Self-Sustaining Spaceship: Síntesis Óptima de Capacidad y Operación con Balances Cerrados y Compilación Determinista mediante SATX

Oscar Riveros
Polímata

2026-01-19

Abstract

Se estudia la síntesis conjunta de *capacidad instalada* y *operación diaria* de un sistema discreto de soporte vital con ciclos cerrados de recursos (O_2/CO_2, agua, comida, residuos, nutrientes) en un horizonte finito. El problema se modela como un sistema dinámico lineal en tiempo discreto con restricciones de seguridad, acoplamiento lineal entre capacidad y operación, y opcionalmente una cota sobre el presupuesto energético diario. Para habilitar operación fraccional sin introducir bilinearidad (capacidad × utilización), se introduce una cuantización Q que transforma el modelo en un programa lineal entero (ILP) puramente integral: las tasas de operación se expresan en "cuantos" enteros y los inventarios se escalan por Q. Se presenta una formulación matemática rigurosa, un esquema de optimización lexicográfica (capacidad, luego energía) implementable con *Big-M* seguro, y un protocolo de verificación pos-solución basado en sustitución exacta y chequeo de invariantes. El papel de SATX es formalizar la teoría operacional sobre dominios finitos y compilarla de manera determinista a formatos estándar (p. ej. MPS/LP), habilitando verificación cruzada con solvers externos y auditoría directa sobre el modelo exportado.

0.1 Planteamiento

Consideramos un sistema de soporte vital discretizado en días, con horizonte $K \in \mathbb{N}$. El sistema administra un conjunto finito de recursos $\mathcal{R}$ y un conjunto de módulos tecnológicos $\mathcal{M}$. Cada módulo puede ser instalado en un número entero de unidades y puede operar diariamente a una tasa fraccional. Los inventarios deben permanecer en rangos seguros (mínimos y máximos) en todo el horizonte y, opcionalmente, se impone sustentabilidad terminal (ciclo cerrado) al final del horizonte.

El objetivo es sintetizar simultáneamente: (i) *capacidad instalada* por módulo y (ii) *agenda de operación* día a día, minimizando capacidad instalada y, opcionalmente, el presupuesto energético diario requerido.

0.2 Datos, conjuntos y notación

Conjuntos

- Recursos: $\mathcal{R} = \{r\}$.
- Módulos: $\mathcal{M} = \{m\}$.
- Tiempo: $\mathcal{T} = \{0, 1, \ldots, K-1\}$ y estados $\{0, 1, \ldots, K\}$.

Parámetros de seguridad e inicialización

Para cada recurso $r \in \mathcal{R}$:

$$S_r^{\min} \in \mathbb{Z}_{\geq 0}, \qquad S_r^{\max} \in \mathbb{Z}_{\geq 0}, \qquad S_r^0 \in \mathbb{Z}_{\geq 0}, \tag{1}$$

con $S_r^{\min} \leq S_r^0 \leq S_r^{\max}$.

Tripulación

Para cada recurso $r \in \mathcal{R}$ se especifican consumos y producciones diarios (en unidades físicas):

$$c_r^{\text{crew}} \in \mathbb{Z}_{\geq 0}, \qquad p_r^{\text{crew}} \in \mathbb{Z}_{\geq 0}. \tag{2}$$

Módulos tecnológicos

Para cada módulo $m \in \mathcal{M}$:

$$\overline{C}_m \in \mathbb{Z}_{\geq 0} \quad \text{(cota superior de unidades instalables)}, \tag{3}$$
$$w_m \in \mathbb{Z}_{\geq 0} \quad \text{(costo/peso de capacidad instalada)}, \tag{4}$$
$$e_m \in \mathbb{Z}_{\geq 0} \quad \text{(energía por unidad de operación)}, \tag{5}$$
$$p_{m,r} \in \mathbb{Z}_{\geq 0} \quad \text{(producción de } r \text{ por unidad de operación completa)}, \tag{6}$$
$$c_{m,r} \in \mathbb{Z}_{\geq 0} \quad \text{(consumo de } r \text{ por unidad de operación completa)}. \tag{7}$$

0.3 Cuantización y escalamiento integral

Una dificultad estructural común es evitar bilinearidad del tipo *operación* $\leq$ *capacidad* cuando la operación admite fracciones de unidad, sin introducir variables continuas que debiliten la interpretación discreta.

