horizon is 1,558.70.

The input parameters `trunc_tol`, `d_spread`, and `s_spread`, and the output parameter `x_range`, relate to the truncation and discretization performed within the function. In particular, truncation and discretization are performed as follows:

(1) All demands and inventory positions are discretized to integers.

(2) The range of demand values is truncated at $\mu \pm$ `d_spread` $\times \sigma$, truncating also at zero, and accounting conservatively for the variations in demand parameters across periods.

(3) If the total probability of demand outside the demand range in any time period is greater than `trunc_tol`, a warning is issued.

(4) Lower and upper bounds on the likely x values are calculated as

$$\underline{x} = s - \texttt{s\_spread} \times \sigma - (\mu + \texttt{d\_spread} \times \sigma) \tag{26}$$

and

$$\overline{x} = S + \mu + \texttt{s\_spread} \times \sigma, \tag{27}$$

respectively, where s is the newsvendor solution and $S = s + Q_{EOQB}$, where Q_{EOQB} is the order quantity from the EOQB problem (see Section 2.1). The intuition in (26) and (27) is that the newsvendor solution provides a reasonable first estimate for the reorder point; the EOQB provides a reasonable first estimate for the difference between S and s; the `s_spread` $\times \sigma$ terms

provide some buffer around these values; and the $\mu + \texttt{d\_spread} \times \sigma$ term accounts for the demand, which reduces the inventory level further. Other details of the calculation, which account conservatively for the variations in demand parameters across periods and adjust the final period to account for the terminal costs, are omitted here. The range of x values is set to $[\underline{x}, \overline{x}]$.

(5) When calculating $H_t(y)$, the demand range is further truncated to avoid $y - d$ falling below the smallest value in the x range.

(6) If, at any point in the optimization, the optimal order-up-to level for any x and t is the largest value in the x range, suggesting that the upper end of the range is too low and that the optimal order-up-to level may be greater than it, the optimization is terminated, the upper end of the x-range is doubled, and the optimization is restarted.

(7) If, at any point in the optimization, the total probability of demand values that could bring the inventory level below the smallest value in the x range is greater than `trunc_tol`, suggesting that the lower end of the range is too high and that reasonable demand values could bump up against it, a warning is generated. The optimization is not terminated, but the user may wish to try again with a larger value of `s_spread` and/or `d_spread`.

The `finite_horizon` module contains one other function, `myopic_bounds()`, which calculates the "myopic bounds" for the finite-horizon problem (Zipkin [37]).

3. Supply Uncertainty

The literature on inventory optimization with supply uncertainty considers two main types of supply uncertainty: disruptions (in which the supplier is unavailable for a random period of time) and yield uncertainty (in which the quantity delivered differs randomly from the quantity ordered).[3] In particular, the `supply_uncertainty` module in Stockpyl implements the following:

- The EOQ model with disruptions (Section 3.1.1)
- The EOQ model with yield uncertainty (Section 3.1.2)
- The newsvendor model with disruptions (Section 3.2.1)
- The newsvendor with yield uncertainty (Section 3.2.2)

The models implemented in Stockpyl (and most inventory models with disruptions) treat disruptions as binary events—either the supplier is completely disrupted and cannot supply any items, or it is completely operational and has infinite supply available. The supplier's status is governed by a two-state Markov chain, with the `UP` and `DOWN` states indicating that the supplier is operational or disrupted, respectively. For continuous-review inventory models (EOQ), we use a continuous-time Markov chain (CTMC) with given transition rates, and for periodic-review models (newsvendor), we use a discrete-time Markov chain (DTMC) with given transition probabilties. These are discussed in more detail in Sections 3.1.1 and 3.2.1. For reviews of inventory models with disruptions, see Atan and Snyder [2] and Snyder and Shen [27].

In contrast, yield uncertainty assumes that the supplier is always operational, but the quantity delivered differs from the quantity ordered by a random amount. Stockpyl handles two common types of yield uncertainty: additive and multiplicative. In additive yield uncertainty, if we order Q units, we receive $Q + Y$, where Y is a random variable. In multiplicative yield uncertainty, we receive QZ units, where Z is a random variable. In both cases, the probability distribution of the yield (Y or Z) is assumed to be known, analogous to the case of demand uncertainty. Sometimes, restrictions are placed on the yield random variables; for example, we might assume $Y \le 0$ or $Z \le 1$ to ensure that the quantity received is not greater than the quantity ordered. (This is realistic in some settings but not in others; for example, in agriculture or chemical manufacturing, the yield is sometimes larger than intended.) However, Stockpyl does not impose such restrictions; if restrictions are required, they should be incorporated into the distribution information passed to the functions. For reviews of inventory models with yield uncertainty, see Grosfeld-Nir and Gerchak [12] and Yano and Lee [33].

3.1. EOQ Problem with Supply Uncertainty

3.1.1. EOQ Problem with Disruptions (EOQD). The EOQ problem with disruptions (EOQD; FoSCT, section 9.2.1) assumes that the supplier can be disrupted randomly, according to a two-state CTMC. The CTMC transitions from the `UP` state to the `DOWN` state with rate λ, known as the *disruption rate*, and from the `DOWN` state to the `UP` state with rate μ, the *recovery rate*. Therefore, the length of a given `UP` (nondisrupted) interval is exponentially distributed with a mean of $1/\lambda$, and the length of a given `DOWN` (disrupted) interval is exponentially distributed with a mean of $1/\mu$. Demand is constant and deterministic, as in the classical EOQ. Inventory may be depleted during a DOWN interval, in which case unmet demands are lost.

The `eoq_with_disruptions()` function solves the EOQD either exactly (using models by Berk and Arreola-Risa [3] and Parlar and Berkin [21] and optimizing numerically) or heuristically (using an approximation by Snyder [26]).

The exact expected cost function is given by

$$g(Q) = \frac{K + hQ^2/2d + pd\psi/\mu}{Q/d + \psi/\mu}, \tag{28}$$

where

$$\psi = \frac{\lambda}{\lambda + \mu}\left(1 - e^{-\frac{(\lambda+\mu)Q}{d}}\right) \tag{29}$$

is the steady-state probability that the supplier is in a down interval when the inventory level hits zero (Berk and Arreola-Risa [3]). (Note that ψ is a function of Q.) The rest of the notation is summarized in Table 7; $g(Q)$ is quasiconvex in Q but is not known to be convex. Moreover, its first-order condition cannot be solved analytically. Therefore, Q^* must be found numerically via line search, and $g(Q^*)$ must be evaluated by plugging Q^* into (28).

Alternatively, the approximation by Snyder [26] sets $\psi = \lambda/(\lambda + \mu)$, a constant, which allows the following closed-form approximate solution for Q^* and $g(Q^*)$ (Snyder [26]):

$$Q^* = \frac{-\psi dh + \sqrt{(\psi dh)^2 + 2hd\mu(K\mu + dp\psi)}}{h\mu}, \tag{30}$$

$$g(Q^*) = hQ^*. \tag{31}$$

Table 7. Notation for `eoq_with_disruptions()` function.

Math notation (FoSCT)	Python variable name (Stockpyl)	Type	Definition
h	`holding_cost`	Parameter	Holding cost per item per unit time
p	`stockout_cost`	Parameter	Stockout cost per item (not per unit time)
K	`fixed_cost`	Parameter	Fixed cost per order
d	`demand_rate`	Parameter	Demand per unit time
λ	`disruption_rate`	Parameter	Transition rate from `UP` to `DOWN` state
μ	`recovery_rate`	Parameter	Transition rate from `DOWN` to `UP` state
Q	`order_quantity`	Decision variable	Order quantity
$g(\cdot)$	`cost`	Objective function	Optimal total expected cost per unit time

The `eoq_with_disruptions()` function has the signature

```
eoq_with_disruptions(fixed_cost, holding_cost, stockout_cost, demand_rate,
    disruption_rate, recovery_rate, approximate=False)
```

and returns two parameters: `order_quantity` and `cost`. The function numerically optimizes the exact cost function (28) if `approximate =False` and uses the approximation (30)–(31) if `approximate =True`. For example (FoSCT, examples 9.1 and 9.2),

```
>>> eoq_with_disruptions(8, 0.225, 5, 1300, 1.5, 14)
(772.8110739983106, 173.95000257319708)
>>> eoq_with_disruptions(8, 0.225, 5, 1300, 1.5, 14, approximate=True)
(773.1432417118889, 173.957229385175)
```

The module also contains the function `eoq_with_disruptions_cost()`, which takes the order quantity as an input parameter and returns the (exact or approximate) cost.

3.1.2. EOQ Problem with Yield Uncertainty. In the EOQ problem with yield uncertainty (FoSCT, section 9.3.1), an order of size Q is placed, and the quantity received is $Q+Y$ (in the case of additive yield) or QZ (in the case of multiplicative yield), where Y and Z are random variables. The lead time is zero, and stockouts are not allowed. See Table 8 for additional notation.

The expected cost function and optimal solution for the EOQ with additive yield uncertainty are given by

$$g(Q) = \frac{2Kd + h\text{Var}[Y]}{2(Q + \mathbb{E}[Y])} + \frac{h(Q + \mathbb{E}[Y])}{2}, \tag{32}$$

$$Q^* = \sqrt{\frac{2Kd}{h} + \text{Var}[Y]} - \mathbb{E}[Y]. \tag{33}$$

For the EOQ with multiplicative yield uncertainty,

$$g(Q) = \frac{Kd}{Q\mathbb{E}[Z]} + \frac{hQ(\text{Var}[Z] + \mathbb{E}[Z]^2)}{2\mathbb{E}[Z]}, \tag{34}$$

Table 8. Notation for EOQ with yield uncertainty.

Math notation (FoSCT)	Python variable name (Stockpyl)	Type	Definition
h	`holding_cost`	Parameter	Holding cost per item per unit time
K	`fixed_cost`	Parameter	Fixed cost per order
d	`demand_rate`	Parameter	Demand per unit time
$\mathbb{E}[Y]$	`yield_mean`	Parameter	Mean of additive yield distribution[a]
$SD[Y]$	`yield_sd`	Parameter	Standard deviation of additive yield distribution[a]
$\mathbb{E}[Z]$	`yield_mean`	Parameter	Mean of multiplicative yield distribution[b]
$SD[Z]$	`yield_sd`	Parameter	Standard deviation of multiplicative yield distribution[b]
Q	`order_quantity`	Decision variable	Order quantity
$g(\cdot)$	`cost`	Objective function	Optimal total expected cost per unit time

[a] `eoq_with_additive_yield_uncertainty()`.
[b] `eoq_with_multiplicative_yield_uncertainty()`.

$$Q^* = \sqrt{\frac{2Kd}{h(\mathrm{Var}[Z] + \mathbb{E}[Z]^2)}}. \tag{35}$$

Remark 4. The expected cost functions and the optimal solutions for the EOQ problems with additive and multiplicative yield uncertainty depend only on the mean and standard deviation of the yield random variable and not the entire distribution. Therefore, the corresponding functions only require these quantities, not an entire distribution function, and they make no assumptions about the shape of the distribution.

