UDEMY-Optimization with Python: Solve operation research problemas

SECTION 4: Introduction to mathematical modelling

Modelizar es convertir un problema del mundo real en un problema matemático.
Los problemas de optimización son problemas de maximización / minimizacion de una variable de costo.

Steps para resolver un problema de optimización:

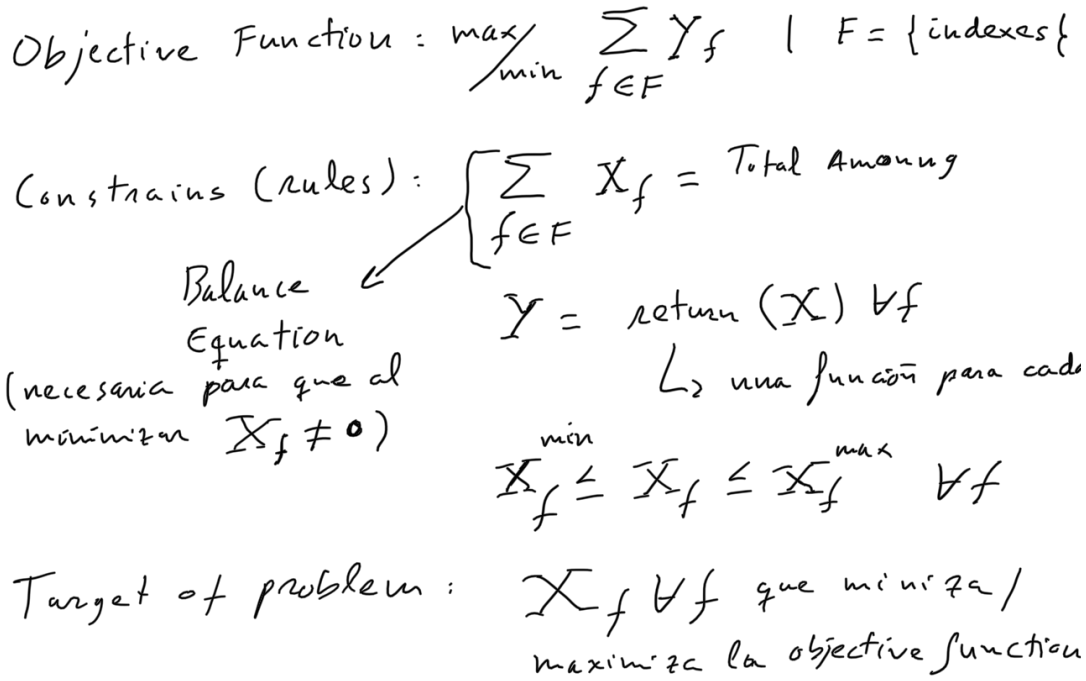
- **PROBLEM UNDERSTANDING**: entendimiento del problema del mundo real.
- **PROBLEM MODELLING**: matematizacion del problema.
- **RESOLUTION** (objetivo del curso):
  - FRAMEWORK: Es la herramienta a usar para pasar el problema matemático al solver. En es curso la herramienta es Python (programming languages).
  - SOLVER: El solver ya esta desarrollado e integrado en el framework. Es el cerebro del algoritmo el cual resolverá el problema matemático.
- **RESULTS PRESENTATION.**

Conviene distinguir entre:

- **Indexes**: Permiten identificar las distintas opciones / vías de una misma variable, añadiendo por decirlo así, otra dimensión al problema. Ej: variable gasto (G), puede ser en productos A y B (indexes)—> Por tanto los gastos serian: GA, GB. Esto es muy importante pues ayuda en la fase de matematizacion del problema y ademas habra que tenerlo en cuenta en el framework Pyono. Puede haber mas de set de index trabajando al mismo tiempo (ej. Aij —> matriz 2D de la variable/ parametro A). ***La selección de los indexes es muy importante y es lo primero que debería hacerse.***
- **Variable**: No conozco antes de la optimización.
- **Sets**: En algunos problemas (como por ejemplo en el *route problem*), es necesario trabajar con sets de indexes en vez de indexes (ver ejemplo en python).
- **Parámetro**: Conozco antes de la optimización
- **Objective function**: Describe el objetivo. Ej: minimize(GA + GB)
- **Constrains**: Son las condiciones de contorno.