Definición 1 (Cuantización Q)**.** *Sea $Q \in \mathbb{N}$, $Q \geq 1$. Definimos:*

- *Operación en cuantos enteros:* $\text{run}_{t,m}^{(Q)} \in \mathbb{Z}_{\geq 0}$.
- *Operación física asociada:* $u_{t,m} := \text{run}_{t,m}^{(Q)}/Q$ *(resolución* $1/Q$*).*
- *Inventario escalado:* $S'_{t,r} := Q\, S_{t,r}$.

El acoplamiento "operación $\leq$ capacidad" queda lineal e integral:

$$0 \leq \text{run}_{t,m}^{(Q)} \leq Q \cdot C_m, \tag{8}$$

donde C_m es la capacidad instalada entera del módulo m.

0.4 Formulación matemática (ILP)

Variables

$$C_m \in \mathbb{Z}_{\geq 0} \qquad \forall m \in \mathcal{M} \quad \text{(capacidad instalada)}, \tag{9}$$
$$\text{run}_{t,m}^{(Q)} \in \mathbb{Z}_{\geq 0} \qquad \forall t \in \mathcal{T},\ \forall m \in \mathcal{M} \quad \text{(operación en cuantos)}, \tag{10}$$
$$S'_{t,r} \in \mathbb{Z}_{\geq 0} \qquad \forall t \in \{0, \ldots, K\},\ \forall r \in \mathcal{R} \quad \text{(inventario escalado)}, \tag{11}$$
$$EB^{(Q)} \in \mathbb{Z}_{\geq 0} \qquad \text{(presupuesto energético diario escalado, opcional)}. \tag{12}$$

Restricciones de capacidad

$$0 \le C_m \le \overline{C}_m, \qquad 0 \le \mathrm{run}_{t,m}^{(Q)} \le Q \cdot \overline{C}_m, \qquad \mathrm{run}_{t,m}^{(Q)} \le Q \cdot C_m. \tag{13}$$

Rangos seguros e inicialización

$$S'_{0,r} = Q\, S_r^0, \tag{14}$$

$$Q\, S_r^{\min} \le S'_{t,r} \le Q\, S_r^{\max} \qquad \forall t \in \{0, \dots, K\}. \tag{15}$$

Dinámica de balances

Para todo $t \in \mathcal{T}$ y $r \in \mathcal{R}$:

$$S'_{t+1,r} = S'_{t,r} + Q\, p_r^{\mathrm{crew}} + \sum_{m \in \mathcal{M}} p_{m,r}\, \mathrm{run}_{t,m}^{(Q)} - Q\, c_r^{\mathrm{crew}} - \sum_{m \in \mathcal{M}} c_{m,r}\, \mathrm{run}_{t,m}^{(Q)}. \tag{16}$$

Nótese que (16) es íntegramente lineal e integral bajo la cuantización.

Sustentabilidad terminal

Tres variantes típicas:

- **Cíclica** (cierre exacto): $S'_{K,r} = S'_{0,r}$.
- **No decreciente**: $S'_{K,r} \ge S'_{0,r}$.
- **Libre**: sin condición terminal.

Presupuesto energético diario (opcional)

Si se modela energía, imponemos para todo $t \in \mathcal{T}$:

$$\sum_{m \in \mathcal{M}} e_m\, \mathrm{run}_{t,m}^{(Q)} \le EB^{(Q)}. \tag{17}$$

0.5 Objetivos: capacidad, energía y lexicografía

Definimos el costo total de capacidad:

$$\mathrm{CAP} := \sum_{m \in \mathcal{M}} w_m\, C_m. \tag{18}$$

Modos

- **Solo capacidad**: minimizar CAP.
- **Ponderado**: minimizar $W_{\mathrm{CAP}}\mathrm{CAP} + W_{\mathrm{EB}} EB^{(Q)}$.
- **Lexicográfico** (capacidad, luego energía): minimizar primero CAP y, entre empates, minimizar $EB^{(Q)}$.