The Stockpyl functions for EOQ with yield uncertainty have the following signatures:

```
eoq_with_additive_yield_uncertainty(fixed_cost, holding_cost, demand_rate,
  yield_mean, yield_sd, order_quantity=None)
eoq_with_multiplicative_yield_uncertainty(fixed_cost, holding_cost,
  demand_rate, yield_mean, yield_sd, order_quantity=None)
```

Each function returns two parameters: `order_quantity` and `cost`. The optimal input parameter `order_quantity` allows you to provide the order quantity, in which case the function will simply calculate the cost for that order quantity.

Here is an example using additive yield uncertainty (FoSCT, example 9.4):

```
>>> eoq_with_additive_yield_uncertainty(18500, 0.06, 75000, -15000, 9000)
(230246.37046881882, 12914.78222812913)
>>> eoq_with_additive_yield_uncertainty(18500, 0.06, 75000, -15000, 9000,
  order_quantity=300000)
(300000, 13426.947368421053)
```

One with multiplicative yield uncertainty (FoSCT, example 9.5):

```
>>> import math
>>> eoq_with_multiplicative_yield_uncertainty(18500, 0.06, 75000, 0.8333,
  math.sqrt(0.0198))
(254477.46130342316, 13086.16169098594)
>>> eoq_with_multiplicative_yield_uncertainty(18500, 0.06, 75000, 0.8333,
  math.sqrt(0.0198), order_quantity=300000)
(300000, 13263.770562822512)
```

3.2. Newsvendor Problem with Supply Uncertainty

We now turn our attention to newsvendor problems with supply uncertainty. These problems assume that the demand is deterministic, but the supply is stochastic (the opposite of the classical newsvendor problem). We consider an infinite-horizon model in which inventory may be carried from one time period to the next (at a holding cost of h per item per period), and unmet demands may be carried from one period to the next in the form of backorders (at a stockout cost of p per item per period).

Remark 5. The newsvendor problem with disruptions is an infinite-horizon problem; it does not make sense to consider this problem in a single-period setting since the only reason to order extra inventory in a given period is to protect against disruptions in future periods. The newsvendor problem with yield uncertainty may be interpreted as either a single-period or infinite-horizon problem. As discussed in Remark 3, we use the term "newsvendor" to describe both the single-period and infinite-horizon problems.

3.2.1. Newsvendor Problem with Disruptions.

The newsvendor problem with disruptions (Güllü et al. [13], Schmitt and Snyder [22], Tomlin [29]; FoSCT, section 9.2.2) assumes that the supplier's disruptions follow a two-state DTMC. If the supplier is in the `UP` state, it

transitions to the DOWN state with probability α (called the *disruption probability*) and remains UP with probability $1-\alpha$. If it is in the DOWN state, it transitions to UP with probability β (the *recovery probability*) and remains DOWN with probability $1-\beta$. Therefore, the length of a given UP (nondisrupted) interval has a geometric distribution with mean $1/\alpha$, and the length of a given DOWN (disrupted) interval is geometrically distributed with mean $1/\beta$. Inventory may be depleted during a DOWN interval, in which case unmet demands are lost. A base-stock policy is optimal (with the modification that the firm orders nothing during DOWN periods), and our goal is to determine the optimal base-stock level. The notation is summarized in Table 9.

For a given base-stock level S, the expected cost per period is given by

$$g(S) = \sum_{n=0}^{\infty} \pi_n [h[S-(n+1)d]^+ + p[(n+1)d - S]^+], \tag{36}$$

where π_n is the steady-state probability of being in a period in which the supplier is in the nth period of a disruption. (If $n=0$, this is the steady-state probability of being in an UP period.) In particular,

$$\pi_0 = \frac{\beta}{\alpha+\beta}, \tag{37}$$

$$\pi_n = \frac{\alpha\beta}{\alpha+\beta}(1-\beta)^{n-1}, \quad n \geq 1, \tag{38}$$

π_n is the pmf of a random variable N representing the length of the current disruption (or zero if no disruption). This random variable also has a cumulative distribution function (cdf),

$$F(n) = \sum_{i=0}^{n} \pi_i, \tag{39}$$

which equals the steady-state probability of being in a disruption that has lasted n periods or fewer. The optimal base-stock level, S^*, is given by

$$S^* = d + dF^{-1}\left(\frac{p}{p+h}\right), \tag{40}$$

where $F^{-1}(\gamma)$ is interpreted as the smallest n such that $F(n) \geq \gamma$.
These calculations are performed by the `newsvendor_with_disruptions()` function, which has the signature

Table 9. Notation for `newsvendor_with_disruptions()` function.

Math notation (FoSCT)	Python variable name (Stockpyl)	Type	Definition
h	`holding_cost`	Parameter	Holding cost per item per period
p	`stockout_cost`	Parameter	Stockout cost per item per period
d	`demand`	Parameter	Demand per period
α	`disruption_prob`	Parameter	Probability of disruption in period $t+1$ given that there is no disruption in period t
β	`recovery_prob`	Parameter	Probability of no disruption in period $t+1$ given that there is a disruption in period t
S	`base_stock_level`	Decision variable	Base-stock level
$g(\cdot)$	`cost`	Objective function	Optimal total expected cost per period

```
newsvendor_with_disruptions(holding_cost, stockout_cost, demand,
    disruption_prob, recovery_prob, base_stock_level=None)
```

and returns two parameters: `base_stock_level` and `cost`. Alternatively, you can use the optional input parameter `base_stock_level` to provide a value for S, for which the function will calculate $g(S)$.

For example (FoSCT, example 9.3),

```
>>> newsvendor_with_disruptions(0.25, 3, 2000, 0.04, 0.25)
(8000, 2737.0689302470355)
>>> newsvendor_with_disruptions(0.25, 3, 2000, 0.04, 0.25,
    base_stock_level=1200)
(1200, 5710.344790717086)
```

3.2.2. Newsvendor Problem with Yield Uncertainty. The last model with supply uncertainty we discuss is the newsvendor problem with yield uncertainty (FoSCT, section 9.3.2). As of this writing, Stockpyl only supports additive yield uncertainty for the newsvendor problem. The notation is summarized in Table 10.

The expected cost per period is given by

$$g(S) = h\int_{d-S}^{\infty}((S+y)-d)f_Y(y)dy + p\int_{-\infty}^{d-S}(d-(S+y))f_Y(y)dy, \tag{41}$$

$$= hn_Y(d-S) + p\overline{n}_Y(d-S), \tag{42}$$

where $n_Y(\cdot)$ and $\overline{n}_Y(\cdot)$ are the loss function and complementary loss function, respectively, of the yield distribution. The optimal base-stock level is given by

$$S^* = d - F_Y^{-1}\left(\frac{h}{h+p}\right). \tag{43}$$

The `newsvendor_with_additive_yield_uncertainty()` function has the signature

```
newsvendor_with_additive_yield_uncertainty(holding_cost, stockout_cost,
  demand, yield_mean=None, yield_sd=None, yield_distribution=None,
  loss_function=None, base_stock_level=None)
```

and returns two parameters: `base_stock_level` and `cost`. Alternatively, you can use the optional input parameter `base_stock_level` to provide a value for S, for which the function will calculate $g(S)$.

Table 10. Notation for `newsvendor_with_additive_yield_uncertainty()` function.

Math notation (FoSCT)	Python variable name (Stockpyl)	Type	Definition
h	`holding_cost`	Parameter	Holding cost per item per period
p	`stockout_cost`	Parameter	Stockout cost per item per period
d	`demand`	Parameter	Demand per period
$\mathbb{E}[Y]$	`yield_mean`	Parameter	Mean of yield distribution
$SD[Y]$	`yield_sd`	Parameter	Standard deviation of yield distribution
$f_Y(\cdot)$	`yield_pdf`	Parameter	Yield pdf or pmf
$n_Y(\cdot)$	`loss_function`	Parameter	Loss function for yield distribution
S	`base_stock_level`	Decision variable	Base-stock level
$g(\cdot)$	`cost`	Objective function	Optimal total expected cost per period

This function provides two ways to specify the yield distribution:

- Using `yield_mean` and `yield_sd`, in which case the yield is assumed to be normally distributed with these parameters
- Using `yield_distribution` as a `scipy.stats.rv_continuous` or `scipy.stats.rv_discrete` object

The cost function (42) requires the loss functions $n_Y(\cdot)$ and $\overline{n}_Y(\cdot)$. If `yield_distribution` is provided, the function will calculate the loss functions using the generic functions

$$n(x) = \mathbb{E}[(X-x)^+] = \int_x^\infty (y-x) f(y)\, dy, \tag{44}$$

$$\overline{n}(x) = \mathbb{E}[(X-x)^-] = \int_{-\infty}^x (x-y) f(y)\, dy, \tag{45}$$

for continuous distributions and

$$n(x) = \mathbb{E}[(X-x)^+] = \sum_{y=x}^\infty (y-x) f(y), \tag{46}$$

$$\overline{n}(x) = \mathbb{E}[(X-x)^-] = \sum_{y=-\infty}^x (x-y) f(y), \tag{47}$$

for discrete ones (FoSCT, section C.3). However, you can optionally also provide a `loss_function` to be used instead. If `loss_function` is provided, it must be a function that takes a single argument and returns a tuple consisting of the loss function and the complementary loss function value of that argument. The `loss_function` parameter is ignored if `yield_distribution` is not provided.

Example 9.6 in FoSCT has a uniform yield distribution. We first solve it without providing the loss functions explicitly (so they are calculated using a generic function).

```
>>> from scipy.stats import uniform
>>> newsvendor_with_additive_yield_uncertainty(15, 75, 1.5e6,
    yield_distribution=uniform(-500000, 1000000))
(1833333.3333333335, 6249999.997499999)
```

Next, we explicitly pass the `uniform_loss()` function to calculate the loss functions, which is more accurate:

```
>>> from stockpyl.loss_functions import uniform_loss
>>> loss_function = lambda x: uniform_loss(x, -500000, 500000)
>>> newsvendor_with_additive_yield_uncertainty(15, 75, 1.5e6,
    yield_distribution=uniform(-500000, 1000000),
    loss_function=loss_function)
(1833333.3333333335, 6250000.000000001)
```

4. Multiechelon Inventory Optimization

Multiechelon inventory optimization (MEIO) is significantly more difficult than single-echelon problems, primarily because a given node or stage (we use the terms interchangeably) experiences stochastic lead times from its upstream nodes, even if the transportation lead time is deterministic, due to random stockouts at the upstream nodes. Moreover, the probability distribution of the lead time depends on the base-stock levels (or other inventory policy parameters) at the upstream nodes, but the optimal upstream base-stock levels depend on the optimal downstream base-stock levels, which in turn depend on the probability distribution of the lead times. This vicious cycle makes it difficult even to calculate the expected cost of a given set of base-stock levels, let alone optimize them.