El formato matemático en el que se puede generalizar un problema de

optimización en general seria la siguiente:



NOTA: La formulación del problema cambia bastante en el *route problem*.

Tipos de variables (es muy importante distinguir el tipo de variable pues de esto depende por ej el solver a usar):

- Continuous.
- Integer.
- Boolean (binary):
  - Estas variables son muy usadas para mostrar decisión de "hacer algo o no".
  - Truco: Para hacer IF statements con variables binarias, se puede jugar con el propio valor de las mismas construyendo constrains del tipo "X1 <= X2" para indicar que "X1 puede ser True solo cuando X2 sea True".
  - Truco: Para poner constrais, también se puede jugar con la suma True's (1) y False's (0).
- Others.

SECTION 5: LINEAR PROGRAMMING (LP)

LP: Cuando no hay multiplicación entre variables o funciones no lineales (funciones trigonométricas, etc). Ademas, todas las variables son continuas (no

integer).

FRAMEWORKS:

- ***ORTOOLS***: Very simple framework but only for Linear problems.
- ***SCIP***: Es muy simple de usar, no hay que seleccionar el solver y se puede usar tanto para Linear como para Non-Linear problemas. El framework se encarga automáticamente de hacer todos los settings. La instalación es un poco mas complicada pues requiere la instalación de una librería previa, meterlo en el path del sistema y entonces ya se puede instalar la librería Python.
- ***PYOMO***: Es fácil de usar. Admite muchos solvers y la resolución de tanto Linear como Non-Linear problemas. Requiere la instalación de los solvers. Me parece ademas muy interesante la información resumen del problema que te muestra por pantalla al ejecutar el script. Es el mas completo de los frameworks aunque puede ser mas difícil de pythonizar para grandes problemas.
  - Documentation: https://pyomo.readthedocs.io/en/stable/index.html
- ***PuLP***: Es muy simple y fácil de implementar en Python. Solo admite Linear problems. Hay mucha documentación disponible. Es una de la librerías predilectas para Python en LP. Si no se selecciona ningún solver, por defecto se usa el CBC solver.

SOLVERS:

- ***GLOP***: Linear solver from Google.
- ***Gurobi*** y ***CPLEX***: Son los solvers comerciales mas famosos para LP. Es necesaria la activación de licencia (aunque hay licencias para uso académico).
  - Gurobi es muy rápido.
  - CPLEX es de IBM.
- ***GLPK***: Es un open source solver que puede ser usado en PYOMO para resolver Linear Problems. Es muy bueno y muy usado.
- ***CBC***: Bien conocido Linear Programming solver disponible para Pyomo.

Framework (AML)	Linear Problems	Nonlinear Problems	How easy to start with	How easy to configure a new solver and about documentation
Pyomo	X	X	High	High
Ortools	X		Very High	Low
PuLP	X		High	High
SCIP	X	X	Very High	Not possible / Low
SciPy	X	X	Low	Medium

Solver	Linear Problemas	Nonlinear	Free / Commercial
Gurobi	X		COMMERCIAL
Cplex	X		COMMERCIAL
CBC	X		FREE
GLPK	X		FREE
IPOPT		X	FREE
SCIP	X	X	FREE
Baron		X	COMMERCIAL

Table: List of frameworks and solver.

Cual es el mejor a usar?

- En general, para resolver un problema rápidamente, el mejor parece ***SCIP*** por permitir Linear / Non-Linear problemas y no tener ni que instalar ni seleccionar solvers.
- Para pythonizar grandes Linear problemas, ***PuLP*** parece la mejor opción aunque sin olvidar que no admite Non-Linear problems.
- ***Pyono*** parece el mas completo, la libreria de referencia.