Implementación lexicográfica por Big-M seguro

Sea $UB(EB^{(Q)})$ una cota superior válida. Una cota simple, cuando $\text{run}_{t,m}^{(Q)} \leq Q\overline{C}_m$, es:

$$UB(EB^{(Q)}) := \sum_{m \in \mathcal{M}} e_m \, (Q\overline{C}_m). \tag{19}$$

Proposición 1 (Lexicografía por Big-M)**.** *Sea* $M := UB(EB^{(Q)}) + 1$*. Entonces minimizar*

$$\text{OBJ} := M \cdot \text{CAP} + EB^{(Q)} \tag{20}$$

es equivalente a la optimización lexicográfica $(\text{CAP}, EB^{(Q)})$*.*

Proof. Para dos soluciones factibles x, y: si $\text{CAP}(x) < \text{CAP}(y)$, entonces

$$\text{OBJ}(x) \leq M\text{CAP}(x) + UB(EB^{(Q)}) < M\text{CAP}(x) + M \leq M\text{CAP}(y) \leq \text{OBJ}(y).$$

Luego toda mejora en capacidad domina cualquier variación posible de energía. Si $\text{CAP}(x) = \text{CAP}(y)$, minimizar OBJ equivale a minimizar $EB^{(Q)}$. □

0.6 SATX como teoría operacional y compilador determinista

El modelo anterior es un ILP puro (variables enteras, restricciones lineales, objetivo lineal). SATX cumple aquí dos funciones formales:

(i) Lenguaje de especificación sobre dominios finitos. La teoría se expresa en términos de variables enteras acotadas y restricciones lineales. La configuración estricta (ancho de palabra y política de constantes) fuerza que el modelo sea *bien tipado* y evita ambigüedades aritméticas en la construcción del IR.

(ii) Compilación a un formato estándar auditable. SATX exporta el IR lineal a formatos portables (LP/MPS), preservando determinismo estructural: secciones completas (objetivo, restricciones, cotas, integridad). Esto habilita auditoría directa del modelo exportado y verificación cruzada con múltiples solvers externos, sin modificar la teoría.

0.7 Validez de resultados y protocolo de verificación

Qué significa "resultado válido"

Dado un conjunto de parámetros y un modelo compilado, una solución es válida si existe una asignación (entera) a $(C_m, \text{run}_{t,m}^{(Q)}, S'_{t,r}, EB^{(Q)})$ tal que:

- satisface todas las restricciones;
- y su valor objetivo coincide con la evaluación aritmética exacta de OBJ.

En modelos enteros lineales con coeficientes enteros, la verificación por sustitución es exacta: basta recomputar cada lado de cada restricción usando aritmética entera.

Chequeo pos-solución (sustitución exacta)

Lema 1 (Verificación por sustitución)**.** *Sea* x *una asignación propuesta. Entonces* x *es factible si y solo si todas las restricciones lineales son verdaderas al sustituir los valores enteros de* x*.*

Proof. Las restricciones son igualdades o desigualdades lineales sobre $\mathbb{Z}$. La satisfacción es semánticamente la verdad de dichas relaciones aritméticas. No hay aproximación numérica si el chequeo se hace en enteros. □

Algoritmo de verificación

1. Verificar cotas e integridad: $0 \le C_m \le \overline{C}_m$, $0 \le \mathrm{run}_{t,m}^{(Q)} \le Q\overline{C}_m$, y $\mathrm{run}_{t,m}^{(Q)} \le QC_m$.
2. Verificar seguridad e inicialización: $S'_{0,r} = QS_r^0$ y $QS_r^{\min} \le S'_{t,r} \le QS_r^{\max}$.
3. Verificar balances: para todo t, r comprobar (16).
4. Si aplica, verificar sustentabilidad terminal: cíclica/no-decreciente.
5. Si aplica energía, verificar (17) para todo t y $0 \le EB^{(Q)} \le UB(EB^{(Q)})$.
6. Verificar objetivo: evaluar OBJ y comparar con el reportado.