Two primary types of models have been developed to deal with this issue. *Stochastic-service models* (SSM) assume that each stage meets the demand from stock whenever possible and experiences stockouts otherwise, much like the newsvendor and other single-stage problems. The upstream lead times will be stochastic due to stockouts, and the expected cost function must account for these lead times either exactly or approximately, leading to considerable challenges in both formulating the expected cost and optimizing it. This is the approach taken in the seminal paper by Clark and Scarf [5] and many subsequent works.

In contrast, *Guaranteed-service models* (GSM) assume that each stage provides a deterministic upper bound on its outbound lead time. In particular, each stage sets a CST and guarantees that it will satisfy every demand within that CST. This allows a more tractable formulation of the MEIO problem because the upstream lead times are now deterministic. Conversely, the tractability comes at an expense, in the form of a relatively strong assumption that the customer demand is bounded. There is a one-to-one relationship between a set of CSTs, one for each node, and a set of corresponding base-stock levels. Therefore, if we think of SSM as a more accurate model of real-world inventory systems (a claim that is open to debate, see Graves and Willems [11]), the GSM approach can be viewed as a heuristic for setting base-stock levels in an SSM system. The GSM approach was pioneered by Kimball [18] and Simpson [25] and brought into more widespread use by Graves and Willems [9].

Stockpyl contains code to solve the following types of MEIO problems:

- Serial systems under the SSM (`ssm_serial` module; see Section 4.3)
- Serial or tree systems under the GSM (`gsm_serial` and `gsm_tree` modules; see Section 4.4)
- SSM systems with arbitrary topology, optimized using enumeration or coordinate descent (`meio_general` module; see Section 4.5)

Before discussing these modules, we discuss an important data type used throughout them.

4.1. `SupplyChainNetwork` and `SupplyChainNode` Classes

The MEIO code in Stockpyl makes use of the `SupplyChainNetwork` class, which contains all of the data for an MEIO instance. For some functions, you provide data in the form of lists, dictionaries, or singletons and the function builds the `SupplyChainNetwork` object for you, whereas other functions require you to pass a `SupplyChainNetwork` object directly.

A `SupplyChainNetwork`, in turn, consists of one or more `SupplyChainNode` objects. When you create a `SupplyChainNode`, you provide an index and any parameters you wish. For example,

```
>>> from stockpyl.supply_chain_node import SupplyChainNode
>>> # Build a node.
>>> my_node = SupplyChainNode(
      index=1,
      local_holding_cost=3,
      stockout_cost=20,
      shipment_lead_time=2
    )
>>> # Build another node.
>>> my_other_node = SupplyChainNode(
      index=2,
      name="other_node",
      echelon_holding_cost=1.5
    )
```

(A list of all available parameters is in the online documentation for the `SupplyChainNode` class.)

You can then add these nodes to a `SupplyChainNetwork`:

```
>>> from stockpyl.supply_chain_network import SupplyChainNetwork
>>> # Create an empty supply chain network.
>>> network = SupplyChainNetwork()
>>> # Add the first node to it.
>>> network.add_node(my_node)
>>> # Add the second node by adding it as a successor to the first node.
>>> network.add_successor(my_node, my_other_node)
```

The `network` object now contains both nodes, and a (directed) edge from `my_node` to `my_other_node`. Nodes can be accessed either from a `SupplyChainNetwork` object's `nodes` attribute (a list of nodes), or using its member function `get_node_from_index()`:

```
>>> n1 = network.nodes[0]
>>> n1.index
1
>>> n1.local_holding_cost
3
>>> n2 = network.get_node_from_index(2)
>>> n2.index
2
>>> n2.echelon_holding_cost
1.5
```

The `supply_chain_network` module contains functions for quickly building certain types of multiechelon networks. For example, `single_stage_system()` generates a single-node network:

```
>>> from stockpyl.supply_chain_network import single_stage_system
>>> network = single_stage_system(holding_cost=0.18,
    stockout_cost=0.70,
    demand_type='N',
    mean=50, standard_deviation=8,
    policy_type='BS',
    base_stock_level=56.6
  )
>>> network.nodes[0].stockout_cost
0.7
```

The `serial_system()` function generates a serial system. Its signature is

```
serial_system(num_nodes, node_order_in_system=None, node_order_in_lists=None,
    **kwargs)
```

The `node_order_in_system` parameter indicates the order of the node indices in the serial system; if omitted, the nodes are assumed to be indexed $0, \ldots,$ `num_nodes` $- 1$, with node 0 upstream and node `num_nodes` $- 1$ downstream. The `node_order_in_lists` parameter allows you to provide data in a different order if you wish. The `kwargs` parameters specify the attributes (data) for the nodes in the network. If they are provided, they must be either a dictionary, a list, or a singleton, with the following requirements:

- If the parameter is a dictionary, then the keys must contain the node indices and the values must contain the corresponding attribute values. If a given node index is contained in `node_order_in_system` (or in $0, \ldots,$ `num_nodes` $- 1$, if `node_order_in_system` is not provided) but is not a key in the dictionary, the attribute value is set to `None` for that node.
- If the parameter is a singleton, then the attribute is set to that value for all nodes.
- If the parameter is a list and `node_order_in_lists` is provided, `node_order_in_lists` must contain the same indices as `node_order_in_system` (if it is provided) or $0, \ldots,$ `num_nodes` $- 1$ (if it is not). The values in the list are assumed to correspond to the node indices in the order they are specified in `node_order_in_lists`. That is, the

value in slot k in the parameter list is assigned to the node with index `node_order_in_lists[k]`.

- If the parameter is a list and `node_order_in_lists` is not provided, the values in the list are assumed to correspond to nodes in the same order as `node_order_in_system` (or in $0, \ldots,$ `num_nodes` $- 1$, if `node_order_in_system` is not provided).

This scheme is meant to provide maximum flexibility, but it can also be confusing at first, so we recommend testing the network you build by querying it to double check, as we do in the following example:

```
>>> from stockpyl.supply_chain_network import serial_system
>>> # Build a 3-node serial system.
>>> network = serial_system(
    num_nodes=3,
    node_order_in_system=[2, 1, 0],
    node_order_in_lists=[0, 1, 2],
    local_holding_cost=[7, 4, 2],
    demand_type='N',
    mean=10,
    standard_deviation=2,
    policy_type='BS',
    base_stock_level=5
  )
>>> # Check the index of the (only) source node.
>>> network.source_nodes[0].index
2
>>> # Check the holding cost of the middle node (source node's successor).
>>> network.source_nodes[0].successors()[0].local_holding_cost
4
```

In this case, the network consists of nodes in the order $2 \rightarrow 1 \rightarrow 0$ (Figure 1), but the `local_holding_cost` parameter is provided as a list with node 0 first, then node 1, and then node 2. Once the network is constructed, we can query in various ways. In the code above, `network.source_nodes[0]` gets the first (actually, only) source node (i.e., a node with no upstream predecessors) and `succesors()` gets a list of that node's successor nodes.

Similarly, the `owmr_system()` and `mwor_system()` functions allow you to quickly build one-warehouse, multiretailer (OWMR) and multiwarehouse, one-retailer (MWOR) systems.

Finally, the `network_from_edges()` function allows you build a network with arbitrary topology by providing a list of the edges in the network, along with the data for each node. The function has the signature

```
network_from_edges(edges, node_order_in_lists=None, **kwargs)
```

The `edges` parameter is a list of edges, with each edge specified as a tuple `(a, b)`, where `a` is the index of the predecessor node and `b` is the index of the successor node. If edges is `None` or an empty list, a single-node network is created. The `node_order_in_lists` parameter works similarly to that in the `serial_system()` function, described previously. The kwargs parameters are handled similarly to `serial_system()` as well, with the modification that indices are matched to the indices in the edges list rather than to `node_order_in_system` or $0, \ldots,$ `num_nodes` $- 1$.

For example, the following code creates a network that represents the system depicted in Figure 2. The holding costs at nodes 1–4 are 4, 7, 2, and 1, respectively, and the stockout costs at nodes 1 and 2 are 20 and 50, respectively.

Figure 1. Three-stage serial system.

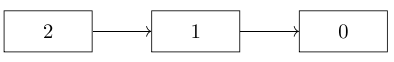
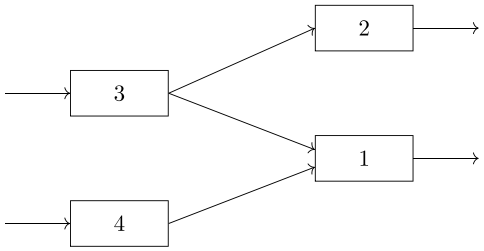


Figure 2. Four-stage general system.




```
>>> from stockpyl.supply_chain_network import network_from_edges
>>> # Create a network by specifying the edges.
>>> network1 = network_from_edges(
      edges=[(3, 1), (3, 2), (4, 1)],
      node_order_in_lists=[1, 2, 3, 4],
      local_holding_cost=[4, 7, 2, 1],
      stockout_cost=[20, 50, None, None],
      demand_type=['N', 'N', None, None],
      mean=[10, 15, None, None],
      standard_deviation=[2, 4, None, None]
    )
>>> # Print some details about each node in the network.
>>> for node in network1.nodes:
    print(f"Node {node.index}: h = {node.local_holding_cost}
      p = {node.stockout_cost} mu = {node.demand_source.mean}
      sigma = {node.demand_source.standard_deviation}")
Node 3: h = 2 p = None mu = None sigma = None
Node 1: h = 4 p = 20 mu = 10 sigma = 2
Node 2: h = 7 p = 50 mu = 15 sigma = 4
Node 4: h = 1 p = None mu = None sigma = None
```

