Model development with IBM ILOG CPLEX Optimization Studio V12.5

“Teach the Teachers”
Charlotte, NC
May 15, 2013

Course Outline

- Introduction to Optimization with IBM ILOG CPLEX Optimization Studio
- Working with OPL
- Working with IBM ILOG Script: basic tasks
- Solving Simple MP Problems
- Scheduling with CP Optimizer
- Supply Chain Network Design with LogicNet Plus XE

Introduction

Introductions

- About your instructor
  - Company
  - Experience
  - Expertise

- About yourself
  - Company
  - Role, position
  - Project
  - Expectations

Course description

- Duration: 3 days

- Purpose
  - Learn how to use IBM ILOG CPLEX Optimization Studio

- Audience
  - Operations Research (OR) professionals and others with similar profiles who will be creating optimization-based business applications.

Course materials

- Workshop exercise files (“work” and “solution” files)
- Slide show presented by the instructor (most slides also appear in the workbook)

Prerequisites

- Working knowledge of the Microsoft Windows operating system
- Knowledge of basic algebra
- Basic knowledge of mathematical programming and modeling concepts

Lesson outline

- **Core lessons:**

1. Introduction to IBM ILOG CPLEX Optimization Studio
2. Working with OPL
3. Working with IBM ILOG Script: basic tasks
4. Solving simple LP problems
5. Solving simple CP problems
6. Infeasibility
7. Data consistency
8. Linking to spreadsheets and databases

- **Optional modules:**

9. Scheduling with CP Optimizer
10. Integer and mixed-integer programming
11. Piecewise linear problems
12. Network models
13. Quadratic programming
14. Flow control with IBM ILOG Script
15. Integrating OPL models with applications
16. Multi-threaded processing
17. Optimization engines and algorithms
18. Performance tuning
19. Optional appendix: The CPLEX Studio IDE graphical user interface

Course objectives

At the end of this course, you should be able to:

- Describe how IBM ILOG CPLEX Optimization Studio fits with the other IBM ILOG optimization products
- Describe the CPLEX Studio features and underlying algorithms
- Use OPL to write an optimization model
- Transfer data between CPLEX Studio and external data sources
- Use IBM ILOG Script for pre-processing, post-processing, and (optionally) flow control
- Use the OPL Application Programming Interfaces (APIs) to integrate an OPL model into an existing application

Introduction to Optimization with CPLEX Optimization Studio

Unit objectives

At the end of this lesson, you should be able to:

- Describe how the IBM ILOG optimization products work together to address optimization problems.
- List the optimization techniques and associated engines available through CPLEX Studio.
- Describe the basic functionality of the IDE.
- Describe what an OPL project is and how OPL projects are organized and managed.
- Point out the basic model elements, namely data, decision variables, objectives and constraints, when looking at an OPL model.

Introduction to Optimization with CPLEX Optimization Studio

- The IBM ILOG optimization products
- Inside CPLEX Studio
- Example: a production planning problem
- Checkpoint

The IBM ILOG optimization products

Learning objective

- Describe the IBM® ILOG® optimization products for custom and packaged applications.

Sections

- The IBM ILOG optimization products
- Benefits of IBM ILOG products for optimization

Operations research and optimization

Operations Research (OR) = the science of better decisions

How to best allocate aircrafts and crews?

inventory cost vs.
customer satisfaction?

Optimization helps businesses to:

- create the best possible plans
- explore alternatives and understand trade-off
- respond to changes in business operations

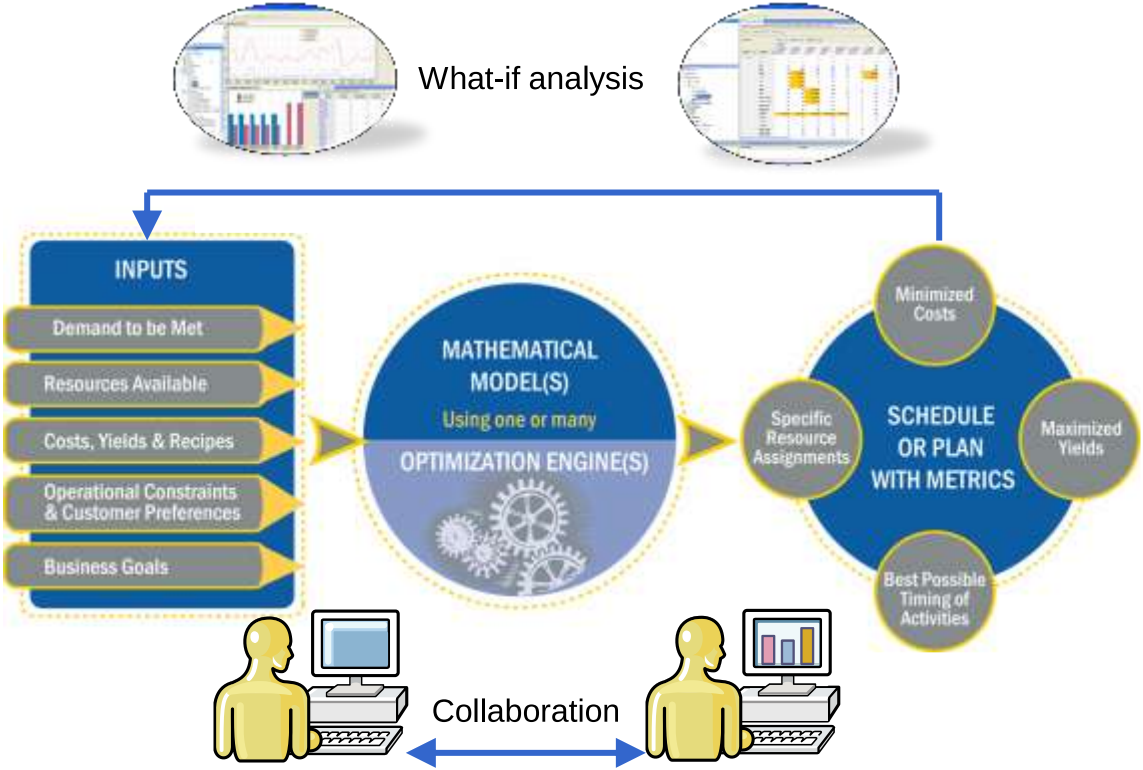
What to build,
where and when?

Risk vs. potential reward?

Cost vs.carbon emission?

The IBM ILOG optimization products bring the power of OR to business users

How does optimization support decision making?



What-if analysis

INPUTS

- Demand to be Met
- Resources Available
- Costs, Yields & Recipes
- Operational Constraints & Customer Preferences
- Business Goals

MATHEMATICAL MODEL(S)

