SoftwareX 35 (2026) 102874

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

s2mflow: A meta-generator for multicommodity flow instances

Felix P. Broesamle*, Stefan Nickel

Discrete Optimization and Logistics, Institute for Operations Research, Karlsruhe Institute of Technology, Kaiserstraße 89, Karlsruhe, 76133, Germany

ARTICLE INFO

Keywords:
Network flows
Benchmarking
Instance generation
Multicommodity flows

ABSTRACT

Benchmark instances for multicommodity flow problems often fail to capture the structural characteristics of real-world networks or preserve a rigorous relationship with their single-commodity counterparts. To address these limitations, this paper presents `s2mflow`: an open-source, cross-platform Python library for lifting single-commodity minimum-cost flow instances into the multicommodity space. Unlike existing generators, `s2mflow` makes this structural relationship an explicit design principle of the instance construction process, enabling the generation of structurally grounded benchmarks. The package combines a Python interface with a high-performance Rust backend built with PyO3 and maturin, and is distributed via PyPI for Linux and macOS on x86_64 and ARM64, and Windows on x64. It provides integer partitioning methods to control commodity-demand heterogeneity, a key factor for the Single-Multi-Commodity Gap and solver performance. Additionally, `s2mflow` introduces the specialized `.mcfmin` file format for compact, reproducible storage. The resulting instances are suitable for benchmarking methods in linear programming, network optimization, and transportation science.

Metadata

Code metadata.

Nr.	Code metadata description	Metadata
C1	Current code version	v0.1.21
C2	Permanent GitHub link to code/repository used for this code version	https://github.com/FelixBroesamle/s2mflow
C3	Legal Code License	MIT License
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	Rust, Python, PyO3, Maturin, Poetry, Cargo
C6	Compilation requirements, operating environments & dependencies	Python $\geq$ 3.13, Rust $\geq$ 1.85
C7	If available Link to developer documentation/manual	https://s2mflow.readthedocs.io/en/latest/
C8	Support email for questions	felix.broesamle@kit.edu

1. Motivation and significance

A fundamental challenge in network optimization is the transition from single-commodity models to multicommodity systems. Real-world networks, from telecommunications to global logistics chains, rarely carry a single, uniform traffic type; instead, they must simultaneously accommodate multiple distinct commodities that share a common infrastructure. At the core of these applications lies the Minimum-Cost Multicommodity Flow (MCMCF) problem, which provides the mathematical foundation for telecommunication networks, global logistics chains, and complex urban traffic problems [1]. Because distinct commodities interact by competing for finite, shared arc capacities, MCMCF becomes a large-scale linear programming challenge that motivates advanced solution methods such as Dantzig-Wolfe decomposition [2], specialized bundle methods [3], interior-point algorithms [4], and GPU-based interior-point algorithms [5].

Rigorous evaluation of these algorithmic advances requires high-fidelity test instances that, ideally, maintain a clear, well-defined relationship with their single-commodity counterparts. In the single-commodity Minimum-Cost Flow (MCF) domain, the community benefits from established generators such as NETGEN [6] and standardized formats like the DIMACS [7] `.min` specification, which ensure reproducible and universally exchangeable benchmarks. In contrast, the multicommodity benchmarking ecosystem is less mature and often lacks a principled mapping from single- to multicommodity instances. This gap makes it difficult to systematically analyze how an algorithm scales

* Corresponding author.
Email addresses: felix.broesamle@kit.edu (F.P. Broesamle), stefan.nickel@kit.edu (S. Nickel).

https://doi.org/10.1016/j.softx.2026.102874
Received 26 May 2026; Received in revised form 22 June 2026; Accepted 6 July 2026
Available online 11 July 2026

when moving from a single-commodity instance to a multicommodity instance, and to study the impact of structural and demand-related features on solver performance [8]. For MCMCF, researchers have traditionally relied on real-world problems, such as the Patient Distribution System (PDS) instances, the widely utilized `MNETGEN` generator [9] and its refined variant [3], or on a meta-generation approach based on stochastic scaling [3,10]. For a comprehensive overview of MCMCF instances and generators we refer to Frangioni [11].

To address this need, we developed `s2mflow`, a formal meta-generation framework that lifts a valid MCF instance $\mathcal{P}_S(V, E, u, w, b)$ into a MCMCF instance $\mathcal{P}_M(V, E, K, u, \{u^k\}_{k\in K}, \{w^k\}_{k\in K}, \{b^k\}_{k\in K})$ while preserving key structural and integrality properties formulated by Broesamle and Nickel [8]. Here, $G = (V, E)$ represents the directed network topology defined by node set V and arc set E. The single-commodity instance is given by the arc capacity vector $u \in \mathbb{Z}_+^{|E|}$, the cost vector $w \in \mathbb{R}^{|E|}$, and the nodal demand vector $b \in \mathbb{Z}^{|V|}$. The resulting MCMCF instances preserve the underlying network topology $G = (V, E)$ along with the shared capacity u. For each commodity $k \in K$, the commodity capacity vector is u^k, the commodity cost vector is w^k, and the commodity demand vector is b^k satisfying $\sum_{i\in V} b_i^k = 0$. The generator enforces four constraints for every generated MCMCF instance:

1. *Role preservation:* A supply node ($b_i > 0$) cannot become a demand node ($b_i^k \geq 0$ for all $k \in K$), a demand node ($b_i < 0$) cannot become a supply node ($b_i^k \leq 0$, for all $k \in K$) and transshipment nodes ($b_i = 0$) stay transshipment nodes ($b_i^k = 0$ for all $k \in K$).
2. *Integrality:* Integral single-commodity demands strictly yield integral multicommodity demand vectors.
3. *Local Balance:* $\sum_{k\in K} b_i^k = b_i$ for each $i \in V$.
4. *Global Balance:* $\sum_{i\in V} b_i^k = 0$ for each $k \in K$, enforcing commodity-wise flow conservation across the network.

By implementing two distinct local partitioning schemes (*Uniform* and *Spread*) [8], `s2mflow` provides researchers control over the degree of commodity-demand heterogeneity. The local partitioning methods generate demand values b_i^k for each commodity k based on the original single-commodity nodal value b_i. The *Uniform* method ensures that the components are distributed as evenly as possible across all commodities to simulate symmetric traffic. Conversely, the *Spread* method utilizes a randomized allocation scheme designed to maximize the variance between commodities, creating highly asymmetric demand profiles [8]. These local partitioning methods inherently satisfy the first three properties (role preservation, integrality, and local balance). To satisfy the fourth property, a global balancing algorithm is subsequently invoked if the global balance constraint is violated; this global balancing phase is mathematically designed to adjust flows across the network without violating any of the properties previously established during the local phase [8]. The correctness of the algorithms, as well as their runtime behavior, is formally proven in [8].

Commodity-demand heterogeneity is a primary driver of the Single-Multi-Commodity Gap [8]. The work by Broesamle and Nickel [8] demonstrated that distinct linear programming solution methods, specifically Interior-Point Methods (IPM) and dual simplex algorithms, exhibit different runtime behaviors relative to the commodity-demand heterogeneity. By allowing users to control commodity-demand heterogeneity via the *Uniform* and *Spread* methods, `s2mflow` provides a reproducible meta-generation framework to systematically test and evaluate solution methods under isolated structural variance.

To integrate this framework naturally into the modern optimization ecosystem, `s2mflow` is designed as a Python package built on a Rust backend using PyO3 and `maturin`. The library exposes a simple, idiomatic Python API, while the underlying computations are handled by a memory-safe, high-performance Rust backend. It is distributed via PyPI as prebuilt wheels for Linux and macOS on x86_64 and ARM64, and for Windows on x64, enabling straightforward deployment in standard Python environments without local compilation. Finally, `s2mflow` introduces the `.mcfmin` file format for compact, reproducible storage and exchange of benchmark instances. Together, these design choices position `s2mflow` as a bridge between classical single-commodity MCF generators and instances and the modern optimization landscape, supporting evaluation and benchmarking of MCMCF solution methods.

2. Software description

The `s2mflow` library is designed to bridge the gap between single-commodity instances and multicommodity instances. To achieve this, the software is built around three core principles: mathematically rigorous structural transformation, efficient execution for large-scale network graphs, and seamless integration into the modern Python-based mathematical optimization, operations research, and data science ecosystem.

2.1. Software architecture

The architecture of `s2mflow` follows a *separation of concerns* paradigm, implemented as a hybrid technology stack that combines a Python front end with a compiled Rust back end. The Python layer provides user-facing functionality and integration with common Python workflows, while Rust implements the graph-processing and partitioning algorithms that benefit from low-level performance and memory safety. The architecture consists of three primary layers:

The Python interface (front end): A lightweight, idiomatic API that allows users to configure lifting parameters, specify the number of commodities K, select partitioning methods (*Uniform*, *Spread*), and export the resulting instances. It serves as the orchestration layer, enabling straightforward integration with standard Python pipelines.

The binding layer (PyO3 and `maturin`): This layer exposes Rust code to Python by generating native extension modules using PyO3. The `maturin` build tool invokes the Rust compiler and produces wheel distributions that contain compiled shared libraries, so users do not need to build the Rust extension manually. By utilizing this workflow, `s2mflow` is distributed on PyPI as prebuilt wheels for Linux and macOS on x86_64 and ARM64, and for Windows on x64.

The Rust core (back end): The computational core of the library, responsible for graph parsing and execution of the partitioning and balancing algorithms. To ensure reproducible benchmark generation across platforms, the Rust core uses ordered data structures (e.g., `BTreeMap`) along with explicitly seeded pseudo-random number generators where randomness is required.

2.2. Software functionalities

The `s2mflow` library provides a suite of core functionalities that gives researchers control over the meta-generation of MCMCF instances, while preserving the structural invariants defined in Section 1:

Network parsing: The library includes an optimized parser that efficiently ingests standard DIMACS `.min` files and loads them into memory as basic single-commodity MCF instances.