0.8 Instancia demostrativa y solución óptima

Se considera la instancia "toy" con $K = 7$ y $Q = 4$ (operación en cuartos de unidad). En esta instancia, el sistema es cíclico y el modelo admite operación fraccional sin bilinearidad.

Parámetros (resumen)

Recursos: O_2, CO_2, H_2O, FOOD, WASTE, NUTR con rangos seguros e inventario inicial. Tripulación: produce CO_2 y WASTE; consume O_2, H_2O, FOOD. Módulos: conversión $CO_2 \to O_2$, reciclaje de agua, compostaje, y granja.

Solución reportada

Una solución óptima (en el sentido lexicográfico capacidad–energía) instala:

$$C_m = 1 \quad \forall m \in \{\texttt{CO2\_to\_O2}, \texttt{Water\_Recycle}, \texttt{Compost}, \texttt{Farm}\},$$

y opera diariamente con:

$$\mathrm{run}_{t,\texttt{CO2\_to\_O2}}^{(4)} = 3, \quad \mathrm{run}_{t,\texttt{Water\_Recycle}}^{(4)} = 4, \quad \mathrm{run}_{t,\texttt{Compost}}^{(4)} = 4, \quad \mathrm{run}_{t,\texttt{Farm}}^{(4)} = 4,$$

para todo $t = 0, \ldots, 6$. En unidades físicas, esto equivale a $u = 0.75$ para `CO2_to_O2` y $u = 1.0$ para los demás.

Lema 2 (Punto fijo de inventarios)**.** *Bajo la solución anterior, el flujo neto diario de cada recurso es cero, por lo que $S'_{t,r}$ permanece constante y la condición cíclica se satisface trivialmente.*

Proof. Basta verificar, por recurso, que producción total diaria = consumo total diario. Por ejemplo, para O_2: la tripulación consume 8; los módulos producen $0.75 \cdot 8 + 1 \cdot 2 = 6 + 2 = 8$. Un cálculo análogo aplica a CO_2, H_2O, FOOD, WASTE y NUTR. Luego $S'_{t+1,r} = S'_{t,r}$ para todo t, y en particular $S'_{K,r} = S'_{0,r}$. □

Energía y objetivo

Con energía por unidad $(2, 1, 1, 3)$, el consumo diario físico es:

$$EB = 2 \cdot 0.75 + 1 \cdot 1 + 1 \cdot 1 + 3 \cdot 1 = 6.5.$$

En escala $Q = 4$: $EB^{(4)} = 26$. El costo de capacidad es $\mathrm{CAP} = 1 + 1 + 1 + 2 = 5$. Con $UB(EB^{(4)}) = 280$, $M = 281$, el objetivo es:

$$\mathrm{OBJ} = 281 \cdot 5 + 26 = 1431.$$

0.9 Extensiones formales

La formulación es un caso base de una teoría operacional de planificación:

- **Robustez por conteo**: contar planes factibles o proyecciones sobre variables observables (p. ej. sólo capacidad).
- **Optimización multi-criterio**: reemplazar Big-M por MaxSAT ponderado o jerarquías por capas.
- **Preimagen e inversión**: dado un estado objetivo, sintetizar estados iniciales o historias consistentes bajo transiciones.

Estas extensiones se benefician de la separación "teoría / compilación / resolución" que habilita SATX.