Using one or many

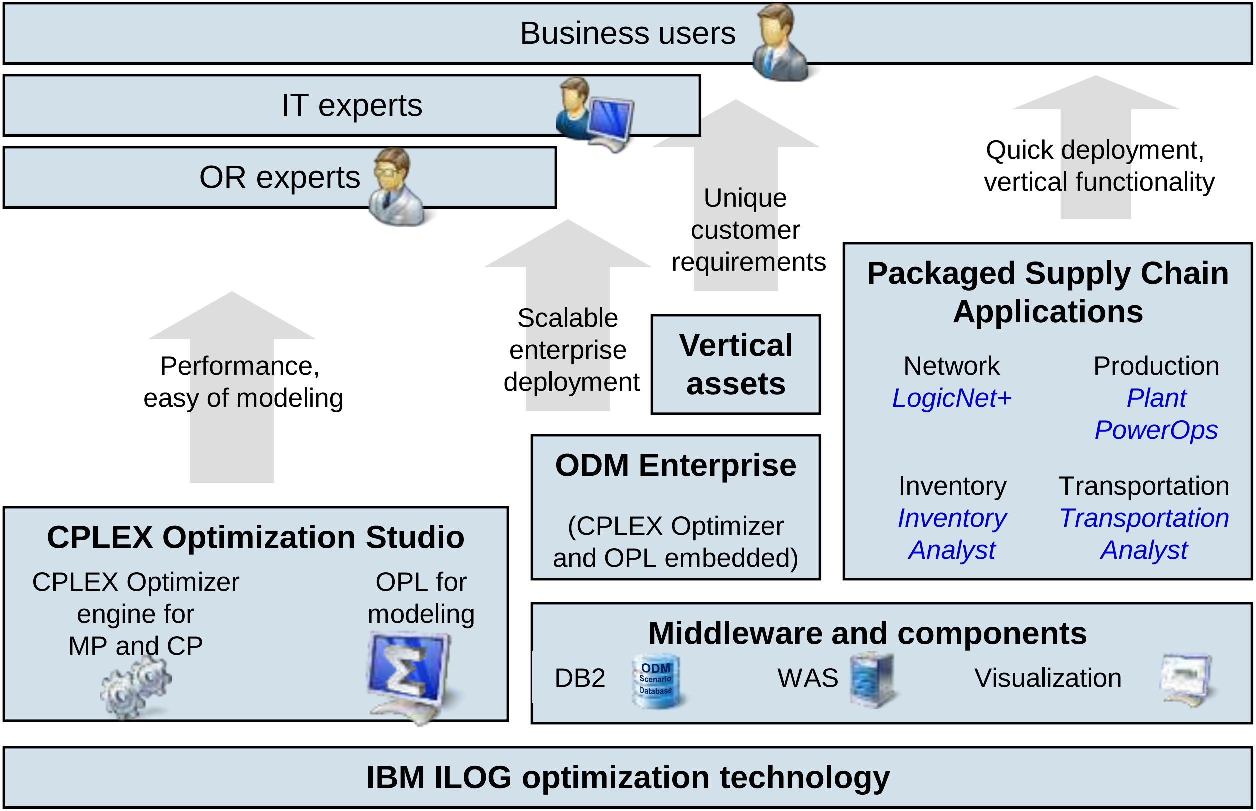
OPTIMIZATION ENGINE(S)

SCHEDULE OR PLAN WITH METRICS

- Minimized Costs
- Specific Resource Assignments
- Maximized Yields
- Best Possible Timing of Activities

Collaboration

The IBM ILOG optimization products



Business users

IT experts

OR experts

Performance, easy of modeling

Scalable enterprise deployment

Unique customer requirements

Quick deployment, vertical functionality

Packaged Supply Chain Applications

Network *LogicNet+*

Production *Plant PowerOps*

Inventory *Inventory Analyst*

Transportation *Transportation Analyst*

Vertical assets

ODM Enterprise

(CPLEX Optimizer and OPL embedded)

CPLEX Optimization Studio

CPLEX Optimizer engine for MP and CP

OPL for modeling

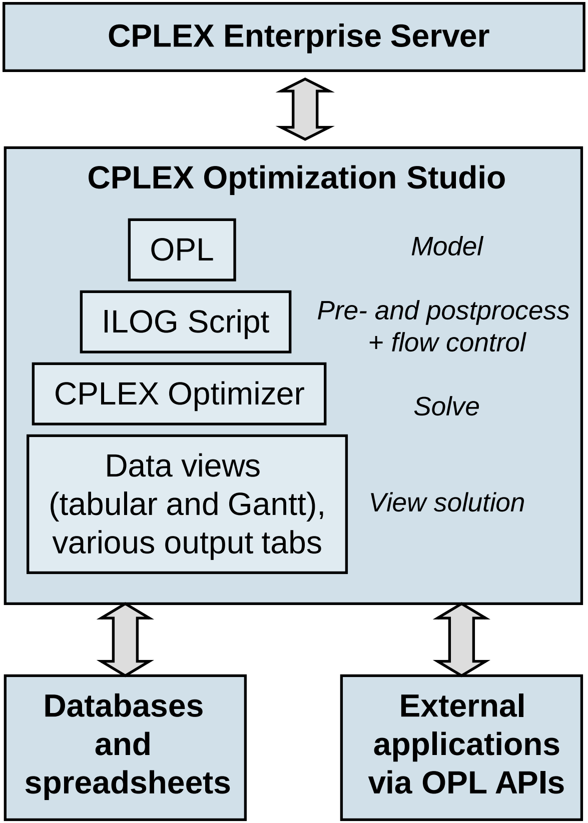
Middleware and components

DB2 WAS Visualization

IBM ILOG optimization technology

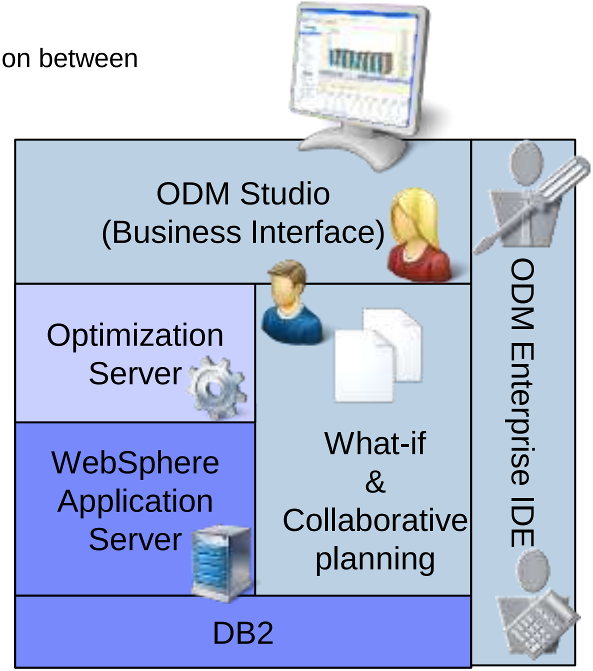
What is IBM ILOG CPLEX Optimization Studio?

- An Integrated Development Environment (IDE) for model building, debugging and tuning
- Optimization Programming Language (OPL) for modeling
- IBM ILOG Script for pre- and postprocessing, and flow control
- CPLEX Optimizer solution engine with:
  - Several MP optimizers
  - CP Optimizers for CP problems
- Database and spreadsheet connectivity
- OPL APIs for integration with external applications
- CPLEX Enterprise server for remote, multi-user solving of OPL applications


CPLEX Enterprise Server
CPLEX Optimization Studio
OPL
Model
ILOG Script
Pre- and postprocess + flow control
CPLEX Optimizer
Solve
Data views (tabular and Gantt), various output tabs
View solution
Databases and spreadsheets
External applications via OPL APIs

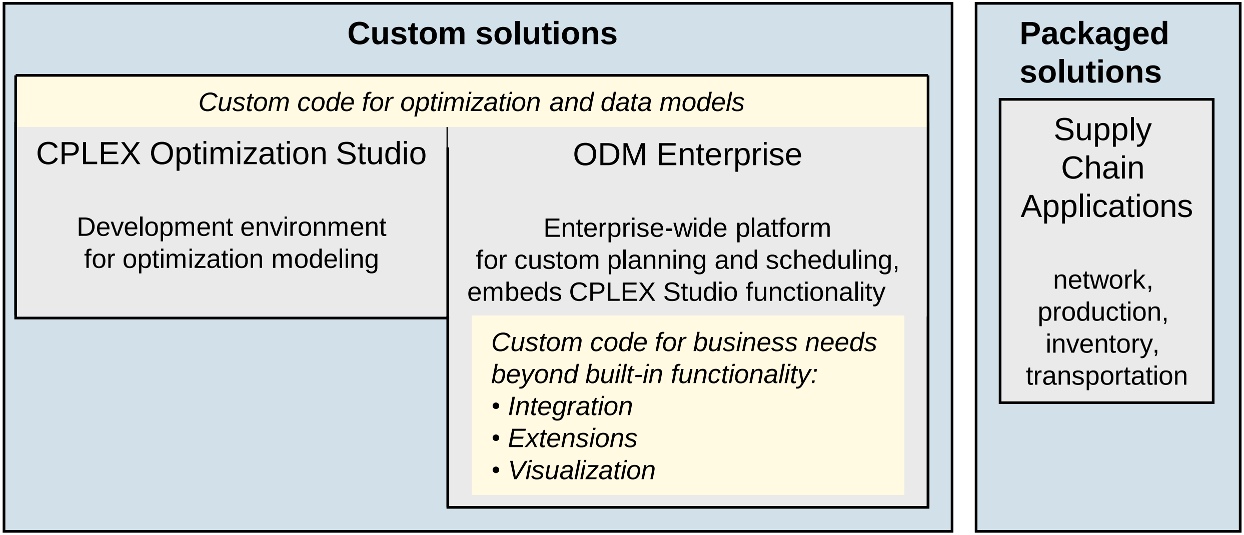
What is IBM ILOG ODM Enterprise?