Uniform partitioning: A deterministic local partitioning method that distributes single-commodity nodal demands as evenly as mathematically possible across all commodities $k \in K$. This enables the generation of symmetric multicommodity demands while strictly preserving integrality and local balance [8].

Spread partitioning: A stochastic local partitioning method designed to yield high variance in nodal demands across commodities. This functionality creates highly asymmetric commodity demands to evaluate algorithmic scaling under extreme commodity-demand heterogeneity [8].

Global balancing: When local demand partitioning induces a violation of global flow conservation for an individual commodity, the library executes a network-wide correction routine. This global balancing mechanism restores $\sum_{i\in V} b_i^k = 0$ for each commodity $k \in K$ while mathematically ensuring that the pre-established role preservation, integrality, and local balance constraints are never violated [8].

Standardized format: To facilitate reproducible benchmark exchange, `s2mflow` introduces the specialized `.mcfmin` file format specification.

This produces compact and human-readable representations of MCMCF instances.

MCMCF parsing: Complementing the export pipeline, the library includes a dedicated `.mcfmin` parser that reconstructs multicommodity instances.

Network utilities: To streamline the integration of generated instances into algebraic modeling languages, the library provides an adjacency-mapping routine that returns the incoming and outgoing neighbors of each node.

2.3. Sample code snippets analysis

A core design objective of `s2mflow` is to make sophisticated instance lifting straightforward, and well integrated within optimization workflows. Listing 1 illustrates an end-to-end workflow: parsing a standard single-commodity DIMACS network file, lifting it into a 3-commodity space with high commodity-demand heterogeneity (Spread), and exporting the output into the custom `.mcfmin` specification.

```python
import s2mflow

# 1. Load a single-commodity network (DIMACS .min format)
# Input file example contents:
# p min 2 1   (2 nodes, 1 arc)
# n 1 10      (Node 1: supply of 10)
# n 2 -10     (Node 2: demand of 10)
# a 1 2 0 10 9  (Arc from 1 to 2, min_cap=0, max_cap=10, cost=9)

network = s2mflow.load_min_instance("input.min")

# 2. Generate multicommodity data for 3 commodities
# is_uniform=False activates the stochastic 'Spread' method
mc_data = s2mflow.generate_multi_commodity_data(
        instance=network,
        num_commodities=3,
        is_uniform=False,
        randomize_caps=False,
        randomize_costs=False,
        seed=512,
)

# 3. Save as a multi-commodity instance (.mcfmin format)
s2mflow.save_multi_commodity_instance("output.mcfmin", network,
    mc_data)

# Output file (.mcfmin format)
# p min 2 1 3 0 0 512 (3 commodities, randomize_caps = False (0)
    randomize_costs = False (0) seed = 512)
# n 1 10 2 5 3 (Supply of 10 split into supplies: 2, 5, 3)
# n 2 -10 -2 -5 -3 (Demand of -10 split into demands: -2, -5, -3)
# a 1 2 0 10 10 9

# 4. Load multicommodity instance
loaded_mc_data = s2mflow.load_multi_commodity_instance("output.mcfmin"
    )

# 5. Direct usage bypassing formal file formats
data = {1: 126, 2: -126}

spread_multi_data = s2mflow.split_supplies_spread(
        data,
        num_commodities=5,
        seed=512,
)
# spread_multi_data = {1: [6, 24, 23, 12, 61], 2: [-6, -24, -23, -12,
    -61]}

uniform_multi_data = s2mflow.split_supplies_uniform(
        data,
        num_commodities=5,
)
# uniform_multi_data = {1: [25, 26, 25, 25, 25], 2: [-25, -26, -25,
    -25, -25]}
```

Listing 1. s2mflow end-to-end workflow.

The `split_supplies_spread` and `split_supplies_uniform` functions expose the core partitioning logic independently of file formats, enabling direct integration with the integer partitioning methods.

The example above focuses primarily on lifting single-commodity instances into the multicommodity space, with particular emphasis on the partition of nodal demands and supplies so that the four core properties introduced in Section 1 are preserved. This design choice aligns closely with the formal definition of the Single–Multi–Commodity Gap [8], which considers commodity-demand heterogeneity rather than changes to capacities or costs. Nonetheless, `s2mflow` also supports explicit generation of commodity-specific capacities and cost vectors when desired; these are drawn as integer values controlled by the `randomize_caps` and `randomize_costs` flags, and their use is illustrated in the examples discussed in the following section.