.1 Implementación SATX (código)

```
"""
Problema: Self-Sustaining Spaceship - síntesis óptima (capacidad + operación).

Se decide (i) capacidad instalada entera por módulo cap_u[m] y (ii) operación
diaria run_q[t,m]
en un horizonte discreto K, manteniendo balances de recursos dentro de rangos
seguros.

Para soportar operación fraccional sin bilinearidad, se introduce una cuantización
 Q:
- cap_u[m] = número entero de unidades instaladas.
- run_q[t,m] = enteros en "cuantos" (p.ej. Q=4 => cuartos de unidad por día).
Se impone run_q[t,m] <= Q * cap_u[m] (lineal).
- Los inventarios S' se escalan por Q: S' = Q * S físico.
Así, con run_q, los balances quedan enteros y lineales.

Objetivo: minimizar capacidad instalada (ponderada) y opcionalmente el presupuesto
 energético
diario EB' (también en escala Q).

Fecha: 2026-01-19
Autor: ASPIE / SATX.
(c) Oscar Riveros. Todos los derechos reservados.
"""

import satx

SAT    = "wsl kissat <"
MC     = "wsl ganak --verb 0 <"
MAXSAT = "wsl open-wbo <"
MIP    = "wsl lp_solve -fmps <"

BITS = 32


def build_spaceship_model(params):
    """
    params:
    K: int >= 1
```

```
    Q: int >= 1  (cuantización; Q=4 => run en cuartos de unidad)
    resources: r -> {safe_min:int, safe_max:int, init:int}   (UNIDADES FÍSICAS; se
 escalan por Q internamente)
    crew: {prod: r->int, cons: r->int}                       (por día, unidades fí
sicas; se escalan por Q)
    modules: m -> {
        cap_ub:int, cap_cost:int, energy:int,
        prod: r->int, cons: r->int                           (por unidad física
completa; NO se escalan)
    }
    sustain_mode: "cyclic" | "nondecreasing" | "free"
    optimize_energy: bool
    objective_mode: "capacity_only" | "capacity_then_energy" | "weighted"
    weights: {W_CAP:int, W_EB:int}
    """

    K = int(params["K"])
    assert K >= 1

    Q = int(params.get("Q", 1))
    assert Q >= 1

    resources = params["resources"]
    crew = params["crew"]
    modules = params["modules"]

    sustain_mode = params.get("sustain_mode", "cyclic")
    optimize_energy = bool(params.get("optimize_energy", True))
    objective_mode = params.get("objective_mode", "capacity_then_energy")
    weights = params.get("weights", {"W_CAP": 1, "W_EB": 1})

    satx.engine(bits=BITS, signed=False, width_policy="strict", const_policy="
strict")

    R = list(resources.keys())
    M = list(modules.keys())

    # -------------------------
    # Variables
    # -------------------------
    cap_u = {m: satx.integer() for m in M}                          # unidades
instaladas (enteras)
    run_q = {(t, m): satx.integer() for t in range(K) for m in M}  # cuantos de
operación (enteros)
    S = {(t, r): satx.integer() for t in range(K + 1) for r in R}  # inventarios
escalados: S' = Q*S_físico
    EBq = satx.integer() if optimize_energy else None              # presupuesto
energético escalado

    # -------------------------
    # Bounds inventarios y estado inicial (escalados por Q)
    # -------------------------
    for r in R:
        smin = int(resources[r]["safe_min"]) * Q
        smax = int(resources[r]["safe_max"]) * Q
        s0   = int(resources[r]["init"]) * Q
        assert 0 <= smin <= smax
        assert smin <= s0 <= smax
```

```
    satx.add(S[(0, r)] == s0)
    for t in range(K + 1):
        satx.add(smin <= S[(t, r)])
        satx.add(S[(t, r)] <= smax)

# -------------------------
# Bounds capacidad y operación
# -------------------------
for m in M:
    cap_ub = int(modules[m]["cap_ub"])
    assert cap_ub >= 0
    satx.add(0 <= cap_u[m])
    satx.add(cap_u[m] <= cap_ub)