- Platform for planning and scheduling solutions
- Supports client-server deployment & collaboration between planners
- Platform contains
  - ODM Studio
  - ODM Optimization Server
  - ODM Enterprise IDE
  - OPL for modeling
  - CPLEX Optimizer solution engine:
    - Several MP optimizers
    - CP Optimizer for CP problems



Custom solutions versus packaged applications

- Some business problems are relatively standard and addressed with packaged applications
  - IBM ILOG Supply Chain Applications
- Others include unique challenges that require custom solutions
  - IBM ILOG CPLEX Optimization Studio
  - IBM ILOG ODM Enterprise



Custom solutions

Custom code for optimization and data models

CPLEX Optimization Studio

Development environment
for optimization modeling

ODM Enterprise

Enterprise-wide platform
for custom planning and scheduling,
embeds CPLEX Studio functionality

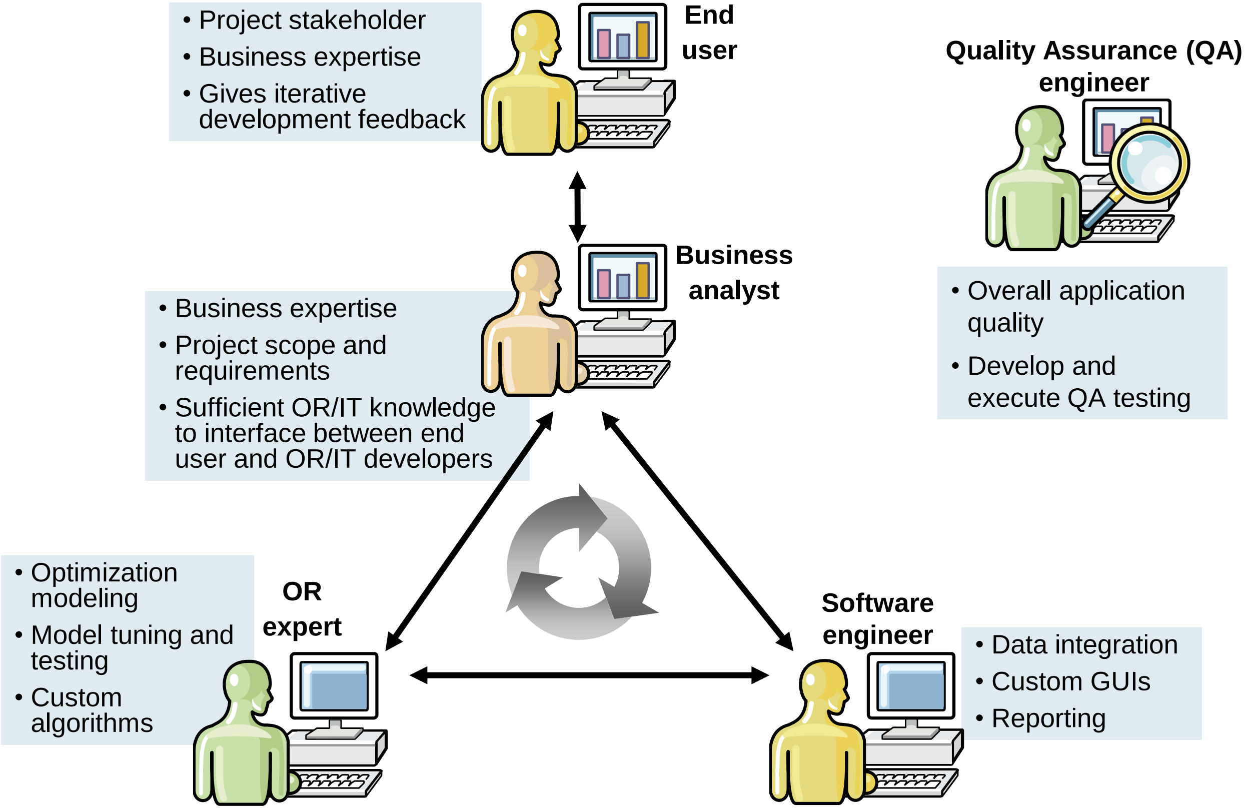
Custom code for business needs beyond built-in functionality:
- *Integration*
- *Extensions*
- *Visualization*

Packaged solutions

Supply Chain Applications

network,
production,
inventory,
transportation

Roles in an optimization-based application



- Project stakeholder
- Business expertise
- Gives iterative development feedback

End user

Quality Assurance (QA) engineer

- Overall application quality
- Develop and execute QA testing

Business analyst

- Business expertise
- Project scope and requirements
- Sufficient OR/IT knowledge to interface between end user and OR/IT developers

OR expert

- Optimization modeling
- Model tuning and testing
- Custom algorithms

Software engineer

- Data integration
- Custom GUIs
- Reporting

The IBM ILOG optimization products

- **Benefits of IBM ILOG for optimization**

- The benefits of using the IBM ILOG products for optimization-based applications include:
  - Intuitive and interactive optimization-based applications that facilitate:
    - “what-if” analysis
    - easy interpretation of solutions to complex optimization problems
    - the power of optimization in the hands of non-experts
    - manual manipulation of solutions based on human insight
  - Reduced development time and reduced risk via tight integration of different components
  - Rapid prototyping
  - Increased collaboration among business stakeholders, OR experts and IT developers.

Inside CPLEX Studio

- **Learning objective**
- Describe the capabilities and features of IBM® ILOG® CPLEX® Optimization Studio.

Sections

- CPLEX Studio in a nutshell
- Mathematical Programming with OPL
- Constraint Programming with OPL
- CPLEX Studio components
- Optimization Programming Language (OPL)
- IBM ILOG Script
- The OPL APIs
- Demo: A quick look at the IDE

Inside CPLEX Studio

- **CPLEX Studio in a nutshell**
  - OR experts use CPLEX Studio to create and test optimization models consisting of a combination of data, decision variables, objectives and constraints.
  - Models in CPLEX Studio are written with Optimization Programming Language (OPL).
  - OPL includes special constructs for both Mathematical Programming (MP) and Constraint Programming (CP).
  - CPLEX Studio includes access to CPLEX Optimizer for solving both MP and CP problems.
  - The CPLEX Studio Integrated Development Environment (IDE) is similar to the OPL perspective in the ODM Enterprise IDE.

Inside CPLEX Studio

- **Mathematical Programming with OPL**
  - The types of MP problems that can be written in OPL include:
    - Linear Programming (LP)
    - Integer Programming (IP)
    - Mixed-Integer Programming (MIP)
    - Quadratic Programming (QP)
    - Mixed-Integer Quadratic Programming (MIQP)
  - Constraints are written as inequalities, equalities, or logical relationships (and, or, not, imply).
  - MP problems in CPLEX Studio are solved with one of the CPLEX Optimizers, such as Simplex Optimizer, Barrier Optimizer and Network Optimizer.