3. Illustrative examples

This section illustrates how `s2mflow` lifts a single-commodity minimum-cost flow instance into a multicommodity instance, preserving the four core invariants (role preservation, integrality, local balance, and global balance) while controlling commodity-demand heterogeneity and, optionally, randomized commodity-specific capacities and costs. The base network is defined in Listing 2 in the standard DIMACS `.min` format. Additionally, this section provides complete end-to-end optimization workflows and a benchmarking analysis evaluating the computational scaling of `s2mflow`.

```
# c Base single-commodity instance (input.min)
# p min 4 5
# n 1 17
# n 4 -17
# a 1 2 0 10 10
# a 1 3 0 15 5
# a 2 4 0 10 10
# a 3 2 0 5 20
# a 3 4 0 15 4
```

Listing 2. Base single-commodity instance.

We consider five variants: *Uniform* and *Spread* partitioning without commodity-specific capacities and costs, and *Spread* partitioning with randomized commodity-specific capacities, costs, and both simultaneously. The complete generation script is included in the repository (`examples/demo.py`). For the first case (*Uniform* partitioning), we demonstrate the complete pipeline. For subsequent variants, we display only the generation function call and the resulting `.mcfmin` output to emphasize the structural format changes in compact form.

3.1. The extended `.mcfmin` format

The library uses a natural extension of the DIMACS `.min` format to support multiple commodities:

- *Problem Line:* `p min <num_nodes> <num_edges> <num_commodities> <randomize_caps> <randomize_costs> <is_uniform>` ⟨*seed*⟩
  - `seed`: Relevant if `is_uniform = 0` (Spread method) or if capacity or cost randomization is enabled.
- *Node Line:* `n <node_id> <total_demand> <demand_com_1> … <demand_com_K>`
- *Arc Line:* Dynamically shifts based on the randomization flags (`randomize_caps, randomize_costs`):
  - Default `(0, 0)`: `a` ⟨*from*⟩ ⟨*to*⟩ ⟨*low*⟩ `<cap_total>` `<cap_total>` ⟨*cost*⟩
  - Commodity-specific capacities `(1, 0)`: `a` ⟨*from*⟩ ⟨*to*⟩ ⟨*low*⟩ `<cap_total> <cap_1> … <cap_K>` ⟨*cost*⟩
  - Commodity-specific costs `(0, 1)`: `a` ⟨*from*⟩ ⟨*to*⟩ ⟨*low*⟩ `<cap_total> <cap_total> <cost_1> … <cost_K>`

– Commodity-specific capacities and costs `(1, 1)`: `a` ⟨*from*⟩ ⟨*to*⟩ ⟨*low*⟩ `<cap_total> <cap_1> … <cap_K> <cost_1> … <cost_K>`

3.2. Uniform partitioning

The *Uniform* partitioning variant in Listing 3 distributes nodal demands as evenly as possible across the 4 commodities.

```python
import s2mflow

NUM_COMMODITIES = 4

# 1. Load the baseline single-commodity network
network = s2mflow.load_min_instance("input.min")

# 2. Generate multi-commodity data via Uniform partitioning
uniform_mc_data = s2mflow.generate_multi_commodity_data(
        instance=network,
        num_commodities=NUM_COMMODITIES,
        is_uniform=True,
        randomize_caps=False,
        randomize_costs=False,
)

# 3. Save as a multi-commodity instance (.mcfmin format)
s2mflow.save_multi_commodity_instance("uniform.mcfmin", network,
    uniform_mc_data)

# 4. Reload functionality (Round-trip validation)
loaded_data = s2mflow.load_multi_commodity_instance("uniform.mcfmin")

# --- Output file contents (uniform.mcfmin) ---
# c Multicommodity flow generated by s2mflow
# p min 4 5 4 0 0 1 0
# n 1 17 5 4 4 4
# n 4 -17 -5 -4 -4 -4
# a 1 2 0 10 10 10
# a 1 3 0 15 15 5
# a 2 4 0 10 10 10
# a 3 2 0 5 5 20
# a 3 4 0 15 15 4
```

Listing 3. Uniform partitioning.

3.3. Spread partitioning

The *Spread* partitioning variant in Listing 4 distributes demand unevenly across commodities.

```python
# 2. Spread partitioning
spread_mc_data = s2mflow.generate_multi_commodity_data(
        instance=network,
        num_commodities=NUM_COMMODITIES,
        is_uniform=False,
        randomize_caps=False,
        randomize_costs=False,
        seed=42,
)

# --- Output file ---
# c Multicommodity flow generated by s2mflow
# p min 4 5 4 0 0 0 42
# n 1 17 4 6 1 6
# n 4 -17 -4 -6 -1 -6
# p min 4 5 4 1 1 0 42
# n 1 17 4 6 1 6
# n 4 -17 -4 -6 -1 -6
# a 1 2 0 10 9 8 8 10 13 15 6 17
# a 1 3 0 15 10 13 10 11 4 10 6 7
# a 2 4 0 10 9 9 8 7 6 6 18 13
# a 3 2 0 5 4 4 5 5 21 34 23 23
# a 3 4 0 15 15 11 12 12 8 6 6 3
```

Listing 4. Spread partitioning.