> NOTA: Yo me estoy centrando en usar Pyomo pues parece el mas usado. No obstante, SCIP es super-flexible y muy muy sencillo. El profesor esta usando ambos dos en sus ejemplos aunque se centra mas en Pyomo.

SECTION 6: WORKING WITH PYOMO

Con el comando pyomo help —solvers, se lista todos los solvers que admite la librería.

A la hora de crear variables, es posible crear arrays con las dimensiones que se quieran y acceder a ellas como si fueran un array de *numpy:*

- Ej: Array 1D de 5 dimensiones: *pyo.Var(range(5), bounds=(0,None))*
- Ej: Array 2D de 5x3 dimensiones: *pyo.Var(range(5), range(3), bounds=(0,None))*

> NOTA: Para acceder a una variable con varias dimensiones es como si fuera un array numpy, pero no se le puede aplicar funciones con la operación vectorizada (por ej. np.sum(), sum(), etc)

Los sumatorios se construyen con la función de Python: *sum*. Estos pueden ser aplicados también sobre una variable a ser optimizada:
sumatorio = sum([Var[id] for id in ids])

La definicion de constraints cambia ligeramente cuando es una constraint individual o una lista:

- Individual: *model.c1 = pyo.Constraint(expr = x_sum + y <= 20)*
- *Lista:*

```
model.c2 = pyo.ConstraintList()
for i in range(1,6,1):
    model.c2.add(expr = x[i] + y >= 15)
```

La función pprint() es muy interesante para visualizar el problema pythonizado:

- Ej. Print todo el problema al final: *model.pprint()*
- Ej. Print una variable (con las dimensiones que sea): *model.VAR.pprint()*
- Ej. Print un constrain: *model.CONSTRAIN.pprint()*
- También es posible ver un summary: *results = opt.solve(model); print(results)*

SECTION 7: MIXED-INTEGER LINEAR PROGRAMMING (MILP)

Constraints y Objective Function tienen parámetros enteros y una o varias variables a optimizar son enteras (también puede haber alguna solución continua, de ahi que es un *mixed*).

Cuando se dice entero, puede ser *integer* e incluso *boolean*.

Para definir una variable, hay que seguir el siguiente orden de los parámetros: *pyo.Var(range(1,6,1), within=Integers, bounds=(0,None))*

SECTION 8: NON-LINEAR PROBLEMS (NLP)

En este tipo de problema, puede haber tanto en el objective fucntion como en algun constraint alguna non-linear function (ej: log, trigonometric, polynomical, etc).

Es mucho mas difícil para el solver conseguir resultados. Puede que no sea posible encontrar soluciones optimas globales para el problema o solver seleccionado, solo **soluciones locales**, o puede que estén disponibles de ambos tipos. Para enfocarse alrededor de una solución local basta con inicializar las variables en su creación: *model.x = pyo.Var(initialize = 0, bounds=(-5,5))*. En el caso de no inicializar, sera extraída la **solución global** (si es posible). También existe la opción de asignar la maxima tolerancia del error cometido para encontrar una solución (un max / min): *opt.options['tol'] = 1e-6* .

Un solver recomendable a usar para este tipo de problemas es ***IPOPT***.

Para definir funciones trigonométricas en la expresión del la Objective Function basta con escribirlo tal cual:
*model.obj = pyo.Objective(expr= cos(x+1) + cos(x)*cos(y), sense=maximize)*

SECTION 9: MIXED-INTEGER NONLINEAR PROGRAMMING (MINLP)

Estos son los tipos de problemas mas complejos de ser resueltos. Muchas veces sera usado para resolver este tipo de*compositions*, *artificial inteligence* or *heuristics* en vez de la metodologia usada hasta ahora.