Inside CPLEX Studio

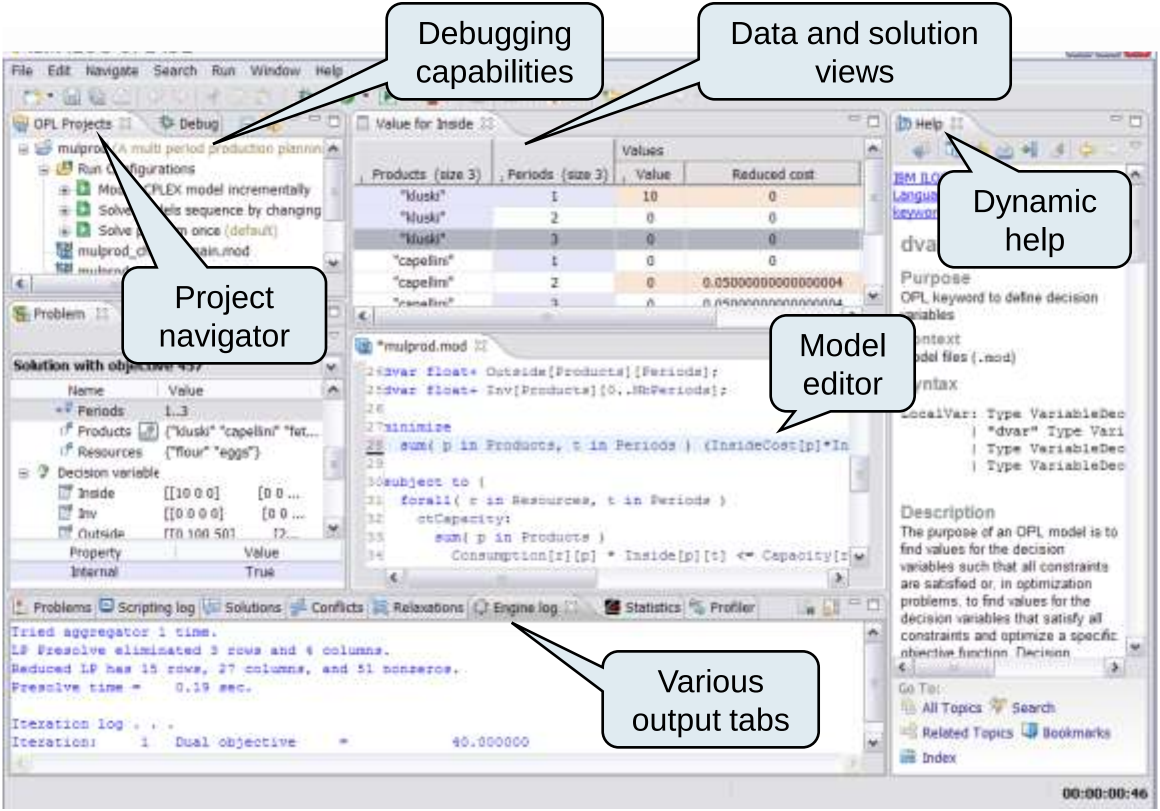
- **Constraint Programming with OPL**
  - CP is typically used for:
    - Detailed scheduling
    - Certain combinatorial problems that are not well-suited for MP
  - OPL includes several keywords and functions for easy modeling of CP constraints.
  - CP problems in CPLEX Studio are solved with IBM ILOG CPLEX CP Optimizer.

Inside CPLEX Studio

- **CPLEX Studio components**

- CPLEX Studio consists of the following components:
  - The IDE (Windows® 64- and 32-bit, Linux® 64- and 32-bit) to write, execute, test, and debug models
  - IBM ILOG Optimization Programming Language (OPL), which allows you to write optimization models in a declarative way
  - IBM ILOG Script, a scripting language for OPL
  - Application Programming Interfaces (APIs) to embed models into standalone applications

The IDE for OPL modeling



Debugging capabilities

Data and solution views

Dynamic help

Project navigator

Model editor

Various output tabs

Solution with objective 457

Name	Value	
Periods	1..3	
Products	{"kluski" "capellini" "fet...	
Resources	{"flour" "eggs"}	
Decision variable		
Inside	[[10 0 0]	[0 0 ...
Inv	[[0 0 0 0]	[0 0 ...
Outside	[[0 100 50]	[2...

Property	Value
Internal	True

		Values	
Products (size 3)	Periods (size 3)	Value	Reduced cost
"kluski"	1	10	0
"kluski"	2	0	0
"kluski"	3	0	0
"capellini"	1	0	0
"capellini"	2	0	0.050000000000000004
"capellini"	3	0	0.050000000000000004

```
dvar float+ Outside[Products][Periods];
dvar float+ Inv[Products][0..NbPeriods];

minimize
  sum( p in Products, t in Periods ) (InsideCost[p]*In

subject to {
  forall( r in Resources, t in Periods )
    ctCapacity:
      sum( p in Products )
        Consumption[r][p] * Inside[p][t] <= Capacity[r
```

```
Tried aggregator 1 time.
LP Presolve eliminated 3 rows and 4 columns.
Reduced LP has 15 rows, 27 columns, and 51 nonzeros.
Presolve time =    0.19 sec.

Iteration log . . .
Iteration:     1   Dual objective     =          40.000000
```

IBM ILOG
Language
keyword

dvar

Purpose

OPL keyword to define decision variables

Context

Model files (.mod)

Syntax

```
LocalVar: Type VariableDec
        | "dvar" Type Vari
        | Type VariableDec
        | Type VariableDec
```

Description

The purpose of an OPL model is to find values for the decision variables such that all constraints are satisfied or, in optimization problems, to find values for the decision variables that satisfy all constraints and optimize a specific objective function. Decision

Go To:

All Topics Search

Related Topics Bookmarks

Index

Inside CPLEX Studio

- **Optimization Programming Language (OPL)**

- This high-level language provides:
  - A compact declarative language for Mathematical Programming (MP) and Constraint Programming (CP)
  - Advanced data types
  - Connections to relational databases and Excel spreadsheets
  - The ability to call external Java^{TM} functions from inside CPLEX Studio

Inside CPLEX Studio

- **IBM ILOG Script**

- This scripting language is used for:
  - Preprocessing of data and engine parameters
  - Postprocessing of solutions, or data, or both
  - Flow control, for example, in decomposition or incremental modification of the model

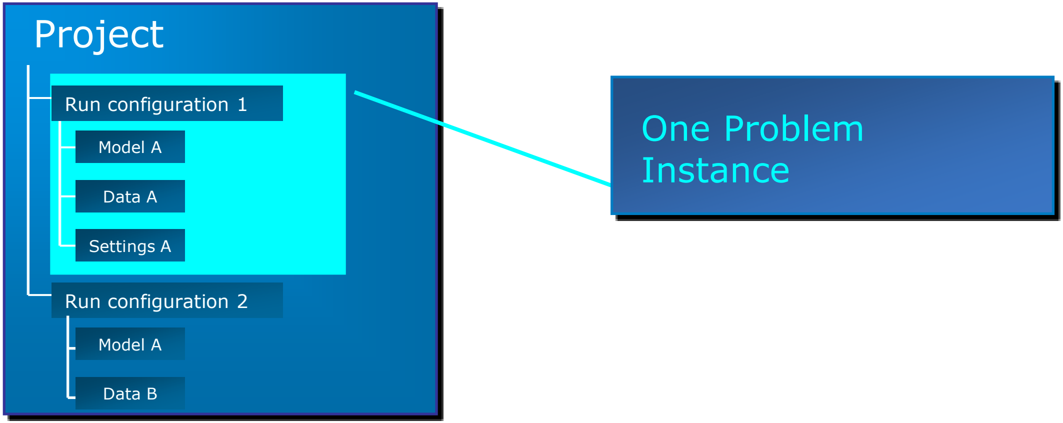
Inside CPLEX Studio

- **The OPL APIs**

- The APIs allow you to embed OPL models and scripts into applications using:
  - C++
  - Microsoft® .NET
    - Visual Basic.NET, C#, etc.
    - Microsoft Office 2003 or higher via Visual Studio Tools for Office
  - Java™
  - ASP.NET, JSP

- The latter two are offered indirectly through the Java and .NET APIs.

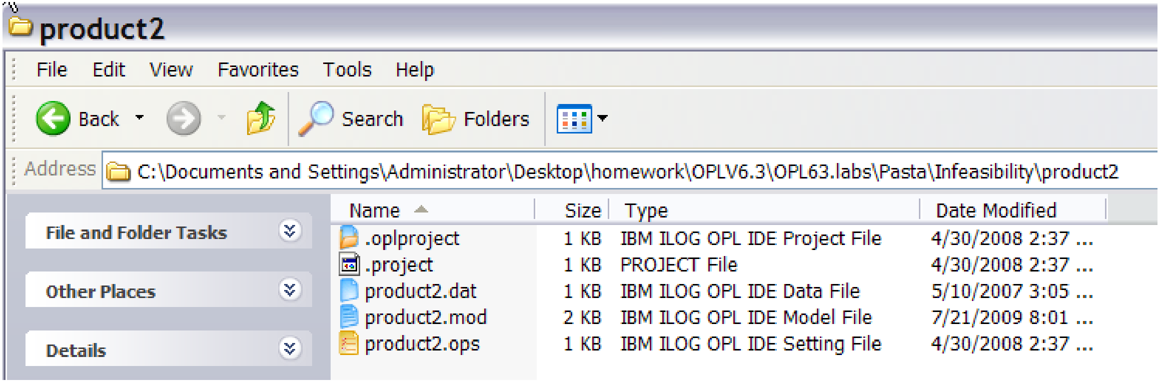
Models, data and projects



- A model is a mathematical representation of a problem.
- In CPLEX Studio, models can exist independent of data.
- A project consists of one or more models, and, optionally, one or more data and settings files.
- A run configuration specifies a problem instance by linking at most one model with one or more data and settings files.