for t in range(K):
    for m in M:
        cap_ub = int(modules[m]["cap_ub"])
        satx.add(0 <= run_q[(t, m)])
        satx.add(run_q[(t, m)] <= Q * cap_ub)      # cota global
        satx.add(run_q[(t, m)] <= Q * cap_u[m])    # run <= Q*cap (lineal)

# -------------------------
# Dinámica de balances (enteros, lineal)
#
# S'_{t+1,r} = S'_{t,r}
#           + Q*crew_prod[r] + Sigma_m prod[m,r]*run_q[t,m]
#           - Q*crew_cons[r] - Sigma_m cons[m,r]*run_q[t,m]
# -------------------------
crew_prod = crew.get("prod", {})
crew_cons = crew.get("cons", {})

for t in range(K):
    for r in R:
        crew_prod_r = int(crew_prod.get(r, 0))
        crew_cons_r = int(crew_cons.get(r, 0))
        assert crew_prod_r >= 0 and crew_cons_r >= 0

        inflow_terms = []
        outflow_terms = []

        for m in M:
            p_mr = int(modules[m].get("prod", {}).get(r, 0))
            c_mr = int(modules[m].get("cons", {}).get(r, 0))
            assert p_mr >= 0 and c_mr >= 0

            if p_mr != 0:
                inflow_terms.append(p_mr * run_q[(t, m)])
            if c_mr != 0:
                outflow_terms.append(c_mr * run_q[(t, m)])

        inflow_mod = satx.ssum(inflow_terms) if inflow_terms else satx.z0()
        outflow_mod = satx.ssum(outflow_terms) if outflow_terms else satx.z0()

        inflow = (Q * crew_prod_r) + inflow_mod
        outflow = (Q * crew_cons_r) + outflow_mod

        satx.add(S[(t + 1, r)] == S[(t, r)] + inflow - outflow)
```

```
# -------------------------
# Sustentabilidad final
# -------------------------
if sustain_mode == "cyclic":
    for r in R:
        satx.add(S[(K, r)] == S[(0, r)])
elif sustain_mode == "nondecreasing":
    for r in R:
        satx.add(S[(K, r)] >= S[(0, r)])
elif sustain_mode == "free":
    pass
else:
    raise ValueError(f"sustain_mode inválido: {sustain_mode}")

# -------------------------
# Energía (escalada): EBq >= Sigma_m E_m * run_q[t,m],  For All t
# -------------------------
EBq_ub = 0
if optimize_energy:
    for m in M:
        e_m = int(modules[m].get("energy", 0))
        cap_ub = int(modules[m]["cap_ub"])
        assert e_m >= 0 and cap_ub >= 0
        EBq_ub += Q * cap_ub * e_m

    satx.add(0 <= EBq)
    satx.add(EBq <= EBq_ub)

    for t in range(K):
        e_terms = []
        for m in M:
            e_m = int(modules[m].get("energy", 0))
            assert e_m >= 0
            if e_m != 0:
                e_terms.append(e_m * run_q[(t, m)])
        satx.add((satx.ssum(e_terms) if e_terms else satx.z0()) <= EBq)

# -------------------------
# Objetivo
# -------------------------
cap_cost_terms = []
for m in M:
    w = int(modules[m].get("cap_cost", 1))
    assert w >= 0
    if w != 0:
        cap_cost_terms.append(w * cap_u[m])

CAPCOST = satx.ssum(cap_cost_terms) if cap_cost_terms else satx.z0()

if objective_mode == "capacity_only" or (not optimize_energy):
    objective = CAPCOST
elif objective_mode == "capacity_then_energy":
    # Lexicográfico (capacidad, luego energía) vía Big-M seguro:
    Mbig = int(EBq_ub) + 1
    assert 0 <= Mbig < (1 << BITS)
    objective = (Mbig * CAPCOST) + EBq
elif objective_mode == "weighted":
```

```
            W_CAP = int(weights.get("W_CAP", 1))
            W_EB  = int(weights.get("W_EB", 1))
            assert W_CAP >= 0 and W_EB >= 0
            objective = (W_CAP * CAPCOST) + (W_EB * EBq)
        else:
            raise ValueError(f"objective_mode inválido: {objective_mode}")

        # Vars a observar (para extracción/validación)
        vars_list = []
        vars_list.extend([cap_u[m] for m in M])