Alternatively, we could construct the same network by constructing each node individually and adding them to the network, specifying the appropriate predecessors and successors. In this case, we need to explicitly set the `demand_source` attribute for the sink nodes (nodes that face external demand) and the `supply_type` attribute for the source nodes (nodes that have external supply). The `demand_source` is set to a `DemandSource` object; in particular, the demand is normally distributed at nodes 1 and 2, with different means and standard deviations. (See the online documentation for more information.) The `supply_type` attribute is set to `'U'` for nodes 3 and 4 to indicate unlimited supply. (This is currently the only type of supply supported in the `SupplyChainNode` class.)

```
>>> from stockpyl.supply_chain_network import SupplyChainNetwork
>>> from stockpyl.supply_chain_node import SupplyChainNode
>>> from stockpyl.demand_source import DemandSource
>>> # Build 4 nodes.
>>> node1 = SupplyChainNode(index=1, local_holding_cost=4, stockout_cost=20,
  demand_source=DemandSource(type='N', mean=10, standard_deviation=2))
>>> node2 = SupplyChainNode(index=2, local_holding_cost=7, stockout_cost=50,
  demand_source=DemandSource(type='N', mean=15, standard_deviation=4))
>>> node3 = SupplyChainNode(index=3, local_holding_cost=2, supply_type='U')
>>> node4 = SupplyChainNode(index=4, local_holding_cost=1, supply_type='U')
>>> # Build the network.
>>> network2 = SupplyChainNetwork()
>>> network2.add_node(node3)
>>> network2.add_successor(node3, node1)
>>> network2.add_successor(node3, node2)
>>> network2.add_predecessor(node1, node4)
>>> # Check that network2 equals the one we built using network_from_edges().
>>> network1.deep_equal_to(network2)
True
```

Instead of adding the nodes by successor, one can also add the nodes directly to the network and then add the edges in a separate step:

```
>>> network3 = SupplyChainNetwork()
>>> network3.add_node(node1)
>>> network3.add_node(node2)
>>> network3.add_node(node3)
>>> network3.add_node(node4)
>>> network3.add_edges_from_list([(3, 2), (3, 1), (4, 1)])
```

Another way to build `SupplyChainNetworks` is using Stockpyl's built-in instances. The `instances` module contains approximately 70 instances, most (but not all) of which are taken from FoSCT and named accordingly. The instances can be loaded using the `load_instance()` function by providing the instance name. A list of the built-in instances is provided in the online documentation for the `instances` module.

For example, the following code loads the "digital camera" network from figure 6.14 in FoSCT:

```
>>> from stockpyl.instance import load_instance
>>> instance = load_instance('figure_6_14')
```

This is equivalent to building the network manually:

```
>>> from stockpyl.supply_chain_network import SupplyChainNetwork
>>> from stockpyl.supply_chain_node import SupplyChainNode
>>> from stockpyl.demand_source import DemandSource
>>> from scipy import stats
>>> instance = SupplyChainNetwork()
>>> instance.add_node(SupplyChainNode(1, 'Raw_Material', instance,
  processing_time=2, local_holding_cost=0.01))
>>> instance.add_node(SupplyChainNode(2, 'Process_Wafers', instance,
  processing_time=3, local_holding_cost=0.03))
>>> instance.add_node(SupplyChainNode(3, 'Package_Test_Wafers', instance,
  processing_time=2, local_holding_cost=0.04))
>>> instance.add_node(SupplyChainNode(4, 'Imager_Base', instance,
  processing_time=4, local_holding_cost=0.06))
>>> instance.add_node(SupplyChainNode(5, 'Imager_Assembly', instance,
  processing_time=2, local_holding_cost=0.12))
>>> instance.add_node(SupplyChainNode(6, 'Ship_to_Final_Assembly', instance,
  processing_time=3, local_holding_cost=0.13))
>>> instance.add_node(SupplyChainNode(7, 'Camera', instance,
  processing_time=6, local_holding_cost=0.20))
>>> instance.add_node(SupplyChainNode(8, 'Circuit_Board', instance,
  processing_time=4, local_holding_cost=0.08))
>>> instance.add_node(SupplyChainNode(9, 'Other_Parts', instance,
  processing_time=3, local_holding_cost=0.04))
>>> instance.add_node(SupplyChainNode(10, 'Build_Test_Pack', instance,
  processing_time=2, local_holding_cost=0.50, external_outbound_cst=2,
  demand_source=DemandSource(type='N', mean=0, standard_deviation=10)))
>>> for n in instance.nodes:
>>>     n.demand_bound_constant = stats.norm.ppf(0.95)
>>> instance.add_edges_from_list([(1, 2), (2, 3), (3, 5), (4, 5), (5, 6),
  (7, 10), (6, 10), (8, 10), (9, 10)])
```

4.2. Local vs. Echelon Quantities

In a serial system indexed $N \rightarrow N-1 \rightarrow \cdots \rightarrow 1$, the *echelon* of stage j is defined as stage j and all stages downstream from j. The *echelon inventory level*, IL_j, consists of all items on hand in stages $j, \ldots, 1$, plus all items in transit between those stages, minus backorders at stage 1. Echelon-based quantities such as these are less intuitive than the *local* quantities we typically deal with, but they are more convenient mathematically, and the functions described in Section 4.3 use them.

The *local holding cost* at stage j, denoted h'_j, is the holding cost charged based on on-hand inventory at stage j (sometimes plus items in transit from stage j to $j-1$). This type of holding cost is intuitive, and is analogous to the holding cost in single-stage problems. On the other hand, in SSM models, it is often convenient to work with the *echelon holding cost*, denoted h_j and defined as

$$h_j = h'_j - h'_{j+1}, \tag{48}$$

with $h'_{N+1} \equiv 0$. One way to think about the echelon holding cost is that it represents the additional holding cost attributed to the value added to the product at stage j. Typically, echelon holding costs are nonnegative (meaning the value of the product, or the cost to store it, increases as we move downstream) because local holding costs tend to increase as we move downstream. However, negative echelon holding costs are possible. Local holding costs can be calculated from the echelon costs as follows:

$$h'_j = \sum_{i=j}^{N} h_j. \tag{49}$$

In an *echelon base-stock policy*, each stage j places an order so that its *echelon inventory position* (defined as its echelon inventory level, IL_j, plus any items on-order from stage $j+1$) equals its *echelon base-stock level*, S_j. Echelon base-stock policies are closely related to the more familiar *local base-stock policies*, in which we bring the local inventory position up to the local base-stock level. Indeed, the two types of policies are equivalent if we relate the echelon base-stock levels S_j and the local base-stock levels S'_j as follows:

$$S'_j = S_j - S_{j-1}, \tag{50}$$

$$S_j = \sum_{i=1}^{j} S'_i. \tag{51}$$

In (50), we define $S_0 \equiv 0$. This assumes that $S_j \geq S_{j-1}$; if not, we let $S^-_j = \min_{i \geq j}\{S_i\}$ and set $S'_j = S^-_j - S^-_{j-1}$.

The serial SSM functions described in Section 4.3 take echelon holding costs as input parameters and return echelon base-stock levels as outputs. GSM models tend to work with local quantities, not echelon ones, so the functions described in Section 4.4 take local holding costs and return local base-stock levels. The functions `local_to_echelon_base_stock_levels()` and `echelon_to_local_base_stock_levels()` in the `supply_chain_network` module convert back and forth between the two types of base-stock levels.

4.3. Serial SSM Systems

4.3.1. Optimization Functions. The `ssm_serial` module contains code to solve serial systems under the stochastic service model (SSM), either exactly, using the `optimize_base_stock_levels()` function (which implements the algorithm by Chen and Zheng [4], which in turn is based on the algorithm by Clark and Scarf [5]), or approximately, using the `newsvendor_heuristic()` function (which implements the newsvendor-based heuristic by Shang and Song [23]). Both functions assume a continuous-review system in which each stage follows an echelon base-stock policy (Section 4.2).

For either function, you may pass the instance data as individual parameters (costs, demand distribution, etc.) or a `SupplyChainNetwork`. Here is example 6.1 from FoSCT with the data passed as individual parameters. In this three-stage instance, the nodes are indexed $3 \to 2 \to 1$. The demand at stage 1 is distributed as $N(5, 1^2)$ per unit time. The lead times are $L_1 = L_2 = 1$ and $L_3 = 2$. Echelon holding costs are $(h_1, h_2, h_3) = (3, 2, 2)$. The stockout cost at stage 1 is $p = 37.12$ per unit time. We solve it using the exact algorithm implemented in the `optimize_base_stock_levels()` function (Section 4.3.2).

```
>>> from stockpyl.ssm_serial import optimize_base_stock_levels
>>> S_star, C_star = optimize_base_stock_levels(
    num_nodes=3,
    echelon_holding_cost=[2, 2, 3],
    lead_time=[2, 1, 1],
    stockout_cost=37.12,
    demand_mean=5,
    demand_standard_deviation=1
  )
>>> S_star
{3: 22.700237234889784, 2: 12.012332294949644, 1: 6.5144388073261155}
>>> C_star
47.668653127136345
```

Here is the same example, first building a `SupplyChainNetwork` and then passing that instead:

```
>>> from stockpyl.ssm_serial import optimize_base_stock_levels
>>> from stockpyl.supply_chain_network import serial_system
>>> example_6_1_network = serial_system(
    num_nodes=3,
    node_order_in_system=[3, 2, 1],
    echelon_holding_cost={1: 3, 2: 2, 3: 2},
    shipment_lead_time={1: 1, 2: 1, 3: 2},
    stockout_cost={1: 37.12, 2: 0, 3: 0},
    demand_type='N',
    mean=5,
    standard_deviation=1,
    policy_type='BS',
    base_stock_level=0,
  )
>>> S_star, C_star = optimize_base_stock_levels(network=example_6_1_network)
>>> S_star
{3: 22.700237234889784, 2: 12.012332294949644, 1: 6.5144388073261155}
>>> C_star
47.668653127136345
```

Example 6.1 is also a built-in instance in Stockpyl, so you can load it directly:

```
>>> from stockpyl.ssm_serial import optimize_base_stock_levels
>>> from stockpyl.instances import load_instance
>>> S_star, C_star = optimize_base_stock_levels(
    network=load_instance("example_6_1")
  )
>>> S_star
{3: 22.700237234889784, 2: 12.012332294949644, 1: 6.5144388073261155}
>>> C_star
47.668653127136345
```

The previous code snippets use the exact optimization algorithm. We can instead solve the instance using the newsvendor-based heuristic (Section 4.3.3):

```
>>> from stockpyl.ssm_serial import newsvendor_heuristic
>>> S_heur = newsvendor_heuristic(network=example_6_1_network)
>>> S_heur
{3: 22.634032391786285, 2: 12.027434723327854, 1: 6.490880975286938}
>>> # Evaluate the (exact) expected cost of the heuristic solution.
>>> from stockpyl.ssm_serial import expected_cost
>>> expected_cost(S_heur, network=example_6_1_network)
47.680099140842174
```

Table 11. Notation for `ssm_serial` module.