A typical OPL project directory

OPL projects are located inside directories, typically with the same name as the project:



Address C:\Documents and Settings\Administrator\Desktop\homework\OPLV6.3\OPL63.labs\Pasta\Infeasibility\product2

Name	Size	Type	Date Modified
.oplproject	1 KB	IBM ILOG OPL IDE Project File	4/30/2008 2:37 ...
.project	1 KB	PROJECT File	4/30/2008 2:37 ...
product2.dat	1 KB	IBM ILOG OPL IDE Data File	5/10/2007 3:05 ...
product2.mod	2 KB	IBM ILOG OPL IDE Model File	7/21/2009 8:01 ...
product2.ops	1 KB	IBM ILOG OPL IDE Setting File	4/30/2008 2:37 ...

A typical project directory contains:

- The project description files (.project and .oplproject)
- Model files (.mod)
- Data files (.dat)
- Setting files (.ops)

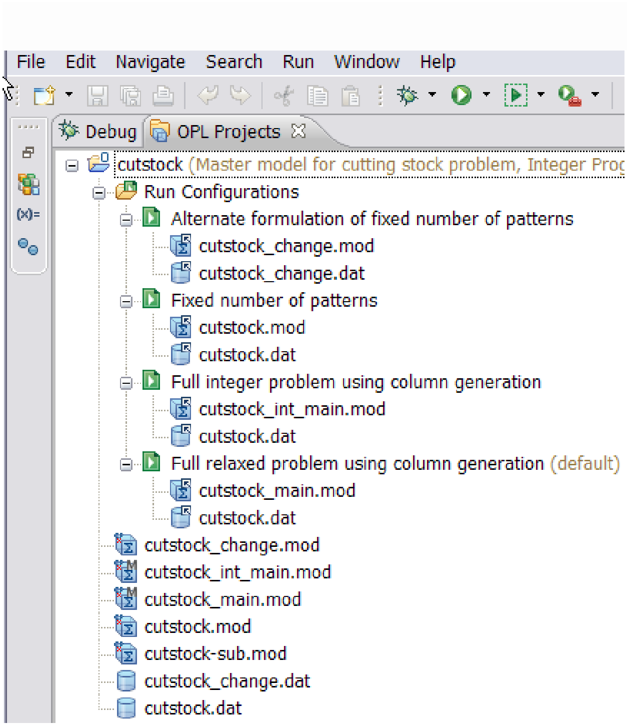
Run configurations

- A run configuration is a combination of exactly one model, and one or more data and settings files in the same project.

- All independent OPL projects require at least one default run configuration.

- Run configurations can be used to:
  - create and test different approaches to solving the same problem
  - create sub-problems for custom solution algorithms

- Unlimited run configurations can be defined within an OPL project.



- cutstock (Master model for cutting stock problem, Integer Prog
  - Run Configurations
    - Alternate formulation of fixed number of patterns
      - cutstock_change.mod
      - cutstock_change.dat
    - Fixed number of patterns
      - cutstock.mod
      - cutstock.dat
    - Full integer problem using column generation
      - cutstock_int_main.mod
      - cutstock.dat
    - Full relaxed problem using column generation (default)
      - cutstock_main.mod
      - cutstock.dat
  - cutstock_change.mod
  - cutstock_int_main.mod
  - cutstock_main.mod
  - cutstock.mod
  - cutstock-sub.mod
  - cutstock_change.dat
  - cutstock.dat

Inside CPLEX Studio

- **Demo: A quick look at the IDE**

Example: a production planning problem

Learning objective

- Learn how to recognize data, decision variables, objectives, and constraints when looking at an OPL model.

Sections

- Problem description
- The decision variables
- The objective function
- The constraints
- Debugging
- Model and data independence

Example: a production planning problem

- **Problem description**

- In this example you'll see how a typical production problem is written, using OPL syntax, in terms of the data, decision variables, objective and constraints:
  - A chemical company produces and sells two products: ammonium gas (NH_3) and ammonium choride (NH_4Cl).
  - Each product has a unit profit associated with it.
  - Each product is manufactured from stocked components (Nitrogen (N), Hydrogen (H), Chlorine (Cl)).
  - The company's objective is to maximize their profit without exceeding the available stock.

- While this example includes some syntax explanations, a more complete explanation of OPL syntax follows in a later lesson.

Generic production planning data

Data	OPL data type	OPL declaration
Set of product names	`string`	`{string} Products = ...;`
Set of component names	`string`	`{string} Components = ...;`
Recipe describing how much of each component is used in each product	`float`	`float usageFactor[Products][Components] = ...;`
Stock on hand of each component	`float`	`float stock[Components] = ...;`
Unit profit for each product	`float`	`float profit[Products] = ...;`

- OPL data is declared in the model file
- Data declarations end with …`;`, unless initialized in the same line
- OPL data is usually initialized in the data file
- Curly brackets, `{}`, denote a set
- Square brackets, `[]`, denote an array

Example: a production planning problem

The decision variables

- The decision variables are the quantities to produce of each product in the set `Products`:
  - `dvar float+ production[Products];`
  - `dvar` is the OPL keyword used to declare decision variables.
  - `float` is the OPL keyword used for real numbers.
  - `+` is added to denote that the quantities are non-negative.
  - `production` is the variable name, and it is defined as an array over the set of `Products`.
- Note that all OPL declarations end with a semicolon.

Example: a production planning problem

- **The objective function**

- The objective is to maximize the total profit over all `Products`:

```
maximize sum(p in Products) profit[p] * production[p];
```

  - `maximize` is the OPL keyword used to declare an objective to be maximized.
  - `sum` is the OPL keyword to compute the summation of a collection of expressions, in this case to sum up the profit over all products.
  - Note the use of normal parentheses, as in `(p in Products)`, to denote the selection to sum over.
  - `p` is an index used to access each element of the set of `Products`.

Example: a production planning problem

- **The constraints**

- The constraints are that the amount of each component used should not exceeded the available stock:

-
  ```
  subject to{
        forall(c in Components)
               sum(p in Products) usageFactor[p,c] * production[p]
  <= stock[c];}
  ```

  - Constraints are written inside a block starting with `subject to {`, and ending with `}`.
  - `forall` is the OPL keyword used when expressions are similar, except for their indices.
  - Only one constraint is written for all components, because the constraints only differ according to the product or component.
  - `c` is an index used to access each element of the set of `Components`.

Simple production planning model

```
{string} Products = ...;
{string} Components = ...;

float demand[Products][Components] = ...;
float profit[Products] = ...;
float stock[Components] = ...;
```

Data Declarations

```
dvar float+ production[Products];
```

Decision Variables

```
maximize
   sum (p in Products) profit[p] * production[p];
```

Objective Function

```
subject to {
 forall (c in Components)
  sum (p in Products)
   usageFactor[p, c] * production[p]
       <= stock[c];
}
```

Constraints

Example: a production planning problem

- **Debugging**
  - The IDE provides debugging facilities to trap errors such as syntax and runtime errors.
  - Errors are listed in the *Problems* Output tab, and are also indicated with an icon in the text editor.
  - Advanced debugging features, such as breakpoints in IBM® ILOG® Script blocks, are available – for more information see the Appendix and the online help.

Example: a production planning problem

- **Model and data independence**
  - Model and data are usually independent entities in the IDE.
  - In the exercise that follows, you'll combine the existing chemicals production model with different data to solve a jewelry production problem instead.

Work with the production planning problem in CPLEX Studio

- **Lab: Gas production planning**

- Go to the *Gas production planning* workshop and complete all the steps.