Usando la metodología que hemos usado hasta ahora, es decir ***la extracción de soluciones por métodos exactos***, con Pyomo por ejemplo usaríamos el solver *Couenne* para resolver el problema. El problema es que este solver **no esta disponible para MacOs.**

Como alternativa se puede seguir usando *IPOPT* pero la solución sera algo menos precisa. Otra opción es usar la técnica de **descomposición** que divide el problema en un problema linear (mixed-integer) y non-linear, teniendo que ser usados dos solvers, uno para cada tipo de problema. Pyomo se encarga de hacer la descomposición de manera automática:
 # solver election = DECOMPOSITION
opt = SolverFactory('mindtpy')
 # linear and non-linear solvers selection and urn
opt.solve(model, mip_solver='cbc', nlp_solver='ipopt')

SECTION 10: GENETIC ALGORITHM AND PARTICLE SWARM

GENETIC ALGORITHM

Es un tipo diferente de algoritmo que se basa en la evolución genética presente en la naturaleza para obtener soluciones casi-exactas. En vez de usar una Objective Function usa un Fitness function la cual evalúa de manera aleatoria. Entonces se evalúa las soluciones para seleccionar las mas aproximada en base a un evolutionary approach.

Es muy importante recordar, que todo lo visto anteriormente (linear programming, etc) supone la resolución de problemas matemáticos de una manera **exacta** mientras que ahora estamos hablando de **soluciones aproximadas**. Este tipo de algoritmos es conveniente usar ante problemas de gran tamaño. Las metodologías exactas pueden ser realmente pesadas computacionalmente mientras que esta técnica aproximada es mucho mas

ligera.

A cada individuo le corresponde un vector de características (este vector es llamado *cromosoma*). Cada iteración es llamada una **generación** y posee los siguientes procesos:

- Fitness function: Evaluación para conseguir posibles soluciones.
- Mutations: Pequeñas variaciones random de los individuos .
- Crossover: Cruce de dos soluciones para crear una mejor pudiendo crearse varias generaciones.
- Stop criterium: Se le aplica un criterio para detener el proceso iterativo (num de generaciones, num de generaciones sin mejora en la solución, calidad de solución, etc).
- Si el stop criterium no detiene el proceso, vuelta a empezar (nueva generación).

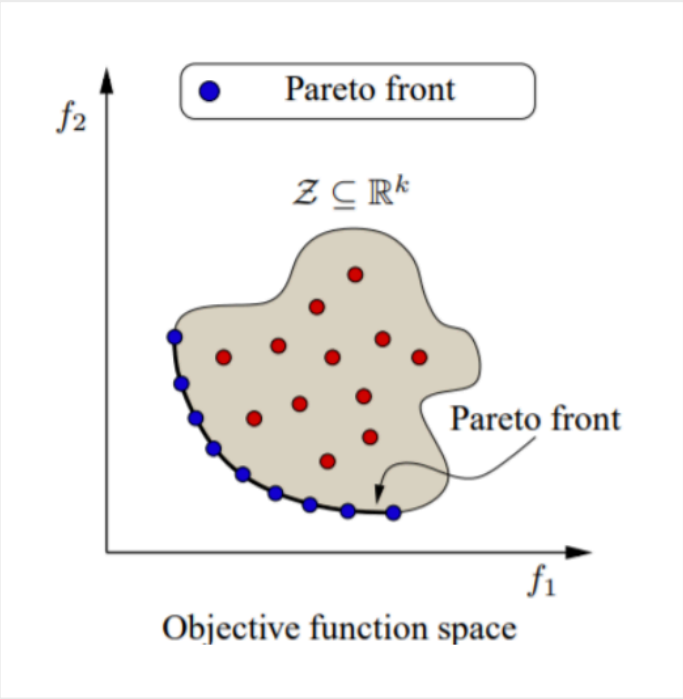
Esta técnica es muy flexible, puede ser usada en muchos tipos de problemas. Eso si, no garantiza encontrar la global solution.

Liberia Python: https://pypi.org/project/geneticalgorithm/

MULTI-OBJECTIVE PROBLEMS

Este tipo de problemas es que el requiere optimizar mas de una Objective Function al mismo tiempo de distinta naturaleza. En el caso de tener la misma naturaleza, las dos funciones podrían agruparse en una sola (ej. Caso de que las dos funciones a maximizar sean de dinero invertida, podría maximizarse directamente la suma de ambas funciones objetivo convirtiéndose en un problema de single Objective Function).