Math notation (FoSCT)	Python variable name (Stockpyl)	Type	Definition
N	`num_nodes`	Parameter	Number of nodes in serial system
h_j	`echelon_holding_cost`	Parameter	Echelon holding cost per item per unit time at stage j
p	`stockout_cost`	Parameter	Stockout cost per item per unit time at downstream node
L_j	`lead_time`	Parameter	Inbound shipment lead time at stage j
μ	`demand_mean`	Parameter	Mean demand per unit time at downstream node
σ	`demand_standard_deviation`	Parameter	Standard deviation of demand per unit time at downstream node
S_j	`S`	Decision variable	Echelon base-stock level at stage j
$g_N(S_N^*)$	`C_star`	Objective function	Optimal expected cost

The notation for the `ssm_serial` module is summarized in Table 11. The `optimize_base_stock_levels()` function has the signature

```
optimize_base_stock_levels(num_nodes=None, node_order_in_system=None,
  node_order_in_lists=None, echelon_holding_cost=None, lead_time=None,
  stockout_cost=None, demand_mean=None, demand_standard_deviation=None,
  demand_source=None, network=None, S=None, plots=False, x=None,
  x_num=1000, d_num=100, ltd_lower_tail_prob=1-stats.norm.cdf(4),
  ltd_upper_tail_prob=1-stats.norm.cdf(4),
  sum_ltd_lower_tail_prob=1-stats.norm.cdf(4),
  sum_ltd_upper_tail_prob=1-stats.norm.cdf(4))
```

and returns two parameters: `S_star` is a dictionary containing the optimal echelon base-stock levels, and `C_star` is the optimal expected cost.

The `newsvendor_heuristic()` function has the signature

```
newsvendor_heuristic(num_nodes=None, node_order_in_system=None,
    node_order_in_lists=None, echelon_holding_cost=None, lead_time=None,
    stockout_cost=None, demand_mean=None, demand_standard_deviation=None,
    demand_source=None, network=None, weight=0.5, round_type=None)
```

and returns a single parameter, `S_heur`.

By default, the nodes in the system are assumed to be indexed $N, \ldots, 1$, with node 1 at the downstream end, but this can be changed by providing either the `node_order_in_system` parameter or by providing a `SupplyChainNetwork` explicitly in the `network` parameter.

The node-based parameters `echelon_holding_cost` and `lead_time` can be specified either as a dictionary, a list, or a singleton. The requirements for specifying these parameters are the same as those described in Section 4.1, except that the default indexing of the nodes for `ssm_serial` functions is $1, \ldots,$ `num_nodes` instead of $0, \ldots,$ `num_nodes`-1.

The easiest way to describe the demand is to provide the `demand_mean` and `demand_standard_deviation` parameters; in this case, the demand is assumed to be normally distributed. Alternatively, you can provide a `DemandSource` object in the `demand_source` parameter, which is a bit more work but is more flexible. Finally, if you pass the whole network as a `SupplyChainNetwork` object in the network parameter, you can specify the demand distribution in that object.

4.3.2. Exact Algorithm. Let D_j be a random variable representing the lead-time demand (the demand during L_j time units) for stage j. Recall that p is the stockout cost at the downstream node and h_j is the echelon holding cost at node j. The exact algorithm by Chen and Zheng [4] and Clark and Scarf [5], implemented in the `optimize_base_stock_levels()` function, makes use of the following theorem (see Snyder et al. [28] and Zipkin [37] for a proof).

Theorem 1. *Let* $\underline{g}_0(x) = (p + h_1')x^-$. *For* $j = 1, \ldots, N$, *let*

$$\hat{g}_j(x) = h_j x + \underline{g}_{j-1}(x), \tag{52}$$

$$g_j(y) = \mathbb{E}[\hat{g}_j(y - D_j)], \tag{53}$$

$$S_j^* = \operatorname{argmin}\{g_j(y)\}, \tag{54}$$

$$\underline{g}_j(x) = g_j(\min\{S_j^*, x\}). \tag{55}$$

Then $\mathbf{S}^* = (S_j^*)_{j=1}^N$ *is the optimal base-stock vector and* $g_N(S_N^*)$ *is the corresponding optimal cost.*

The upshot of Theorem 1 is that we do not need to optimize all of the base-stock levels simultaneously. Instead, we can optimize sequentially, starting at stage 1 and moving upstream, using the recursive cost functions from the theorem. Moreover, $g_j(y)$ is convex for all j, so the minimization in (54) is straightforward. Conversely, the optimization is computationally expensive because the expectation in (53) must be calculated numerically.

Remark 6. Theorem 1 can also be used to evaluate the cost of a *given* vector of base-stock levels $\mathbf{S}$, rather than to find the optimal one. In that case, we simply skip the optimization step (54) and evaluate the functions using $\mathbf{S}$ instead of $\mathbf{S}^*$. You can ask `optimize_base_stock_levels()` to evaluate a given $\mathbf{S}$ by passing it in the optional input parameter S.

The `optimize_base_stock_levels()` function calculates these cost functions for a finite set of x and y values (see Section 4.3.4 for details of how Stockpyl truncates and discretizes the possible ranges). Therefore, although Theorem 1 provides an exact algorithm for optimizing the base-stock levels and evaluating their expected cost, it is only exact up to the truncation and discretization.

One final note about the exact algorithm: The $\hat{g}_j(x)$ function sometimes needs to be evaluated for x values outside the truncation range. (This happens, for example, when evaluating $g_j(y)$ for y values near the lower end of the x range, in which case we need to evaluate $\hat{g}_j(y - d)$ for d values that would take $y - d$ outside the x range.) One can show (see problem 6.13 in FoSCT) that $\hat{g}_j(\cdot)$ has the following limits:

$$\lim_{x \to -\infty} \hat{g}_j(x) = \sum_{i=1}^{j} h_i \left(x - \sum_{k=i}^{j-1} \mathbb{E}[D_k] \right) - (p + h_1') \left(x - \sum_{k=1}^{j-1} \mathbb{E}[D_k] \right), \tag{56}$$

$$\lim_{x \to +\infty} \hat{g}_j(x) = \begin{cases} h_j x + g_{j-1}(S_{j-1}^*), & \text{if } j > 1 \\ h_j x, & \text{if } j = 1 \end{cases}. \tag{57}$$

Therefore, if `optimize_base_stock_levels()` needs to evaluate $\hat{g}_j(x)$ for x less than the lower end of the x range, it calculates it using (56) instead. We never need to evaluate $\hat{g}_j(x)$ for x greater than the upper end of the x range, but `optimize_base_stock_levels()` calculates (57) anyway, so that it can be plotted if the optional input `parameter plots=True`.

4.3.3. Newsvendor-Based Heuristic. The `newsvendor_heuristic()` function implements the heuristic by Shang and Song [23] for optimizing the base-stock levels. This heuristic approximates each stage in the serial system as a newsvendor. This is only approximate because a newsvendor faces deterministic lead times, whereas a stage in a serial system sees stochastic lead times due to upstream stockouts.

In particular, let $\tilde{D}_j$ be the lead-time demand for a single-stage system with lead time $\sum_{i=1}^{j} L_i$, and let $\tilde{F}_j(\cdot)$ be its cdf. Consider two newsvendor systems for stage j: one has holding cost h_j and the other has holding cost $\sum_{k=1}^{j} h_k$, where h_j is the *echelon* holding cost at stage j. Both systems have a stockout cost of $p + \sum_{i=j+1}^{N} h_i$ and a demand cdf of $\tilde{F}_j(\cdot)$. The optimal base-stock levels of the two systems are given by

$$S_j^l = \tilde{F}_j^{-1}\left(\frac{p + \sum_{i=j+1}^{N} h_i}{p + \sum_{i=1}^{N} h_i}\right), \tag{58}$$

and

$$S_j^u = \tilde{F}_j^{-1}\left(\frac{p + \sum_{i=j+1}^{N} h_i}{p + \sum_{i=j}^{N} h_i}\right), \tag{59}$$

respectively. (The two right-hand sides differ only in the lower limit of the sum in the denominator.) These newsvendor-based base-stock levels provide bounds on the optimal base-stock levels (Shang and Song [23]).

Theorem 2. *For any j and y,*

$$S_j^l \le S_j^* \le S_j^u.$$

Therefore, we can approximate S_j^* using a weighted average of S_j^l and S_j^u:

$$\tilde{S}_j = \eta \tilde{F}_j^{-1}\left(\frac{p + \sum_{i=j+1}^{N} h_i}{p + \sum_{i=j}^{N} h_i}\right) + (1-\eta)\tilde{F}_j^{-1}\left(\frac{p + \sum_{i=j+1}^{N} h_i}{p + \sum_{i=1}^{N} h_i}\right) \tag{60}$$

for $0 \le \eta \le 1$. In fact, Shang and Song [23] suggest using $\eta = \frac{1}{2}$ and report that the resulting approximation is very accurate, typically with less than 1% error. It is also extremely fast because it only requires calculating $2N$ inverse cdfs.

4.3.4. Truncation and Discretization. Consider a single-stage inventory system under continuous review. It is well known that at any time t,

$$IL(t + L) = IP(t) - D(L),$$

where L is the lead time, $IL(t+L)$ is the inventory level (on-hand minus backorders) at time $t+L$, $IP(t)$ is the inventory position (inventory level plus on-order items) at time t, and $D(L)$ is the (random) demand during L time units. (Zipkin [37] refers to this as a "conservation-of-flow" equation.) If the stage is using a base-stock policy, then, by definition, the inventory position always equals the base-stock level, S:

$$IL(t + L) = S - D(L).$$

In steady state, we can write

$$IL = S - D_L, \tag{61}$$

where IL is a steady-state variable representing the inventory level, S is the base-stock level, and D_L is the lead-time demand.

The recursive cost functions evaluated by `optimize_base_stock_levels()` (see Theorem 1) have arguments x (analogous to the inventory level IL) and y (analogous to the base-stock level S). Unfortunately, we do not know a priori how large or small x and S might be. In the case of S, this is because we have not optimized yet. For x, if the lead-time demand D_L can be arbitrarily large, then IL can be arbitrarily *small*; because we do not know S before we begin the optimization, we do not know in advance how *large* IL can be. Moreover, if D_L has a

continuous distribution, then x also takes values in a continuous range. Of course, we can only calculate the cost functions for a finite number of x values. Therefore, the `optimize_base_stock_levels()` function truncates and discretizes both the range of possible inventory positions and the possible demand values.

Stockpyl uses (61) and bounds on the demand in order to derive approximate bounds on x and S. In particular, let D_Σ be a random variable representing the demand over $\sum_{j=1}^{N} L_j$ time units. Call this the "sum-of-lead-time demand distribution" and denote its mean μ_Σ and its cdf $F_\Sigma(\cdot)$. The variable D_Σ is a sort of worst-case distribution for D_L at any stage: If no stages hold any inventory, then stage 1 would face a lead time of $\sum_{j=1}^{N} L_j$ and therefore a lead-time demand of D_Σ. We use this worst-case distribution to derive bounds on S and x at all nodes. If we have (approximate) bounds

$$\texttt{sum\_ltd\_lo} \le \mathrm{D}_\Sigma \le \texttt{sum\_ltd\_hi} \tag{62}$$

on D_Σ, they provide *approximate* (but conservative) bounds on the range of inventory levels x we need to consider.