3.4. Spread partitioning with randomized capacities

In Listing 5, commodity-specific capacities are additionally sampled.

```python
# 2. Spread partitioning, randomized capacities
spread_rand_caps_mc_data = s2mflow.generate_multi_commodity_data(
        instance=network,
        num_commodities=NUM_COMMODITIES,
        is_uniform=False,
        randomize_caps=True,
        cap_a=0.6,
        cap_b=1.0,
        randomize_costs=False,
        seed=42,
)

# --- Output file contents ---
# c Multicommodity flow generated by s2mflow
# p min 4 5 4 1 0 0 42
# n 1 17 4 6 1 6
# n 4 -17 -4 -6 -1 -6
# a 1 2 0 10 9 9 9 8 10
# a 1 3 0 15 10 12 14 15 5
# a 2 4 0 10 7 7 10 9 10
# a 3 2 0 5 4 4 5 4 20
# a 3 4 0 15 10 13 10 13 4
```

Listing 5. Spread partitioning, randomized capacities.

3.5. Spread partitioning with randomized costs

In Listing 6, commodity-specific costs are additionally sampled.

```python
# 2. Spread partitioning, randomized costs
spread_rand_costs_mc_data = s2mflow.generate_multi_commodity_data(
        instance=network,
        num_commodities=NUM_COMMODITIES,
        is_uniform=False,
        randomize_caps=False,
        randomize_costs=True,
        cost_a=0.5,
        cost_b=2.0,
        seed=42,
)

# --- Output file contents ---
# c Multicommodity flow generated by s2mflow
# p min 4 5 4 0 1 0 42
# n 1 17 4 6 1 6
# n 4 -17 -4 -6 -1 -6
# a 1 2 0 10 10 13 14 15 12
# a 1 3 0 15 15 3 6 9 9
# a 2 4 0 10 10 7 6 19 13
# a 3 2 0 5 5 22 15 26 17
# a 3 4 0 15 15 3 6 3 6
```

Listing 6. Spread partitioning, randomized costs.

3.6. Spread partitioning with randomized capacities and costs

In Listing 7, commodity-specific capacities and costs are additionally sampled.

```
# 2. Spread partitioning, randomized capacities and costs
spread_rand_caps_costs_mc_data = s2mflow.generate_multi_commodity_data
    (
        instance=network,
        num_commodities=NUM_COMMODITIES,
        is_uniform=False,
        randomize_caps=True,
        cap_a=0.6,
        cap_b=1.0,
        randomize_costs=True,
        cost_a=0.5,
        cost_b=2.0,
        seed=42,
)

# --- Output file contents ---
# c Multicommodity flow generated by s2mflow

# p min 4 5 4 1 1 0 42
# n 1 17 4 6 1 6
# n 4 -17 -4 -6 -1 -6
# a 1 2 0 10 9 8 8 10 13 15 6 17
# a 1 3 0 15 10 13 10 11 4 10 6 7
# a 2 4 0 10 9 9 8 7 6 6 18 13
# a 3 2 0 5 4 4 5 5 21 34 23 23
# a 3 4 0 15 15 11 12 12 8 6 6 3
```

Listing 7. Spread partitioning, randomized capacities and costs.

3.7. Integration into optimization workflows

To demonstrate the practical utility of `s2mflow` within modern optimization pipelines, this section illustrates the direct ingestion of multicommodity data into solver environments. By leveraging the package's native data structures, researchers can bypass manual array formatting and seamlessly construct mathematical programming models using standard algebraic modeling languages.

Listing 8 demonstrates this integration using Pyomo [12], a widespread open-source algebraic modeling language embedded in Python, paired with the HiGHS solver [13] using data parsed from a `.mcfmin` file. Conversely, Listing 9 illustrates a pathway utilizing the `gurobipy` interface for the commercial solver Gurobi [14], bypassing file I/O entirely by generating the multicommodity instance directly in-memory.

3.8. Computational scaling and performance

To demonstrate the efficiency of the Rust backend, we benchmarked the `NETGEN-SR` instance family provided by the LEMON graph library project [15]. The evaluation was executed using Python 3.13.5 and Rustc 1.93.0 on a machine equipped with an Intel Core Ultra 7 255U processor and 32 GB of RAM.

The structural characteristics of the evaluated base topologies are summarized in Table 1. Each instance is parameterized by its number of nodes, arcs, supply nodes, demand nodes, total commodity demand, and the execution time required to parse the initial network (`Load Min`) in seconds.