Para resolver este tipo de problemas, hay que encontrar el Pareto front en el espacio de objectives functions (o espacio de soluciones), es decir, aquellas soluciones en el especio que son mejores que el resto:



En este ejemplo, se podría adoptar una estrategia basada en el Pareto front para objener la solución mas optima para ambas funciones: el punto medio del Pareto front que en este caso es el punto mas cercano al punto (0,0).

One of the most known algorithms is the **NSGA-II**, which is a variation of the genetic algorithm. And to solve multi-objective problems using NSGA-II, I recommend you to read about the Pymoo package (https://pymoo.org/).

PARTICLE SWARM (PSO)

Es otra aproximación que simula como trabaja las abejas en la naturaleza. Requiere la definicion de un punto inicial. Una buena seleccion de dicho punto hace que el algoritmo trabaje mas rápido.

Sera usada la libreria **pyswarm** (https://pypi.org/project/pyswarm/).

SECTION 11: CONSTRAIN PROGRAMMING (CP)

Problemas en los que los constrains son mas importantes que la Objective Function. No obstante, en este tipo de problema se puede decidir maximizar / minimizar una objective functions pero los resultados en algunos problemas específicos serán mejores con CP al enfocarse mas en el dominio de los constrains. En caso de no definir una Objective Function este approach te permite sacar todas las posibles soluciones que cumplen los constrains definidos.

Hay dos casos en los que es preferible CP:

- Dominio de los constrains es muy extrecho: No hay mucho campo para conseguir la mejor solución por lo que pueden ser mostradas todas las posibles soluciones.Ej: Scheduling problems.
- Cuando hay muchas soluciones para el mismo valor del Objective Function. Ej: Esto puede suceder en routing problems.

SECTION 12: SPECIAL CASES (que pueden ser linearizados)

Linearizacion

Los problemas no lineales no garantizan obtener la global solution de un problema, es decir, la mejor solución para todo el dominio. Es por eso que hay que intentar siempre que se puede usar linear solvers. Para conseguir esto existe la posibilidad de **linealizar** un problema. Eso si, no es siempre posible linearizar un problema no lineal, solo es posible en algunos casos especiales que se van a mostrar ahora:

Second Order Cone Problem

El solver Gurobi (non-free) usado en Pyomo permite la linealizacion de un problema no lineal. Gurobi, a pesar de ser un linear-solver, es capaz linealizar el Second Order Cone problem automaticamente.

Non-Convex Quadratic Problem

Gurobi también puede resolver Non-Convex Quadratic Problem (igual que el anterior pero incluyendo otro constrain no linear no cónico). En este caso hay que incluir la siguiente opción: *opt.options['NonConvex'] = 2* después de la selección del solver (El *2* es un parámetro interno de *Gurobi* que le indica como resolver este tipo de problema).

Convex vs Non-Convex problem

La diferencia entre estos dos tipos de problemas es cuando se traza una linea entre dos puntos del dominio y todos los puntos de la misma están dentro del dominio. Este es el caso de Convex problem. En el caso de Non-Convex problem esto no sucede (ver en la figura de abajo).