In particular, if $S < \texttt{sum\_ltd\_lo}$, then $IL < 0$ at all times. Although it is theoretically possible for this to be optimal, it almost never is; therefore, `sum_ltd_lo` is a reasonable lower bound on S. Similarly, if $S > \texttt{sum\_ltd\_hi}$, then we never have stockouts ($IL > 0$), but we still hold inventory, so setting $S = \texttt{sum\_ltd\_hi}$ would be cheaper; therefore, `sum_ltd_hi` is a reasonable upper bound on S.

Therefore, we have bounds

$$\texttt{sum\_ltd\_lo} \le S \le \texttt{sum\_ltd\_hi} \tag{63}$$

on S. Given (61), (62), and (63), we obtain the following bounds on x:

$$\texttt{sum\_ltd\_lo} - \texttt{sum\_ltd\_hi} \le x \le \texttt{sum\_ltd\_hi}. \tag{64}$$

It remains to determine the bounds `sum_ltd_lo` and `sum_ltd_hi`. If the support of the demand distribution is infinite, then Stockpyl sets

$$\texttt{sum\_ltd\_lo} = F_\Sigma^{-1}(\texttt{sum\_ltd\_lower\_tail\_prob}),$$
$$\texttt{sum\_ltd\_hi} = F_\Sigma^{-1}(\texttt{sum\_ltd\_upper\_tail\_prob}),$$

where `sum_ltd_lower_tail_prob` and `sum_ltd_upper_tail_prob` are optional input parameters that default to $1 - \Phi(4) \approx 3.17 \times 10^{-5}$. That is, we trim the distribution of D_Σ so that the probability in the lower and upper tails equals `sum_ltd_lower_tail_prob` and `sum_ltd_upper_tail_prob`, respectively.

Once we have truncated the range of x values via (64), we still need to discretize this truncated range, which we do by taking `x_num` + 1 equally spaced points in [`sum_ltd_lo`–`sum_ltd_hi`, `sum_ltd_hi`], where `x_num` is an optional input parameter that defaults to 1,000.

The resulting set of discrete points is used both as the set of values of x for which to evaluate the cost functions and the set of values of y to consider in the arg min in (54). Alternatively, you can bypass Stockpyl's truncation and discretization and explicitly specify the set of x values to use by providing the optional input parameter `x`, which is simply a list of x values to use as the truncated and discretized set.

In addition, the lead-time demand distribution at each stage j is truncated at [`ltd_lo`, `sum_ltd_hi`], where

$$\texttt{ltd\_lo} = F_j^{-1}(\texttt{ltd\_lower\_tail\_prob})$$
$$\texttt{ltd\_hi} = F_j^{-1}(\texttt{ltd\_upper\_tail\_prob}),$$

$F_j(\cdot)$ is the cdf of the lead-time demand distribution at stage j, and `ltd_lower_tail_prob` and `ltd_upper_tail_prob` are optional input parameters that default to $1 - \Phi(4) \approx 3.17 \times 10^{-5}$.

That range is then discretized to `d_num` equally spaced points, where `d_num` is an optional input parameter that defaults to 100.

Stockpyl uses the discretized demand distribution to evaluate $g_j(y)$ in (53) for a given value of y. If, for a given demand value d, $y - d$ is not in the set of x values for which $\hat{g}_j(x)$ has been calculated, the nearest x value to $y - d$ is used.

If the demand has finite support, then we do not truncate the demand distribution. If it has all-integer support, then we do not discretize it; the solutions returned will also be integer.

Remark 7. In general, larger values of `x_num` and `d_num` and smaller values of `ltd_lower_tail_prob`, `ltd_upper_tail_prob`, `sum_ltd_lower_tail_prob`, and `sum_ltd_upper_tail_prob` result in more accurate solutions but longer run times.

4.4. Serial or Tree GSM Systems

4.4.1. Overview and Assumptions. GSM models assume a periodic-review system in which each stage follows a local base-stock policy. In GSM systems, we assume that each stage offers a CST to its customers (which may be downstream stages or external demands) within which it promises to satisfy each order. The CSTs are the decision variables in the optimization model. To ensure that each stage meets its CST, the GSM model assumes that the external demands are bounded. In particular, if the demands in one time period are distributed as $N(\mu, \sigma^2)$, then we assume that the total demand in any τ time periods is less than or equal to

$$D(\tau) = \mu\tau + z_\alpha \sigma \sqrt{\tau} \tag{65}$$

for some constant α. If the demand in a given τ-period interval exceeds $D(\tau)$, the excess demands are assumed to be handled outside the system, for example, by outsourcing.

Remark 8. It is tempting to set α to something large so that the demand bound is quite loose and the demand model is more accurate. However, as we will see later, the base-stock levels are directly related to α, so setting α to a large value will also result in unnecessarily large base-stock levels and resulting holding costs. This tension is one of the main disadvantages of the GSM approach.

The GSM approach was first discussed by Kimball [18]. Simpson [25] applied it to serial systems, and Inderfurth [16] (see also Graves [8]) discussed how to solve it using DP; this is the basis for the approach implemented in the `gsm_serial` module. For pure assembly and distribution systems, see Inderfurth [16], Minner [20], and Inderfurth and Minner [17]. Graves and Willems [9, 10] present a DP model for tree systems, that is, networks that have no directed cycles; this is the algorithm implemented in the `gsm_tree module`. Magnanti et al. [19] present an algorithm based on mixed-integer programming (MIP) for solving general systems.

4.4.2. Serial Systems. First consider a serial network indexed $N \longrightarrow \cdots \longrightarrow 1$. At each stage i, let T_i be the *processing time* at i, representing the number of periods required to perform the activities at the stage. (GSM models use processing times T at the nodes rather than lead times L on the edges.) The T_i are constants. Furthermore, let S_i be the *outbound CST* that stage i promises its customers, and let SI_i be the *inbound CST* that stage i receives from its suppliers.[4] The inbound CST at j is simply the outbound CST at its predecessor. (Table 12 summarizes the notation.)

Node 1 faces external demand and has a CST of s_1; this is a constant and not a decision variable, representing the contract agreed to by the firm and the external customer. Similarly, stage N has a fixed inbound CST si_N from its (external) supplier. The optimization problem finds S_i for $2 \le i \le N$.

The quantity $SI_i + T_i - S_i$ is called the *net lead time*; it represents the number of periods that stage i's inventory must cover. In particular, if stage i uses a (local) base-stock level of

$$y_i = \mu(SI_i + T_i - S_i) + z_\alpha \sigma \sqrt{SI_i + T_i - S_i}, \tag{66}$$

then the stage will always be able to meet its CST.

Table 12. Notation for `gsm_serial` module.

Math notation (FoSCT)	Python variable name (Stockpyl)	Type	Definition
N	`num_nodes`	Parameter	Number of nodes in system
h_i	`local_holding_cost`	Parameter	Local holding cost per item per period at stage j
T_i	`lead_time`	Parameter	Processing time at stage i
μ	`demand_mean`	Parameter	Mean demand per period at stage 1
σ	`demand_standard_deviation`	Parameter	Standard deviation of demand per period at stage 1
z_α	`demand_bound_constant`	Parameter	Constant to use for setting demand bound
s_1	`external_outbound_cst`	Parameter	Outbound CST at stage 1
si_N	`external_inbound_cst`	Parameter	Inbound CST at stage N
S_i	`opt_cst`	Decision variable	Outbound committed service time (CST) at stage i
$g_N(S_N^*)$	`opt_cost`	Objective function	Optimal expected cost

Remark 9. Equation (66) gives us a one-to-one relationship between the CSTs and the base-stock levels. Although the CSTs are the decision variables in a GSM model, it is usually more intuitive to convert these to base-stock levels for managerial purposes.

Equation (66) is a consequence of the demand bound (65) and an equation similar to (61). The safety stock (the expected inventory level at the end of a period) at stage i is approximately

$$z_\alpha \sigma \sqrt{SI_i + T_i - S_i}, \tag{67}$$

so the total expected cost of the system, as a function of the vector $\mathbf{S} = (S_1, \ldots, S_N)$ of CSTs, is

$$g(\mathbf{S}) = \sum_{i=1}^{N} h_i z_\alpha \sigma \sqrt{SI_i + T_i - S_i}, \tag{68}$$

where h_i is the (local) holding cost at stage i.

The `optimize_committed_service_times()` function in the `gsm_serial` module implements the DP by Inderfurth [16] for serial GSM systems. Let $\theta_k(SI)$ be the optimal cost in stages $1, \ldots, k$ if stage k receives an inbound CST of SI. Then $\theta_k(SI)$ can be computed recursively as follows:

$$\theta_1(SI) = h_1 z_\alpha \sigma \sqrt{SI + T_1 - s_1}, \tag{69}$$

$$\theta_k(SI) = \min_{0 \le S \le SI + T_k} \{h_k z_\alpha \sigma \sqrt{SI + T_k - S} + \theta_{k-1}(S)\}, \quad k \ge 2. \tag{70}$$

The `optimize_committed_service_times()` function has the following signature:

```
optimize_committed_service_times(num_nodes=None, local_holding_cost=None,
  processing_time=None, demand_bound_constant=None,
  external_outbound_cst=None, external_inbound_cst=None,
  demand_mean=None, demand_standard_deviation=None, demand_source=None,
  network=None)
```

It returns two parameters: `opt_cst` and `opt_cost`.

You may specify the network either by its individual parameters (`num_nodes`, `local_holding_cost`, etc.) or in the `network` parameter as a `SupplyChainNetwork`. If you specify individual parameters, the nodes of the network must be indexed $N \longrightarrow \cdots \longrightarrow 1$. (If you

specify it in the `network` parameter, the nodes may be indexed in any way.) The node-specific parameters (`local_holding_cost`, `processing_time`, and `demand_bound_constant`) may be a dictionary, a list, or a singleton, with the following requirements:

- If the parameter is a dictionary, its keys must equal $1, \ldots,$ `num_nodes`, each corresponding to a node index.
- If the parameter is a list, it must have length `num_nodes`; the nth entry in the list corresponds to the node with index `n` + 1.
- If the parameter is a singleton, all nodes will have that parameter set to the singleton value.

These requirements are less flexible than those for the `ssm_serial` module (Section 4.3.1), but you can always build a `SupplyChainNetwork` in whatever way you like and then pass it in the `network` parameter, which offers full flexibility.

You must either pass the `demand_mean` and `demand_parameters`, or pass a `DemandSource` object in the `demand_source` parameter.