Checkpoint

- **True or false?**
  1. CPLEX® Studio and ODM Enterprise are used to develop packaged optimization-based applications.
  2. CPLEX Optimizer includes both MP and CP optimization engines.
  3. CPLEX Studio is embedded in ODM Enterprise for rapid prototyping.
  4. A run configuration in CPLEX Studio can contain more than one model.

Unit summary

- Having completed this lesson, you should be able to:
  - Describe how the IBM ILOG optimization products work together to address optimization problems.
  - List the optimization techniques and associated engines available through CPLEX Studio.
  - Describe the basic functionality of the IDE.
  - Describe what an OPL project is and how OPL projects are organized and managed.
  - Point out the basic model elements, namely data, decision variables, objectives and constraints, when looking at an OPL model.

Working with OPL

Unit objectives

- At the end of this lesson you should be able to:
  - Describe the structure of an OPL model
  - Describe the data types, data structures, and types of variables available in OPL
  - Describe some of the constraints available in OPL
  - Understand the concept of sparsity
  - Write a simple model by using OPL syntax

Working with OPL

- OPL is a declarative language – you do not need to write procedures when constructing a model.

- To build and solve an optimization model:
  - declare the data elements, decision variables, objectives and constraints
  - call the appropriate solver engine to solve the model

- You can optionally add procedures for pre- and postprocessing, as well as flow control, using IBM ILOG Script.

Working with OPL

- OPL model structure
- OPL data files
- OPL data structures
- Exercise: A telephone production problem
- Combining OPL data structures
- Sparsity and slicing
- Exercise: A pasta production model
- Checkpoint

OPL model structure

Learning objective

- Learn how an OPL model is structured.

Sections

- The choice of solver
- Data
- Decision variables
- Decision variable expressions
- Objective function
- Constraints

OPL model structure

- An OPL model is typically structured in the following sequence (some of these are optional):
  - The choice of solver engine
  - Data declarations
  - Decision variables
  - Objective function
  - Constraints

OPL model structure

- An OPL model file can also contain IBM® ILOG® Script statements:
  - before the objective for preprocessing
  - after the constraints for postprocessing
  - before the objective or after the constraints for flow control

- This lesson focuses on pure OPL only, and does not cover IBM ILOG Script.

OPL model structure

- **The choice of solver**

- By default, OPL models are assumed to be Mathematical Programming (MP) problems, and solved with one of the MP optimizers, such as CPLEX® Simplex Optimizer.

- The user can specify that CP Optimizer should be used, by starting the model with the following text:
  - `using CP;`

- CP Optimizer is appropriate for the following types of Constraint Programming (CP) problems:
  - Detailed scheduling problems
  - Certain combinatorial problems not well-suited for MP

OPL model structure

- **Data**

- When declaring data, you need to decide:
  - The name for the data item
  - The data type:
    - Integer (OPL keyword `int`)
    - Real (OPL keyword `float`)
    - String (OPL keyword `string`)
  - The data structure, which can be a scalar, a range, a set, an array, or a tuple.

OPL model structure

- An example of a simple OPL data declaration is:

  - `float unitProfit = ...;`

  - `float` is the OPL keyword used for real (fractional) data or decision variables
  - `unitProfit` is the name of the data item
  - in this case, the data structure is a scalar – more complex data structures such as sets, arrays and tuples are indicated by a special syntax, which you'll learn about later.
  - all data declarations end with `...;`, unless the data is initialized in the same line.

OPL model structure

- **Decision variables**

- When declaring decision variables, you need to decide:
  - The name of the variable
  - The variable type:
    - Integer (OPL keyword `int`)
    - Real (OPL keyword `float`, for MP only)
    - Boolean (OPL keyword `boolean`)
    - Interval (OPL keyword `interval`, for CP only)
    - Sequence (OPL keyword `sequence`, for CP only)
  - The data structure, which can be a scalar, a range, a set, an array, or a tuple
  - Optionally, the domain, which is the set of possible values the variable can take

OPL model structure

- An example of a simple OPL decision variable declaration is:
  - `dvar float+ production in 0..maxCapacity;`
  - `dvar` is the OPL keyword used to declare decision variables
  - the (optional) `+` sign is OPL syntax that indicates this variable can take only non-negative values
  - `production` is the name of the decision variable
  - `in 0..maxCapacity` defines the domain of the variable to include only values between 0 and the maximum capacity (`maxCapacity` being another data item)

OPL model structure

- **Decision variable expressions**

- OPL decision variable expressions can be used to write more complex expressions in a compact way.

- An example of a simple OPL decision expression is:

```
dexpr float+ profit =  production * unitProfit;
```

  - `dexpr` is the OPL keyword used to declare decision expressions
  - `profit` is the name of this particular decision expression
  - this expression defines `profit` to equal the `production` decision variable multiplied by the `unitProfit` associated with each unit produced.

OPL model structure

- **Objective function**

- When defining an objective function, you need to decide whether it's a maximization or minimization, and the expression to optimize.

- An example of a simple OPL objective declaration is:

  - `maximize profit;`

  - `maximize` is the OPL keyword used for maximization problems. `minimize` is used for minimization problems.
  - `profit` is the decision expression to be maximized. The objective expression can vary in complexity, depending on the problem.

- Alternatively, one could omit the `profit` decision expression and write this objective as follows:

  - `maximize production*unitProfit;`

OPL model structure

- **Constraints**

- Constraints in OPL are written in a block starting with `subject to {` and ending with `}`.

- An example of a simple OPL constraint block containing a single constraint is:

-
```
subject to {
productionConstraint: production <= capacity;
}
```

  - `productionConstraint` is the name, or constraint label, of this constraint. Labeling constraints is optional.
  - this constraint states that the `production` quantity must be less than or equal to the `capacity`

Expressing variable bounds as a constraint

- Upper and lower bounds on a decision variable (or an expression containing a decision variable) *where the bounds themselves involve decision variables* must be expressed separately as two OPL constraints.

- For example, the following is not allowed:

```
dvar int x in 0..5;
dvar int y;
dvar int z;
minimize x;
subject to {
  z <= y <= x;
}
```

- Instead, you must write

```
dvar int x in 0..5;
dvar int y;
minimize x;
subject to {
  z <= y;
  y <= x;
}
```

- However, `z <= y <= x` is allowed in the case where `z` and `x` are data items.

OPL data files

Learning objective

- Learn what the contents of an OPL .dat file are, and how to write a simple data initialization statement.

Sections

- What's in a data file?
- Basic data initialization

OPL data files

- **What's in a data file?**

  - Data (.dat) files facilitate separation of the model and the data, thus allowing the use of several different data instances with the same model.
  - Data files may include:
    - Data initialization
    - Statements to connect to spreadsheets and databases
    - Statements to initialize data by reading values from spreadsheets and databases
    - Statements to write solution values to spreadsheets and databases

OPL data files

- Some of the characteristics of data files are:
  - Data initialization directly in a `.dat` file cannot contain computations or expressions of any kind, only raw data.
  - Data types are not required in the data file, because these have already been defined in the model file. Simply use the name of the data item and assign a value to it.
  - It is possible to use IBM® ILOG® Script in a `.dat` file for custom reading and formatting of data.
  - The data file and model file within the same project need not have the same name.
  - You can use several different data files with a model file within the same project.
  - You can concatenate multiple data files in one project, or run different data sets on the same model in the same project by using run configurations.

OPL data files

- **Basic data initialization**
- Consider the following OPL data declaration:
  - `float unitProfit = ...;`
- This data can be initialized in the model file by replacing the declaration with:
  - `float unitProfit = 2.5;`
- or keep the declaration as before in the model file and initialize the data item in data file:
  - `unitProfit = 2.5;`
- The data type is not used when initializing data in the data file. It's generally better to initialize data in the data file, because it keeps the model generic with respect to data.

OPL data structures

Learning objective

- Learn what the basic OPL data structures are, and how to declare them. Learn how to use indices.

Sections

- The OPL data structures
- Ranges
- Sets
- Using indices
- Initializing sets
- Arrays
- Initializing arrays
- Array indices
- Multidimensional arrays
- Arrays of decision expressions
- Tuples
- Tuple keys
- Initializing tuples

OPL data structures

- **The OPL data structures**

  - When designing an optimization model, it's essential to choose the correct data structures for declaring data.
  - In this topic you'll learn about the basic use of the OPL data structures:
    - range
    - set
    - array
    - tuple
  - Later in this lesson you'll learn how to combine these data structures for more versatility.