Table 2 tracks the subsequent performance scaling across an increasing number of commodities (K). We profile three distinct execution phases (measured in seconds): multicommodity parameter generation (Gen), writing to the `.mcfmin` file format (Write), and reloading the compiled instance (Load). We compare two generation methods: `std` (spread partitioning) and `rand` (spread partitioning with randomized commodity capacities and arc costs). The empirical results highlight

```
import pyomo.environ as pyo
import s2mflow

# 1. Load multi-commodity benchmark instance
mc = s2mflow.load_multi_commodity_instance("input.mcfmin")

# 2. Initialize Model
model = pyo.ConcreteModel("MCMCF")

# 3. Decision Variables: 0 <= x_e^k <= u_e^k
def commodity_edge_bounds(m, k, u, v):
        upper_bound = mc.commodity_capacities[(u, v)][k]
        return (0.0, float(upper_bound))

model.flow = pyo.Var(
        mc.commodity_edges,
        domain=pyo.NonNegativeReals,
        bounds=commodity_edge_bounds,
        name="x"
)

# 4. Objective: Minimize total routing cost
model.obj = pyo.Objective(
        expr=sum(
                model.flow[k, u, v] * mc.commodity_weights[(u, v)][k]
                for (k, u, v) in mc.commodity_edges
        ),
        sense=pyo.minimize
)

# 5. Shared Mutual Capacity Constraints
model.shared_caps = pyo.ConstraintList()
for i, (u, v) in enumerate(mc.edges):
        model.shared_caps.add(
                sum(
                        model.flow[k, u, v]
                        for k in range(mc.num_commodities)
                ) <= mc.capacities[i]
)

# 6. Flow Conservation Constraints
model.flow_balance = pyo.ConstraintList()
incoming, outgoing = s2mflow.get_adjacency_mapping(mc.nodes, mc.edges)

for k in range(mc.num_commodities):
        for node in mc.nodes:
                in_flow = sum(model.flow[k, u, node] for u in incoming.
                    get(node, []))
                out_flow = sum(model.flow[k, node, v] for v in outgoing
                    .get(node, []))

                demand = mc.commodity_supply_demand_data[node][k] if
                    node in mc.commodity_supply_demand_data else 0.0
                model.flow_balance.add(out_flow - in_flow == demand)

# 7. Solve
solver = pyo.SolverFactory("highs")
results = solver.solve(model, tee=True)
print(f"Optimal Objective Value: {pyo.value(model.obj)}")
```

Listing 8. Open-source integration (Pyomo & HiGHS).

sustained efficiency across all problem scales. For small networks such as `netgen_sr_08a`, the full generation and serialization sequence completes in under 60 milliseconds. In the largest randomized variant (`rand`, ID 6, $K = 50$), the compiled Rust core synthesizes more than 104 million independent parameters. The multicommodity data generation phase remains performant, requiring only an additional 3.74 s relative to `std`. The total pipeline execution time increases primarily due to exporting and reloading the expanded (full) `.mcfmin` file.

We remark that these problem sizes are significantly larger than typical instances found in current research. For example, in [5], instances were considered with up to 10,000 nodes, 120,000 arcs, and 10 commodities, while in [8], instances reached up to 800 nodes, 24,000 arcs, and 50 commodities.

```python
import gurobipy as grb
import s2mflow

# 1. Load baseline and generate data in-memory
net = s2mflow.load_min_instance("input.min")
mc_data = s2mflow.generate_multi_commodity_data(net, num_commodities
    =3, is_uniform=False, seed=512)

# 2. Initialize Gurobi Model
model = grb.Model("MCMCF")

# 3. Decision Variables
upper_bounds = [mc_data.commodity_capacities[(u, v)][k] for (k, u, v)
    in mc_data.commodity_edges]
flow = model.addVars(
       mc_data.commodity_edges,
       lb=0.0,
       ub=upper_bounds,
       vtype=grb.GRB.CONTINUOUS,
       name="x"
)

# 4. Objective Function
model.setObjective(
       grb.quicksum(flow[k, u, v] * mc_data.commodity_weights[(u, v)
           ][k]
       for (k, u, v) in mc_data.commodity_edges
       ),
       sense=grb.GRB.MINIMIZE
)

# 5. Shared Mutual Capacity Constraints
model.addConstrs(
       (
              grb.quicksum(
                     flow[k, u, v]
                     for k in range(mc_data.num_commodities)
              ) <= net.capacities[i]
              for i, (u, v) in enumerate(net.arcs)
       ),
       name="Shared_Cap"
)

# 6. Flow Conservation Constraints
incoming, outgoing = s2mflow.get_adjacency_mapping(net.nodes, net.arcs
    )

for k in range(mc_data.num_commodities):
       for node in net.nodes:
              in_flow = grb.quicksum(flow[k, u, node] for u in
                  incoming.get(node, []))
              out_flow = grb.quicksum(flow[k, node, v] for v in
                  outgoing.get(node, []))

              demand = mc_data.supply_partition[node][k] if node in
                  mc_data.supply_partition else 0.0
              model.addConstr(
                     out_flow - in_flow == demand,
                     name=f"balance_{k}_{node}"
              )

# 7. Solve
model.optimize()
print(f"Optimal Objective Value: {model.ObjVal}")
```

Listing 9. Gurobi and `gurobipy`, in-memory generation.

Table 1
Base topology Configurations for NETGEN-SR Instances.