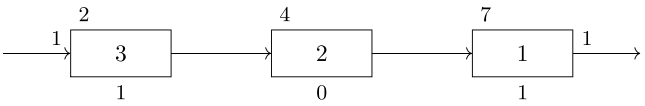
Here is example 6.3 from FoSCT, passing the data as individual parameters. The network for this example is shown in Figure 3. The numbers below the stages are the processing times T_i. The number on the inbound arrow to stage 3 indicates that $si_3 = 1$, whereas the outbound number from stage 1 indicates that the fixed CST $s_1 = 1$. The holding costs at stages 1, 2, and 3 are 7, 4, and 2, respectively, and are noted above each stage. Finally, $z_\alpha = \sigma_i = 1$ at all stages.

```
>>> from stockpyl.gsm_serial import optimize_committed_service_times
>>> opt_cst, opt_cost = optimize_committed_service_times(
    num_nodes=3,
    local_holding_cost=[7, 4, 2],
    processing_time=[1, 0, 1],
    demand_bound_constant=1,
    external_outbound_cst=1,
    external_inbound_cst=1,
    demand_mean=0,
    demand_standard_deviation=1
  )
>>> opt_cst
{3: 0, 2: 0, 1: 1}
>>> opt_cost
2.8284271247461903
```

Or passing a `SupplyChainNetwork`:

```
>>> from stockpyl.gsm_serial import optimize_committed_service_times
>>> from stockpyl.supply_chain_network import network_from_edges
>>> example_6_3_network = network_from_edges(
    edges=[(3, 2), (2, 1)],
    node_order_in_lists=[1, 2, 3],
    processing_time=[1, 0, 1],
    external_inbound_cst=[None, None, 1],
    local_holding_cost=[7, 4, 2],
    demand_bound_constant=1,
    external_outbound_cst=[1, None, None],
    demand_type=['N', None, None],
    mean=0,
    standard_deviation=[1, 0, 0]
  )
>>> optimize_committed_service_times(network=example_6_3_network)
({3: 0, 2: 0, 1: 1}, 2.8284271247461903)
```

Figure 3. Example network for SSSPP DP algorithm for serial systems.



Note. Numbers below stages are processing times T_i; numbers above stages are holding costs h_i; and numbers on arrows are required inbound or outbound CSTs.

Or loading the instance directly:

```
>>> from stockpyl.instances import load_instance
>>> optimize_committed_service_times(network=load_instance("example_6_3"))
({3: 0, 2: 0, 1: 1}, 2.8284271247461903)
```

4.4.3. Tree Systems. Now consider a tree system, that is, a network with no directed cycles. Let A be the set of arcs (edges) in the network. The network may have multiple demand stages (stages that face external demand); each faces $N(\mu_i, \sigma_i^2)$ demand per period and has a fixed outbound CST of s_i. Similarly, it may have multiple supply stages (stages that receive external supply); each has a fixed inbound CST of si_i.

Stages that are not demand stages see demand that is derived from their successor stages. The resulting demand is normally distributed with mean and standard deviation

$$\mu_i = \sum_{(i,j)\in A} \mu_j, \tag{71}$$

$$\sigma_i = \sqrt{\sum_{(i,j)\in A} \sigma_j^2}. \tag{72}$$

If stage i has more than one successor, we assume that it quotes the same CST to each of them. If stage i has more than one predecessor, then

$$SI_i = \max_{(j,i)\in A} \{S_j\}, \tag{73}$$

because stage i requires all predecessor units to be present before it begins processing.

The relationship between the CSTs and the base-stock levels is the same as in the serial case (66) as is the objective function (68). The `optimize_committed_service_times()` function in the `gsm_tree` module implements the DP by Graves and Willems [9, 10] for tree GSM systems. This DP is considerably more complicated than the one for serial systems, so we omit a detailed discussion here.

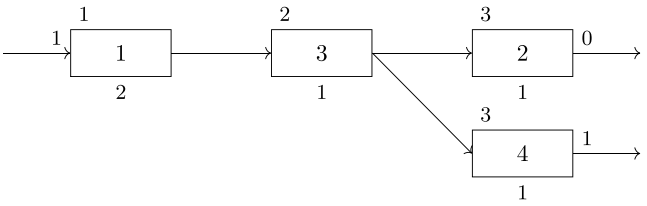
The `optimize_committed_service_times()` has the following signature:

```
optimize_committed_service_times(tree)
```

It returns two parameters, `opt_cst` and `opt_cost`. This function does not allow you to specify the instance as individual parameters; instead, you must provide a `SupplyChainNetwork` object in the `tree` parameter.

The four-node system in Figure 4 is from example 6.5 in FoSCT. The numbers below the stages are the processing times T_i. The number on the inbound arrow to stage 1 indicates that $si_1 = 1$, whereas the outbound numbers from stages 2 and 4 indicate the fixed CSTs s_i. The holding costs are noted above each stage and are equal to one at stage 1, two at stage 3, and three at stages 2 and 4. At all stages, $z_\alpha = 1$ and $\sigma_2 = \sigma_4 = 1$; then, $\sigma_1 = \sigma_3 = \sqrt{2}$. The

Figure 4. Example network for SSSPP DP algorithm for tree systems.



Note. Numbers below stages are processing times T_i; numbers above stages are holding costs h_i; and numbers on arrows are required inbound or outbound CSTs.

following code snippet solves this instance:

```
>>> from stockpyl.gsm_tree import optimize_committed_service_times
>>> from stockpyl.supply_chain_network import network_from_edges
>>> example_6_5_network = network_from_edges(
    edges=[(1, 3), (3, 2), (3, 4)],
    node_order_in_lists=[1, 2, 3, 4],
    processing_time=[2, 1, 1, 1],
    external_inbound_cst=[1, None, None, None],
    local_holding_cost=[1, 3, 2, 3],
    demand_bound_constant=[1, 1, 1, 1],
    external_outbound_cst=[None, 0, None, 1],
    demand_type=[None, 'N', None, 'N'],
    mean=0,
    standard_deviation=[None, 1, None, 1]
  )
>>> opt_cst, opt_cost = optimize_committed_service_times(tree=
                                        example_6_5_network)
>>> opt_cst
{1: 0, 3: 0, 2: 0, 4: 1}
>>> opt_cost
8.277916867529369
```

Example 6.5 is a built-in instance in Stockpyl, so it can be loaded directly instead:

```
>>> from stockpyl.instances import load_instance
>>> optimize_committed_service_times(tree=load_instance("example_6_5"))
({1: 0, 3: 0, 2: 0, 4: 1}, 8.277916867529369)
```

4.5. General MEIO Systems

For MEIO systems with arbitrarily topology (not necessarily serial or tree systems), the `meio_general` module can optimize base-stock levels approximately using relatively brute-force approaches—either coordinate descent or enumeration. These heuristics tend to be quite slow and not particularly accurate, but they are sometimes the best methods available for complex systems that are not well solved in the literature.

For both approaches, you may provide an objective function that will be used to evaluate each candidate solution, or you may omit the objective function and the algorithm will evaluate solutions using simulation. Obviously, evaluating using simulation is typically much slower than using an objective function. (See the online documentation for more information about Stockpyl's simulation features.)

Both approaches assume that each stage follows a base-stock policy, and they attempt to optimize the base-stock levels. (Future versions of Stockpyl may allow a wider range of policies.) Both functions use local, not echelon, costs and base-stock levels.

The `meio_by_enumeration()` function optimizes an MEIO instance by enumerating combinations of values of the base-stock levels. It has signature

```
meio_by_enumeration(network, base_stock_levels=None, truncation_lo=None,
  truncation_hi=None, discretization_step=None, discretization_num=None,
  groups=None, objective_function=None, sim_num_trials=10,
  sim_num_periods=1000, sim_rand_seed=None, progress_bar=True,
  print_solutions=False)
```

and returns two parameters, `best_S` (the best base-stock levels found) and `best_cost` (the corresponding expected cost).

The function provides several ways to control how the enumeration is performed, via optional input parameters. In particular,

- `base_stock_levels` is a dictionary indicating, for each node index, the base-stock levels to test in the enumeration.
- `truncation_lo` and `truncation_hi` indicate the low and high ends of the truncation range for base-stock levels to test for each node. You can specify either a singleton (in which case every node uses the same value) or a dictionary whose keys are the node indices.
- `discretization_step` indicates the interval width to use for discretizing the base-stock levels to test for each node. The `discretization_num` parameter indicates how many intervals to use. Only one should be provided. Either can be a dictionary or singleton.
- `groups` is a list of sets, each of which contains indices of nodes that should have the same base-stock level. This speeds the optimization since the base-stock levels for the nodes in a given group do not have to be optimized individually. Any nodes not contained in any set in the list are optimized individually.
- `objective_function` is the function to use to evaluate a given solution. The function must take a single argument, the dictionary of base-stock levels, and return a single output, the expected cost per period. If this parameter is omitted, simulation will be used instead to evaluate solutions.
- `sim_num_trials`, `sim_num_periods`, and `sim_rand_seed` can be used to customize the simulation used to evaluate solutions.
- `progress_bar` specifies whether a progress bar is displayed during the search.
- `print_solutions` specifies whether to display each solution and its cost during the search.

In the following code snippet, we solve the three-node serial SSM system in example 6.1 from FoSCT using enumeration. We specify upper and lower bounds on the base-stock levels to test for each node and evaluate each candidate set of base-stock levels using simulation (three trials, 100 periods per trial—a very coarse approximation since the simulation runs are very small).

```
>>> from stockpyl.meio_general import meio_by_enumeration
>>> from stockpyl.instances import load_instance
>>> example_6_1_network = load_instance("example_6_1")
>>> best_S, best_cost = meio_by_enumeration(
    network=example_6_1_network,
    truncation_lo={1: 5, 2: 4, 3: 10},
    truncation_hi={1: 7, 2: 7, 3: 12},
    sim_num_trials=3,
    sim_num_periods=100,
    sim_rand_seed=42
  )
>>> best_S
{1: 7, 2: 5, 3: 11}
>>> best_cost
47.994095842987605
```

Recall that the optimal local base-stock levels are 6.51, 5.50, and 10.69 (converting from the echelon base-stock levels in Section 4.3.1), with an optimal cost of 47.69. Therefore, the solution found by enumeration is quite good—It is only 0.7% worse than optimal. On the other hand, we stacked the deck by giving the function a pretty narrow range of base-stock levels to test. For a fairer experiment, we would test a broader range of base-stock levels, but then the execution would be even slower.