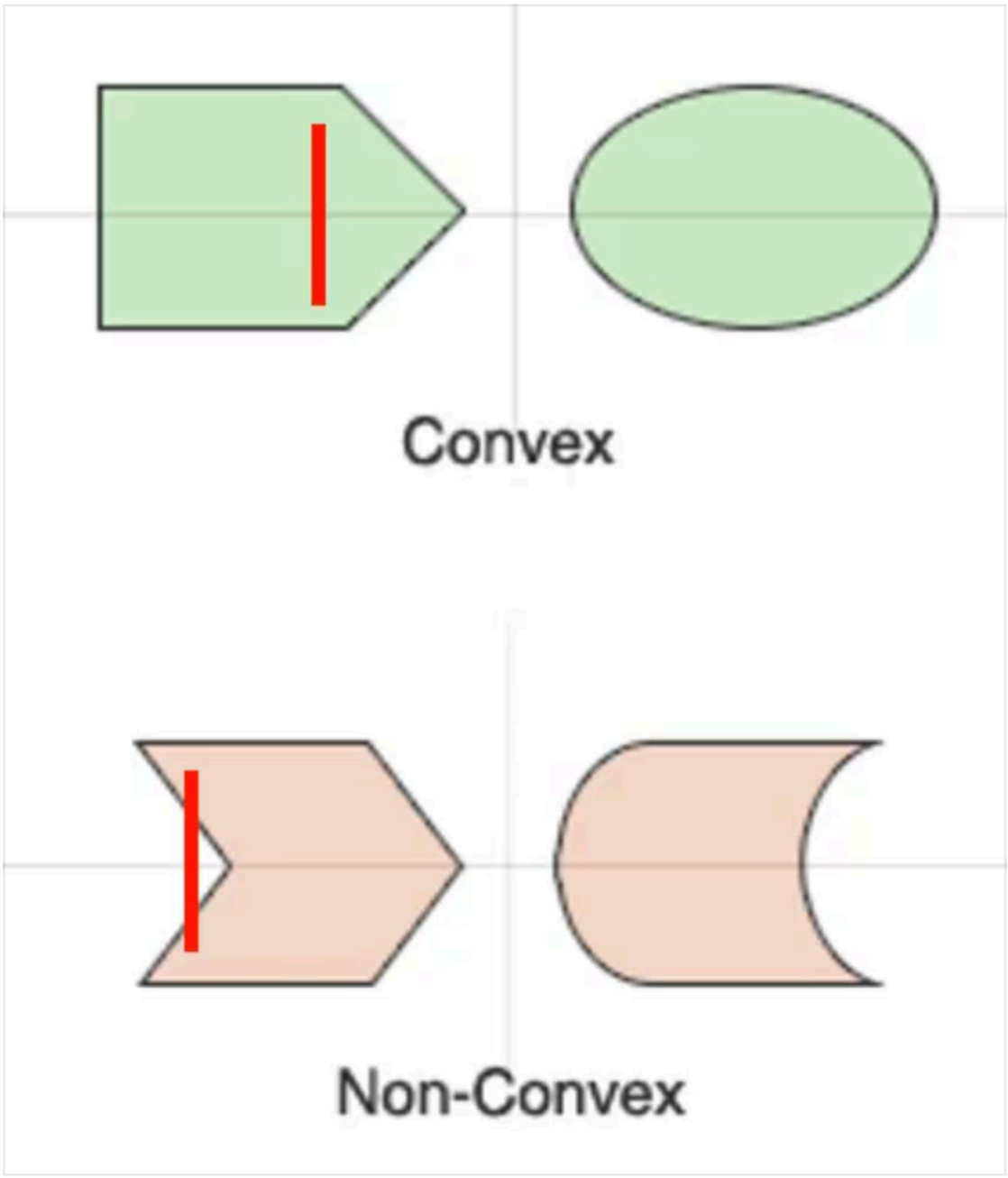


Figure: Convex vs Non-Convex.

No obstante, no es sencillo discernir si un problema es Convex o Non-Convex. Lo que se puede hacer es prueba y error, es decir, intentar:

- Resolver el problema como si fuera Convex con *Gurobi / Pyomo* (linealizar) y si da error es porque realmente es Non-Convex.
- Intentar resolver el problema con un solver Non-Linear directamente.

Vehicle Routing Problem (VRP)

Hay solvers para resolver de manera especifica este tipo de problema (ademas de usar Linear or Mixed Linear Programming). Es el caso de ***OR-Tool*** (https://developers.google.com/optimization/routing). Esta herramienta simplifica enormemente la manera de resolver este tipo de problema.