OPL data structures

- **Ranges**

  – Integer ranges are often used in arrays and decision variable declarations, as well as in aggregate operators, queries, and quantifiers.

  – An integer range is specified with the keyword `range` and lower and upper bounds:

  ```
  range Rows = 1..10;
  ```

  – The lower and upper bounds can also be given by expressions:

  ```
  int n = 8;
  range Rows = n+1..2*n+1;
  ```

OPL data structures

- An integer range is typically used:
  - as an array index in an array declaration

    ```
    range R = 1..100;
    int A[R]; // A is an array of 100 integers
    ```

  - as an iteration range

    ```
    range R = 1..100;
    forall(i in R) {
       //element of a loop
    }
    ```

  - as the domain of an integer decision variable

    ```
    dvar int i in R;
    ```

- You'll learn more about arrays and the `forall` statement later in this lesson.

OPL data structures

- **Sets**
  - OPL sets are non-indexed collections of unique elements.
  - OPL supports sets of arbitrary types: If `T` is a data type, then `{T}`, or `setof(T)`, denotes a set with elements of type T.
  - For example, to declare a set called `myIntegerSet` containing elements of type `int`:

```
{int} myIntegerSet = ...;
setOf(int) myIntegerSet = ...;
```

OPL data structures

- Sets may be:
  - Ordered (default):
    - Set elements are considered in the order in which they have been created.
    - Functions and operations applied to ordered sets preserve the order.
    - For example, `ordered {int} myIntSet = {3,2,5};`
  - Sorted:
    - Set elements are arranged in their natural, ascending order. For strings, this is the lexicographic order.
    - For example, iterating over a set declared as `sorted {int} myIntSet = {3,2,5};` will iterate in the order {2, 3, 5}
  - Reversed: Set elements are arranged in descending order.

OPL data structures

- A set is often used as an array index in an array declaration.

- In this example, production is a decision variable array indexed over the set of Products:

- `{string} Products ={”product1”,”product2”,”product3”};`
- `dvar float production[Products];`

OPL data structures

- **Using indices**

  – Use indices to concisely write similar expressions that only differ in the item(s) the expression is declared for.

  – For example, use the set `Products` as in index to declare the variable `production`:

  ```
  dvar float production[Products];
  ```

  – Without an index, the production variable would have to be defined for each product individually:

  ```
  dvar float production_product1;
  dvar float production_product2;
  dvar float production product3;
  ```

Using indices - example

- Declare a set or range to use as an index:

```
{int} ProductNumbers = {1, 2, 3};
```

- Use the index to declare arrays:

```
float capacity[ProductNumbers] = ...;
dvar float production[ProductNumbers];
```

- Define a parameter, `p`, to refer to each index element in the constraint declaration:

```
forall(p in ProductNumbers)
      production[p] <= capacity[p];
```

- Filter the index by using a filtering condition:

```
forall(p in ProductNumbers : p <= 2)
      production[p] <= capacity[p];
```

OPL data structures

- Several indices can be combined in a comma-separated list to produce more compact statements. For instance:
  - `int s = sum(i,j in 1..n: i < j) i*j;`
- is equivalent to
  - `int s = sum(i in 1..n) sum(j in 1..n: i < j) i*j;`
- which is less readable.

OPL data structures

- The quantifiers, `forall` and `all`, work with indices to simplify model declarations:
  - `forall`
    - used in MP and CP models
    - generates one constraint for each instance of the indexed entity, for example:

      ```
      forall (p in ProductNumbers)
                             production[p] <= capacity[p];
      ```
  - `all`
    - used in CP models
    - filters a set of objects to be used as arguments for certain CP constructs

OPL data structures

- Aggregator operators (for example `sum`, `prod`, `min` or `max`) work with indices to construct integer and float expressions.

  - An example of using the aggregate operator `sum`:

    ```
    maximize sum (p in Products) profit[p] * production[p];
    ```

  - An example of using a quantifier and aggregator together:

    ```
    forall (p in products)
                sales[p] == sum (l in locations) locSales[l][p];
    ```

OPL data structures

- **Initializing sets**

- Sets can be initialized in the following ways:

  – Internally in the model file:
  `{int} myIntegerSet = {1, 3, 5, 7};`

  – Externally in the data file:
  `myIntegerSet = {1, 3, 5, 7};`

  – In a generic way using another range or set, for example use `IntegerSet` to create `anotherIntegerSet` with elements 1 and 3:
  `{int} anotherIntegerSet = {i | i in myIntegerSet : i <= 3};`

OPL data structures

- Some other examples of set initialization are:
  - Using the `asSet` keyword to convert a range to a set:
    ```
    {int} mySet = asSet(1..10);
    ```
  - Using the modulus (`mod`) operator to create a set of every third number between 1 and 10:
    ```
    {int} mySet = {i | i in 1..10 : i mod 3 == 1};
    ```
    This is equivalent to `{int} mySet = {1, 4, 7, 10};`.
  - Using the `union` operator on two other sets to create a new set:
    ```
    {int} mySet3 = mySet1 union mySet2;
    ```

OPL data structures

- **Arrays**

  - OPL arrays can be of the basic data types or more complex data structures:
    - `int` , `float` or `string`
    - Sets
    - Tuples

  - OPL arrays can be multidimensional.

  - This topic focusses on arrays of the basic data types. You'll learn more about arrays of sets or arrays of tuples in a later topic.

OPL data structures

- **Initializing arrays**

- Arrays can be initialized in the following ways:

  – In the model file:

  ```
  int myIntegerArray[1..4] = [1, 3, 5, 7];
  ```

  – In the data file:

  ```
  myIntegerArray = [1, 3, 5, 7];
  ```

  – In a generic way (known as a generic array):

  ```
  int anotherIntegerArray[i in 1..10] = [i+1];
  ```

  – In an IBM® ILOG® Script execute block:

  ```
  range R = 1..8;
  int a[R];
  execute {for(var i in R) {a[i] = i + 1;}}
  ```

OPL data structures

- **Array indices**

- An array index can be:

  – a range:

  ```
  float unitProfit[1..4] = ...;
  ```

  – a set:

  ```
  float unitProfit[Products] = ...;
  ```

  – defined by a generic expression (known as a generic indexed arrays):

  ```
  int myArray[1..10] = [ n-1 : n | n in 90..99 ];
  ```

OPL data structures

- **Multidimensional arrays**
  - A 2-dimensional array declaration and initialization:

    ```
    int my2DArray[1..2][1..3] = [[5, 2],[4, 4],[3, 6]];
    ```

  - The declaration and initialization can be done separately with declaration in the model file:

    ```
    int my2DArray[1..2][1..3] = ...;
    ```

    and initialization in the data file:

    ```
    my2DArray = [ [5, 2], [4, 4], [3, 6] ];
    ```

  - A 2-dimensional array with different types of indices:

    ```
    int numberOfWorkers[Days][1..3] = ...;
    ```

  - A generic multidimensional array:

    ```
    int m[i in 1..10][j in 0..10] = 10*i +j;
    ```

OPL data structures

- **Arrays of decision expressions**

  – Use arrays together with the keyword `dexpr` to create more compact, efficient models.

  – For example,

  ```
  dexpr int surplus[i in Stock] = stock[i] – demand[i]
  ```

  – The surplus array is handled very efficiently because the definition is kept as a reusable object – it is not necessary to store every value of the expression for every index value.

OPL data structures

- **Tuples**

  - A tuple is analogous to a row in a database table.

  - For example, the tuple:

    ```
    tuple productData{
       string product;
       int timePeriod;
       float demand;
    }
    ```

    is analogous to a row in a database table with three columns: `product`, `timePeriod`, and `demand`.

  - A set of tuples is analogous to a database table, for example the set `ProductData`

    ```
    {productData} ProductData = ...;
    ```

    is analogous to a database table containing product data.