Instance	Nodes	Arcs	Srcs	Dmds	Tot Demand	Load Min
1: netgen_sr_08a	256	4096	16	16	16,000	0.0014
2: netgen_sr_10a	1024	32,768	32	32	32,000	0.0079
3: netgen_sr_11a	2048	92,682	45	45	45,000	0.0288
4: netgen_sr_12a	4096	262,144	64	64	64,000	0.0717
5: netgen_sr_13a	8192	741,455	91	91	91,000	0.1984
6: netgen_sr_14a	16,384	2,097,152	128	128	128,000	0.8777

Table 2
Benchmark results comparing std/rand generation (times in seconds).

Inst.	K	Gen (std/rand)	Write (std/rand)	Load (std/rand)
1:	5	0.0030/0.0031	0.0019/0.0049	0.0158/0.0190
	10	0.0039/0.0048	0.0049/0.0050	0.0168/0.0135
	20	0.0061/0.0068	0.0022/0.0096	0.0111/0.0173
	50	0.0127/0.0142	0.0026/0.0217	0.0181/0.0216
2:	5	0.0277/0.0349	0.0132/0.0239	0.0406/0.0390
	10	0.0487/0.0504	0.0190/0.0407	0.0389/0.0465
	20	0.0861/0.0759	0.0164/0.0733	0.0365/0.0639
	50	0.1936/0.1788	0.0164/0.1651	0.0477/0.1051
3:	5	0.0872/0.0891	0.0390/0.0639	0.0755/0.0892
	10	0.1392/0.1526	0.0433/0.1123	0.0727/0.1201
	20	0.2487/0.2565	0.0490/0.2050	0.0818/0.1656
	50	0.5252/0.5722	0.0486/0.4520	0.0944/0.2797
4:	5	0.2492/0.3510	0.1025/0.2484	0.2692/0.2558
	10	0.4230/0.4720	0.1147/0.3328	0.1941/0.4041
	20	0.7596/0.7833	0.1311/0.5724	0.2219/0.4668
	50	1.6150/1.6995	0.1344/1.3241	0.2645/0.7686
5:	5	0.7915/0.8514	0.2969/0.5204	0.4610/0.6832
	10	1.3138/1.4663	0.3430/0.8733	0.6083/1.0841
	20	2.3683/2.4099	0.3825/1.4801	0.6710/1.7515
	50	4.9060/5.2603	0.4100/3.3010	0.8825/2.5859
6:	5	4.2976/4.4220	0.8563/1.5197	1.6861/2.8761
	10	7.1258/8.8457	0.9693/2.4662	3.6271/4.8378
	20	12.5907/12.2959	1.1096/4.6455	2.9332/5.2400
	50	20.8690/24.6117	1.1872/10.1748	2.7607/7.7355

4. Impact

Research in MCMCF solution methods remains highly active [5,8]. `s2mflow` provides a high-performance Python library with a Rust backend for the meta-generation of test instances based on [8].

4.1. New research questions enabled

By isolating commodity-demand heterogeneity while preserving the four core invariants (role preservation, integrality, local balance, and global balance), `s2mflow` enables new research questions:

- *Runtime Scaling:* How does commodity-demand heterogeneity affect the runtime of solution methods?
- *Numerical Stability:* How does commodity-demand heterogeneity affect the numerical stability of solution methods?
- *Solution Quality:* How does commodity-demand heterogeneity affect the solution quality?

Without a mathematically verified and reproducible generator that strictly enforces these structural properties while scaling efficiently to

large-scale instances (as evaluated in Section 3.8), researching these questions systematically is difficult.

4.2. Improving benchmarking practices

Rigorous solver benchmarking requires large and structurally consistent datasets. While the scale of mathematical programming research is immense, evidenced by over 23,000 publications since 2020 on Google Scholar referencing the commercial solver Gurobi [14] and approximately 143,000 referencing interior point linear methods, benchmark instance generation often remains ad-hoc.

The `s2mflow` framework standardizes this generation pipeline for multicommodity problems. Central to this standardization is the specialized `.mcfmin` file format introduced in Section 3.1, which directly addresses modern benchmarking bottlenecks through three primary design advantages. First, the format maintains strict human-readability, thereby simplifying manual verification and debugging routines. Second, it is storage-efficient, as it incorporates non-randomized capacities and costs without saving commodity-wise arc data explicitly. Finally, the format and underlying framework are extensible, enabling users to incorporate supplementary network data attributes seamlessly. When combined with native `s2mflow` data structures, this format interfaces directly with algebraic modeling languages, as demonstrated in Section 3.7.

4.3. Daily practice: workflow integration

`s2mflow` simplifies daily research practice by embedding complex lifting and balancing operations into a concise Python API. Rather than acting solely as a standalone file generator, the native Rust core efficiently parses DIMACS `.min` and `s2mflow`-native `.mcfmin` files, mapping them directly into in-memory data structures. The PyO3 bindings further facilitate seamless integration into existing Python-based optimization pipelines. These data structures are designed to interface directly with algebraic modeling languages and modern optimization solvers. For example, researchers can inject generated instances into Pyomo (e.g., using the HiGHS solver) or optimization solver APIs like `gurobipy`, as detailed in Section 3.7. This automates the end-to-end pipeline from reproducible generation to direct solver evaluation.