Linearizacion para casos Binary variable * Continuos variable

En el caso de tener un constrain en el que se multiplican un binary variable y un continuos variable tenemos un problema Non-Linear. En este caso se puede usar una técnica de linearizacion llamada ***BigM***. Esto consiste en:

El constrain *C = b * x*, tal que *x* es continuous y *b* es binary,

transformarlo en la forma equivalente:
*-b * M <= C <= b * M*
*-(1-b) * M <= C - x <= (1-b) * M*
donde *M* es un parámetro escogido muy grande (siempre mas grande que el max de *x*).

Tomando el ejemplo de un binary variable que solo puede tomar valores *0* y *1* tendríamos:
Para *b = 0 —> C = 0*, *x* puede ser cualquier valor.
Para *b = 1 —> C = x*, *x* puede ser cualquier valor.

Ademas de la aplicación directa de este técnica, el solver *Gurobi* también es capaz de hacer esto automáticamente.

Linearizacion para casos Binary variable * Binary variable

En el caso de tener un constrain en el que se multiplican un binary variable y un binary variable el constrain puede ser reformulado:

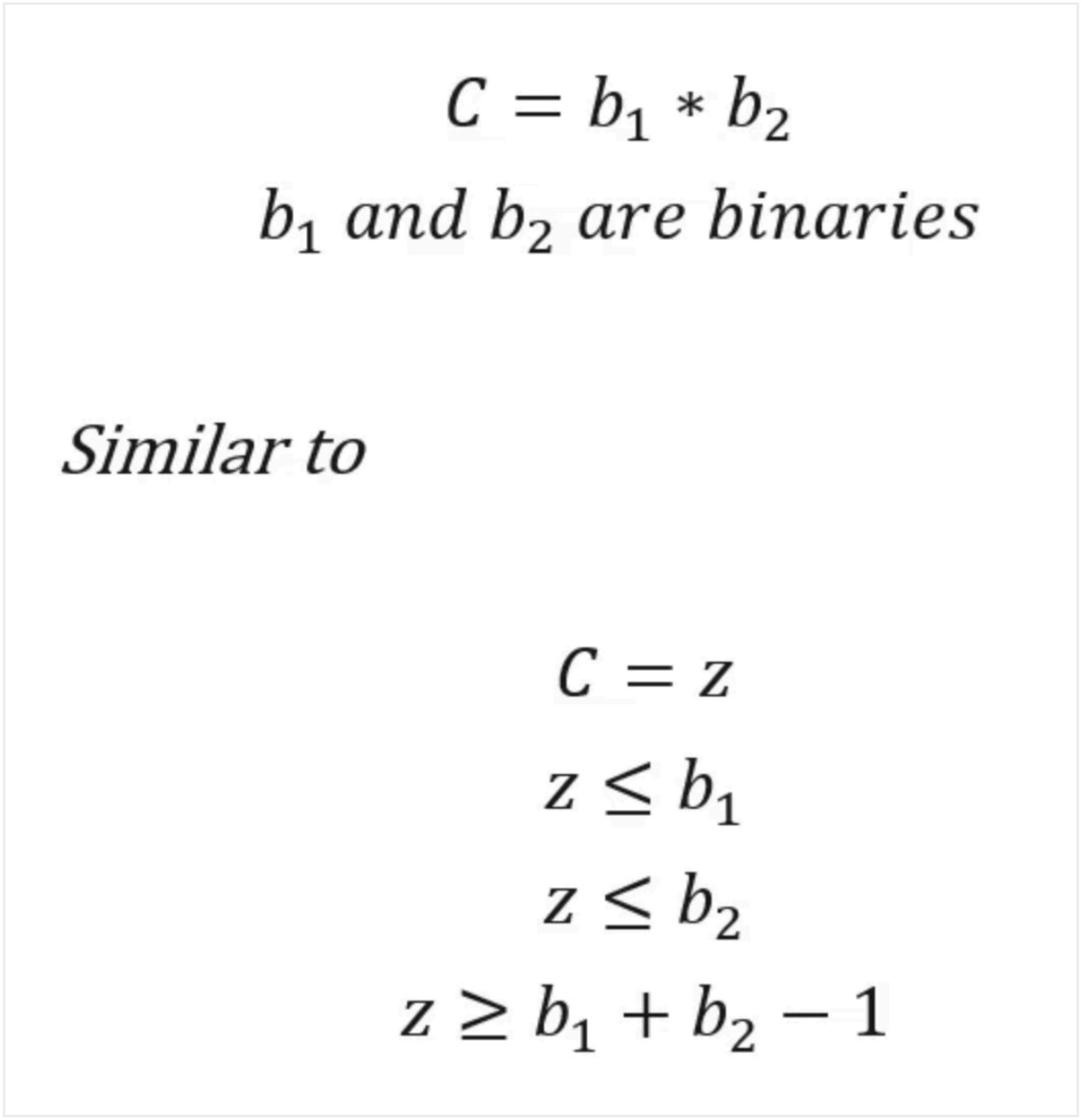


Figure: binary plus binary case.

, donde *z = b1 * b2* .

Con esta reformulación el problema puede ser resuelto como un problema linear (por ejemplo usando el solver *gltk*). No obstante, *gurobi* de nuevo es capaz de resolverlo automáticamente sin reformular (linealizar).

SECTION 13: ADVANCED FEATURES FOR PYOMO

En esta sección se van a mostrar advanced features de Pyomo a través de un use case.

- El use case propuesto usa una variable 2D cuya una de las dependencias es temporal.
- Incluir tee = True permite mostrar mas información del solver: *results =*

opt.solve(model, tee=True). Entre la nueva información mostrada, hay una tabla que muestra las iteraciones necesarias para encontrar la solución. La columna *gap* muestra en % cuanto de lejos esta la iteración de encontrar la solución optima. Si el valor de gap de una iteración es cero quiero decir que se ha obtenido la solución mas optima exactamente, no es posible por tanto encontrar una solución mejor. También es interesante la ultima columna que muestra el tiempo invertido en cada iteración.