Alternatively, we can provide an objective function. This is more accurate and faster than evaluating solutions using simulation, but if the objective function must be evaluated numerically (as it does for serial SSM systems), speed and accuracy are still nontrivial issues to consider. In the following code, we first define an objective function using a Python lambda function; it evaluates each solution by first converting the local base-stock levels to echelon and then passing them to the `expected_cost()` function for serial SSM systems, which requires echelon base-stock levels as inputs. The discretization settings used below (`x_num=100, d_num=10`) are relatively coarse, producing inaccurate solutions but fairly quickly.

```
>>> from stockpyl.ssm_serial import expected_cost
>>> from stockpyl.supply_chain_network import
                                        local_to_echelon_base_stock_levels
>>> obj_fcn = lambda S: expected_cost(local_to_echelon_base_stock_levels(
    example_6_1_network, S), network=example_6_1_network, x_num=100, d_num=10
  )
>>> best_S, best_cost = meio_by_enumeration(
    network=example_6_1_network,
    truncation_lo={1: 5, 2: 4, 3: 10},
    truncation_hi={1: 7, 2: 7, 3: 12},
    objective_function=obj_fcn
  )
>>> best_S
{1: 7, 2: 5, 3: 11}
>>> best_cost
48.21449789525488
```

The `meio_by_coordinate_descent()` function optimizes (approximately) using coordinate descent. In principle, coordinate descent will find the globally optimal solution if the objective function is jointly convex in the base-stock levels, but if solutions are evaluated using simulation, then there are no guarantees. Like enumeration, coordination can be quite slow and not particularly accurate.

The function's signature is as follows:

```
meio_by_coordinate_descent(network, initial_solution=None,
  search_lo=None, search_hi=None, groups=None, objective_function=None,
  sim_num_trials=10, sim_num_periods=1000, sim_rand_seed=None, tol=1e-2,
  line_search_tol=1e-4, verbose=False)
```

It returns two parameters: `best_S` and `best_cost`. The optional input parameters give you control over how the search is performed. Several are identical to those for `meio_by_enumeration()` (see earlier discussion). Others include the following:

- `initial_solution` is the starting solution for the coordinate descent search, specified as a dictionary whose keys are node indices and whose values are base-stock levels. If omitted, the initial solution will be set automatically.
- `search_lo` and `search_hi` indicate the low and high ends of the search range for the base-stock levels at each node. You can specify either a singleton (in which case every node uses the same value) or a dictionary whose keys are the node indices.
- `tol` is a tolerance; the algorithm terminates when an iteration fails to improve the objective function by more than `tol`.

- `line_search_tol` is a tolerance to use for the line search (golden section search) component of the coordinate descent algorithm.
- `verbose` indicates whether messages should be displayed at each iteration.

In the following code snippets, we again solve example 6.1, this time using coordinate descent. In the first snippet, we use simulation, while in the second, we provide an objective function.

```
>>> from stockpyl.meio_general import meio_by_coordinate_descent
>>> from stockpyl.ssm_serial import expected_cost
>>> from stockpyl.supply_chain_network import
                                    local_to_echelon_base_stock_levels
>>> best_S, best_cost = meio_by_coordinate_descent(
    network=example_6_1_network,
    search_lo={1: 5, 2: 4, 3: 10},
    search_hi={1: 7, 2: 7, 3: 12},
    sim_num_trials=3,
    sim_num_periods=100,
    sim_rand_seed=762
  )
>>> best_S
{1: 6.5135882931757045, 2: 5.758375187638261, 3: 10.0631099655841}
>>> best_cost
46.90365729191992
```

```
>>> obj_fcn = lambda S: expected_cost(local_to_echelon_base_stock_levels(
  example_6_1_network, S), network=example_6_1_network, x_num=20, d_num=10
  )
>>> best_S, best_cost = meio_by_coordinate_descent(
    example_6_1_network,
    search_lo={1: 5, 2: 4, 3: 10},
    search_hi={1: 7, 2: 7, 3: 12},
    objective_function=obj_fcn
  )
>>> best_S
{1: 5.892036436905893, 2: 5.995771265226075, 3: 11.99995914365099}
>>> best_cost
62.33491040676202
```

5. Feedback and Contributions

Feedback, suggestions, feature requests, and bug reports about Stockpyl are always welcome. The preferred method is to post an issue on the Stockpyl GitHub repo at https://github.com/LarrySnyder/stockpyl/issues. Alternatively, feel free to email me.

If you wish to contribute to the Stockpyl project, you can browse the outstanding issues at the previous link or develop code to address an aspect you are interested in. Once your code is ready for review, please submit a pull request at https://github.com/LarrySnyder/stockpyl/pulls. Thank you for your contributions.

Endnotes

[1]At the time of this writing, this feature has not yet been implemented in Stockpyl.

[2]The Wagner-Whitin problem is sometimes referred to as the dynamic economic lot-sizing or uncapacitated lot-sizing model.

[3]Other forms of supply uncertainty, such as lead time uncertainty or capacity uncertainty, are somewhat less well studied in the literature and are not implemented in Stockpyl.

[4]Unfortunately, the SSM literature has tended to use S to represent a base-stock level, whereas the GSM literature has tended to use S to represent a CST. We will stay consistent with these two bodies of literature, which means using S in two different ways in Section 4.3 versus Section 4.4.

References

[1] K. J. Arrow, T. Harris, and J. Marschak. Optimal inventory policy. *Econometrica* 19(3): 250–272, 1951.

[2] Z. Atan and L. V. Snyder. Inventory strategies to manage supply disruptions. Gurnani H., Mehrotra A., Ray S., eds. *Disruptions in Supply Chains: Theory and Practice of Managing Risk.* Springer-Verlag, London, 115–139, 2012.

[3] E. Berk and A. Arreola-Risa. Note on "Future supply uncertainty in EOQ models". *Naval Research Logistics* 41(1):129–132, 1994.

[4] F. Chen and Y.-S. Zheng. Lower bounds for multi-echelon stochastic inventory systems. *Management Science* 40(11):1426–1443, 1994.

[5] A. J. Clark and H. Scarf. Optimal policies for a multi-echelon inventory problem. *Management Science* 6(4):475–490, 1960.

[6] R. Ehrhardt and C. Mosier. A revision of the power approximation for computing (*s, S*) policies. *Management Science* 30(5):618–622, 1984.

[7] A. Federgruen and Y.-S. Zheng. An efficient algorithm for computing an optimal (*r, Q*) policy in continuous review stochastic inventory systems. *Operations Research* 40(4):808–813, 1992.

[8] S. C. Graves. Safety stocks in manufacturing systems. *Journal of Manufacturing and Operations Management* 1:67–101, 1988.

[9] S. C. Graves and S. P. Willems. Optimizing strategic safety stock placement in supply chains. *Manufacturing & Service Operations Management* 2(1):68–83, 2000.

[10] S. C. Graves and S. P. Willems. Erratum: Optimization strategic safety stock placement in supply chains. *Manufacturing and Service Operations Management* 5(2):176–177, 2003.

[11] S. C. Graves and S. P. Willems. Supply chain design: Safety stock placement and supply chain configuration. In de Kok AG, Graves SC, eds. *Supply Chain Management: Design, Coordination and Operation*, vol. 11 of Handbooks in Operations Research and Management Science. Elsevier, Amsterdam, 95–132, 2003.

[12] A. Grosfeld-Nir and Y. Gerchak. Multiple lotsizing in production to order with random yields: Review of recent advances. *Annals of Operations Research* 126(1–4):43–69, 2004.

[13] R. Güllü, E. Onol, and N. Erkip. Analysis of a deterministic demand production/inventory system under nonstationary supply uncertainty. *IIE Transactions* 29(8):703–709, 1997.

[14] G. Hadley and T. M. Whitin. *Analysis of Inventory Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1963.

[15] F. W. Harris. How many parts to make at once. *Operations Research* 38(6):947–950, 1990.

[16] K. Inderfurth. Safety stock optimization in multi-stage inventory systems. *International Journal of Production Economics* 24:103–113, 1991.

[17] K. Inderfurth and S. Minner. Safety stocks in multi-stage inventory systems under different service measures. *European Journal of Operational Research* 106:57–73, 1998.

[18] G. E. Kimball. General principles of inventory control. *Journal of Manufacturing and Operations Management* 1:119–130, 1988.

[19] T. L. Magnanti, Z.-J. M. Shen, J. Shu, D. Simchi-Levi, and C.-P. Teo. Inventory placement in acyclic supply chain networks. *Operations Research Letters* 34:228–238, 2006.

[20] S. Minner. Dynamic programming algorithms for multi-stage safety stock optimization. *Operations Research Spektrum* 19(4):261–271, 1997.

[21] M. Parlar and D. Berkin. Future supply uncertainty in EOQ models. *Naval Research Logistics* 38(1):107–121, 1991.

[22] A. Schmitt and L. V. Snyder. Infinite-horizon models for inventory control under yield uncertainty and disruptions. *Computers and Operations Research* 39(4):850–862, 2012.

[23] K. H. Shang and J.-S. Song. Newsvendor bounds and heuristic for optimal policies in serial supply chains. *Management Science* 49(5):618–638, 2003.

[24] E. A. Silver. A simple method of determining order quantities in joint replenishments under deterministic demand. *Management Science* 22(12):1351–1361, 1976.

[25] K. F. Simpson Jr. In-process inventories. *Operations Research* 6(6):863–873, 1958.

[26] L. V. Snyder. A tight approximation for a continuous-review inventory model with supplier disruptions. *International Journal of Production Economics* 155:91–108, 2014.

[27] L. V. Snyder and Z.-J. M. Shen. *Fundamentals of Supply Chain Theory*, 2nd ed. Wiley, Hoboken, NJ, 2019.

[28] L. V. Snyder, Z. Atan, P. Peng, Y. Rong, A. Schmitt, and B. Sinsoysal. OR/MS models for disruptions in supply chains: A review. *IIE Transactions* 48(2):89–109, 2016.

[29] B. T. Tomlin. On the value of mitigation and contingency strategies for managing supply chain disruption risks (unabridged version). Working paper, Kenan-Flagler Business School, University of North Carolina, Chapel Hill, NC, 2005.

[30] A. F. Veinott Jr. On the optimality of (*s, S*) inventory policies: New conditions and a new proof. *SIAM Journal on Applied Mathematics* 14(5):1067–1083, 1966.

[31] H. M. Wagner and T. M. Whitin. Dynamic version of the economic lot size model. *Management Science* 5(1):89–96, 1958.

[32] T. M. Whitin. *The Theory of Inventory Management.* Princeton University Press, Princeton, NJ, 1953.

[33] C. A. Yano and H. L. Lee. Lot sizing with random yield: A review. *Operations Research* 43(2):311–334, 1995.

[34] Y.-S. Zheng. On properties of stochastic inventory systems. *Management Science* 38(1): 87–103, 1992.

[35] Y.-S. Zheng and A. Federgruen. Finding optimal (*s, S*) policies is about as simple as evaluating a single policy. *Operations Research* 39(4):654–665, 1991.

[36] Y.-S. Zheng and A. Federgruen. Errata: Finding optimal (*s, S*) policies is about as simple as evaluating a single policy. *Operations Research* 40(1):192 1992.

[37] P. H. Zipkin. *Foundations of Inventory Management.* McGraw-Hill, New York, 2000.