        vars_list.extend([run_q[(t, m)] for t in range(K) for m in M])
        vars_list.extend([S[(t, r)] for t in range(K + 1) for r in R])
        if optimize_energy:
            vars_list.append(EBq)

        return {
            "K": K, "Q": Q,
            "R": R, "M": M,
            "cap_u": cap_u,
            "run_q": run_q,
            "S": S,
            "EBq": EBq,
            "objective": objective,
            "vars": vars_list,
            "EBq_ub": EBq_ub,
        }


    def example_params_toy_feasible():
        """
        Toy corregido para que 'cyclic' sea consistente:
        - Water_Recycle produce 5 de H2O por unidad (en vez de 2).
        - Q=4 permite duty-cycle fraccional (p.ej. CO2_to_O2 a 0.75/día => run_q=3).
        """
        return {
            "K": 7,
            "Q": 4,
            "resources": {
                "O2":   {"safe_min": 40, "safe_max": 120, "init": 80},
                "CO2":  {"safe_min":  0, "safe_max":  80, "init": 10},
                "H2O":  {"safe_min": 30, "safe_max": 120, "init": 60},
                "FOOD": {"safe_min": 10, "safe_max":  80, "init": 30},
                "WASTE":{"safe_min":  0, "safe_max":  80, "init":  0},
                "NUTR": {"safe_min":  0, "safe_max":  80, "init": 10},
            },
            "crew": {
                "prod": {"CO2": 8, "WASTE": 3},
                "cons": {"O2": 8, "H2O": 4, "FOOD": 2},
            },
            "modules": {
                "CO2_to_O2": {
                    "cap_ub": 10, "cap_cost": 1, "energy": 2,
                    "prod": {"O2": 8},
                    "cons": {"CO2": 8},
                },
                "Water_Recycle": {
                    "cap_ub": 10, "cap_cost": 1, "energy": 1,
                    "prod": {"H2O": 5},
```

```
                "cons": {"WASTE": 2},
            },
            "Compost": {
                "cap_ub": 10, "cap_cost": 1, "energy": 1,
                "prod": {"NUTR": 1},
                "cons": {"WASTE": 1},
            },
            "Farm": {
                "cap_ub": 10, "cap_cost": 2, "energy": 3,
                "prod": {"FOOD": 2, "O2": 2},
                "cons": {"NUTR": 1, "H2O": 1, "CO2": 2},
            },
        },
        "sustain_mode": "cyclic",
        "optimize_energy": True,
        "objective_mode": "capacity_then_energy",
    }


if __name__ == "__main__":
    params = example_params_toy_feasible()
    mdl = build_spaceship_model(params)

    satx.to_mip(
        format="mps",
        objective=mdl["objective"],
        sense="min",
        path="spaceship.mps",
    )

    # Resolver desde Python (opcional):
    result = satx.mip(
        mdl["vars"],
        objective=mdl["objective"],
        sense="min",
        format="mps",
        solver=MIP,
    )

    print(result["status"], result.get("objective"))
    print("Q =", mdl["Q"])
    if mdl["EBq"] is not None:
        print("EBq =", mdl["EBq"].value, "=> EB =", mdl["EBq"].value / mdl["Q"])
    for m in mdl["M"]:
        print("cap_u", m, mdl["cap_u"][m].value)

    Q = mdl["Q"]

    print("=== RUN SCHEDULE (physical units) ===")
    for t in range(mdl["K"]):
        e_t_q = 0
        row = []
        for m in mdl["M"]:
            rq = mdl["run_q"][(t, m)].value
            row.append((m, rq, rq / Q))
            e_t_q += int(params["modules"][m]["energy"]) * rq
        print("t=", t, row, "E_t_q=", e_t_q, "E_t=", e_t_q / Q)
```

```
        print("=== STATES ===")
        for t in range(mdl["K"] + 1):
            print("t=", t, {r: mdl["S"][(t, r)].value / Q for r in mdl["R"]})
```