- También es posible definir un valor de *gap limite* al solver. Esto puede ser util cuando la solución de un problema tarda demasiado para conseguir la mejor solución posible pudiendo ganar solo unas "décimas" de exactitud. Al relajar la expectativa, es decir al imponer un *gap mínimo* mayor que cero puede que el proceso tarde mucho menos.
- También es posible incluir un time limit en segundo en el solver: *results = opt.solve(model, tee=True, timelimit = 10)*
- También es posible un constrain con desigualdades de la forma: A <= x <= B. Originalmente, para hacer esto creamos dos constrains pero realmente no es necesario y pueden ser incluidos ambos bounds en una sola linea. La manera seria: *model.C.add(pyo.inequality(A, x, B))*.
- *Pyomo* proporciona una función de sumacion: *pyo.summation(x)* que reemplaza por ejemplo a *sum([x[m,t] for m in range(1,M+1) for t in range(1,T+1)])*. Lo que hace es sumar todos los elementos de una variable (todas las dimensiones). Cuidado cuando se quiera hacer una sumacion de una variable para una sola dimension por ej., en este caso no valdría.
- También es posible inicializar *parameters*, *sets* y *set ranges* dentro de objetos pyomo. Esto es recomendable para grandes problemas pues te permite cambiar mas adelante en el código el parámetro por ejemplo afectando a todo el problema. Hasta ahora lo definiamos sin incluirlos en pyomo objects, lo definiamos a través de variables Python.
- Es muy interesante el uso de *rules* en pyomo. Mediante *pyo.Constrain(rule = function_rule)*, es posible pasar una expresión dentro de una función (en su *return*).
- Además es posible inicializar una posible solución para que el algoritmo tenga que trabajar menos. Hay que incluir la opción *warmstar* en el Solver.
- Plomo también posee una extensión para resolver Differencial Algebraic Equations (DAE):

Dynamic Optimization with pyomo.DAE — Pyomo 6.2 documentation

pyomo.readthedocs.io