4.4. Widespread use and applications

Multicommodity flow models serve as the computational backbone for massive logistics, transportation, and telecommunication networks. Recent publication volumes illustrate the broad applicability of the MCMCF (Google Scholar under the following terms since 2020):

- *Network Flow*: ≈ 61,200 publications.
- *Multicommodity*: ≈ 15,400 publications.
- *Traffic Flow Optimization*: ≈ 94,500 publications.
- *Supply Chain Optimization*: ≈ 60,100 publications.
- *Telecommunications Networks*: ≈ 28,100 publications.

By providing a standardized, reproducible framework to generate structurally grounded benchmarks, `s2mflow` supports methodological evaluation and solution method benchmarking across these industries.

5. Conclusions

This paper introduces `s2mflow`, an open-source Python library for lifting single-commodity minimum-cost flow instances into the multicommodity space while preserving key structural properties. Implemented as a Python package with a high-performance Rust backend and built using PyO3 and `maturin`, the library integrates naturally into modern Python workflows and is distributed via prebuilt PyPI wheels for Linux, macOS, and Windows. The resulting benchmarks are relevant for diverse domains such as linear programming, network optimization, logistics, transportation science, and telecommunications. The `.mcfmin` file format enables compact, reproducible storage and exchange of MCMCF instances. By providing a meta-generation framework with control over commodity-demand heterogeneity, `s2mflow` enables systematic studies of the Single-Multi-Commodity Gap and its effect on solver behavior.

CRediT authorship contribution statement

Felix P. Broesamle: Writing – review & editing, Writing – original draft, Validation, Software, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Stefan Nickel:** Writing – review & editing, Supervision, Project administration, Funding acquisition, Conceptualization.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used Gemini (Google, Gemini 3) to improve the readability and language of the manuscript. After using these services, the authors carefully reviewed, revised, and edited the content as needed and take full responsibility for the content of the published article.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was supported by the Bundesministerium für Forschung, Technologie und Raumfahrt (BMFTR, German Federal Ministry of Research, Technology and Space) [QuSol: Quantum Optimization Solver Kit].

References

[1] Ahuja RK, Magnanti TL, Orlin JB. Network flows: theory, algorithms, and applications. Upper Saddle River, NJ: Prentice Hall; 1993.
[2] Dantzig GB, Wolfe P. Decomposition principle for linear programs. Oper Res 1960;8(1):101–11. http://www.jstor.org/stable/167547.
[3] Frangioni A, Gallo G. A bundle type dual-ascent approach to linear multicommodity min-cost flow problems. INFORMS J Comput 1999;11(4):370–93. https://doi.org/10.1287/ijoc.11.4.370
[4] Castro J. A specialized interior-point algorithm for multicommodity network flows. SIAM J Optim 2000;10(3):852–77. https://doi.org/10.1137/S1052623498341879
[5] Zhang F, Boyd S. Solving large multicommodity network flow problems on GPUs. arXiv:2501.17996. 2025.
[6] Klingman D, Napier A, Stutz J. NETGEN: a program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. Manag Sci 1974;20(5):814–21. https://doi.org/10.1287/mnsc.20.5.814
[7] DIMACS. DIMACS: center for discrete mathematics and theoretical computer science; 2026 http://dimacs.rutgers.edu [accessed 21 June 2026].
[8] Broesamle FP, Nickel S. On the single-multi-commodity gap: lifting single- to multicommodity flow instances, Optimization Online Preprint; 2026 https://optimization-online.org/?p=34287 [accessed 21 June 2026].
[9] Ali A, Kennington JL. MNETGEN program documentation. Technical Report 77003 Dallas, TX: Department of Industrial Engineering and Operations Research, Southern Methodist University; 1977.
[10] Castro J, Nabona N. Computational tests of a linear multicommodity network flow code with linear side constraints through primal partitioning; 2004. https://api.semanticscholar.org/CorpusID:7023201.
[11] Frangioni A. CommaLAB: datasets, multicommodity flow problems; 2026 https://commalab.di.unipi.it/datasets/mmcf/ [accessed 21 June 2026].
[12] Bynum ML, Hackebeil GA, Hart WE, Laird CD, Nicholson BL, Siirola JD, Watson J-P, Woodruff DL. Pyomo – optimization modeling in Python. 3 ed. Springer; 2021.
[13] Huangfu Q, Hall JAJ. Parallelizing the dual revised simplex method. Math Program Comput 2018;10(1):119–42. https://doi.org/10.1007/s12532-017-0130-5
[14] Gurobi Optimization, LLC. Gurobi optimizer reference manual; 2026 https://www.gurobi.com [accessed 21 June 2026].
[15] Kovács P. LEMON graph library; 2026 https://lemon.cs.elte.hu/trac/lemon/wiki/MinCostFlowData [accessed 21 June 2026].