OPL data structures

- **Tuple keys**
  - As in database systems, tuple structures can contain one or more keys.
  - Tuple keys enable you to access data using a set of unique identifiers, for example:

```
tuple nurse {
   key string name;
   int seniority;
   int qualification;
   int payRate;
}
```

OPL data structures

- **Initializing tuples**
  - Tuples are initialized by listing the values of the various fields, within the delimiters "<" and ">".
  - For example:

    ```
    tuple Point {
         int x;
         int y;
      };
    ```

    A tuple, `p`, of type `Point`, can be initialized:

    - In the model file:

      ```
      Point p = <2,3>;
      ```

    - In the data file:

      ```
      p = <2,3>;
      ```

  - Access a tuple field by suffixing the tuple name with a dot and the field name:

    ```
    int x = p.x;
    ```

Summary of OPL data structures

Data structure	Example	Example description
Range	`range weeks = 1..52;`	A range of integers between 1 and 52
	`range float x = 1.0..73.8;`	A range of real numbers between 1.0 and 73.8
Set	`{int} myIntegerSet = {1,2,3};`	A set of integers
	`setOf(string) Products = {“product1”,”product2”,”product3”};`	A set of strings
Array	`dvar float production[Products];`	A decision variable array of type float indexed over the set of Products

Tuple	`tuple productData {`
	`string name;`
	`int timePeriod;`
	`float demand;`
	`};`
	`{productData} ProductData = …;`
		A tuple called productData with fields name, timePeriod and demand. A set ProductData of type productData.

Exercise: A telephone production problem

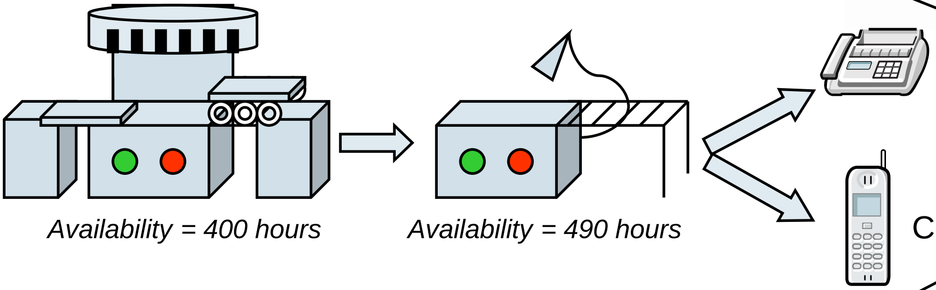
Learning objective

- Write a simple optimization model using OPL.

Telephone production - problem description

A company produces two types of phones. Determine the **optimal amount** of each type **to produce**, so as to **maximize profit without violating production constraints**.

Profit = $12/unit
Assembly time = 12 minutes/unit
Painting time = 30 minutes/unit
Minimum production = 100 units



Assembly machine

Painting machine

Desk phone

Cellular phone

Availability = 400 hours

Availability = 490 hours

Profit = $20/unit
Assembly time = 24 minutes/unit
Painting time = 24 minutes/unit
Minimum production = 100 units

Telephone production – descriptive model

Decision variables	Objective function	Constraints
Quantity of each item to produce: ▪ DeskProduction ▪ CellProduction	**Maximize profit:** 12 * DeskProduction + 20 * CellProduction	**Subject to:** ▪ DeskProduction >= 100 ▪ CellProduction >= 100 ▪ 0.2 * DeskProduction + 0.4 * CellProduction <= 400 (time limit in hours on assembly machine) ▪ 0.5 * DeskProduction + 0.4 * CellProduction <= 490 (time limit in hours on painting machine)

In this descriptive model:

- DeskProduction is a decision variable that represents the number of desk phones to manufacture
- CellProduction is a decision variable that represents the number of cell phones to manufacture
- 0.2 * DeskProduction and 0.4 * CellProduction represent the time, converted into hours, necessary to assemble each type of phone.
- 0.5 * DeskProduction and 0.4 * CellProduction represent the time, converted into hours, necessary to paint each type of phone.

Exercise: A telephone production problem

- **Lab**
  Write the telephone production model in OPL

- In this exercise you'll create an OPL project, model file, data file, and run configuration to model and run the telephone production problem.

- Complete the first two steps of the *Telephone production* workshop:
  - Write the model
  - Separate the data from the model

Combining OPL data structures

Learning objective

- Learn how to combine different data structures. Understand especially how to use sets of tuples.

Sections

- Sets inside tuples
- Arrays inside tuples
- Tuples inside tuples
- Sets of tuples
- Arrays of tuples
- Arrays of sets
- Converting an array to a tuple set
- Using sets to instantiate arrays

Combining OPL data structures

- In this topic, you'll learn how to combine the different data structures available in OPL for more versatile use. Specifically, you'll learn how to use:
  - Sets inside tuples
  - Arrays inside tuples
  - Tuples inside tuples
  - Sets of tuples
  - Arrays of tuples
  - Arrays of sets

- You'll also learn about some useful data structure manipulations, namely:
  - Converting an array to a tuple set
  - Using sets to instantiate arrays

Combining OPL data structures

- **Sets inside tuples**

- Sets can be used as tuple elements, for example:

-
  ```
  tuple Members {
    int memberNumber;
    {string} memberInterests;
  };
  ```

- This declares a tuple type `Members` containing a tuple element for the `memberNumber`, as well as a tuple element for the set of the member's interests.

Combining OPL data structures

- **Arrays inside tuples**

- One-dimensional arrays can be used as tuple elements, for example:

-
  ```
  tuple ProductData {
       int productId;
       float insideCost;
       float outsideCost;
       float consumption[Resources];
  }
  ```

- This code declares a tuple type `ProductData` consisting of 4 elements:
  - `productId` of type `int`
  - `insideCost` and `outsideCost` of type `float`
  - `consumption[Resources]` - an array of type `float`

- You can declare and initialize a tuple of this type as follows:
- `ProductData p = <1, 0.52, 0.17, [0.7, 0.9]>;`

Combining OPL data structures

- **Tuples inside tuples**

- A tuple can have one or more tuples as elements. For example, consider a case where you'd like to associate products with production plants. Then you may have an additional tuple type for the production plant data:

-
  ```
  tuple PlantData {
      int plantId;
      float capacity;
   }
  ```
- and associate the products with the production plants in a tuple type called `ProductAtPlant`:

-
  ```
  tuple ProductAtPlant {
      ProductData product;
      PlantData plant;
   }
  ```

Combining OPL data structures

Some limitations apply to the contents of tuples:

- Multidimensional arrays are not allowed.
- Arrays of strings, arrays of tuples and arrays of tuple sets are not allowed.
- Sets of tuples are not allowed.

Combining OPL data structures

- **Sets of tuples**

- Once a tuple type T has been declared, you can declare sets of tuples of type T. Consider again the `ProductData` tuple type defined earlier:
-
  ```
  tuple ProductData {
       int productId;
       float insideCost;
       float outsideCost;
       float consumption[Resources];
  }
  ```
- Using this tuple type, you can declare and initialize a set of tuples as follows:
-
  ```
  {ProductData} pDataSet = {<1, 0.52, 0.17, [0.7, 0.9]>, <2, 0.21,
  0.44, [0.6, 0.3]>};
  ```

- The set `pDataSet` contains two tuples of type `ProductData`.

Combining OPL data structures

- **Arrays of tuples**

- You can declare and initialize an array of tuples of type `ProductData` as follows:
- `ProductData pDataArray[i in 1..5] = <i, 2*i, 2.5*i, [0.7, 0.9]>;`

- This example declares an array of 5 tuples of type `ProductData`, where the values of `insideCost` and `outsideCost` change, depending on the value of the `productId` (which in this particular case corresponds to the array index).

Combining OPL data structures

- **Arrays of sets**

- OPL supports arrays of sets, for example:

- `{int} myIntArray[1..2] = [{1,2},{3,4}];`

- It is also possible to initialize an array of sets in a generic way. For example, the declaration:

- `{int} a[i in 3..4] = {e | e in 1..10: e mod i == 0};`

- instantiates `a[3]` to {3,6,9} and `a[4]` to {4,8}.

Combining OPL data structures

- **Converting an array to a tuple set**

- It is possible to convert array data to a tuple set by using a generic set initialization.

- For example, this 2D Boolean array, `edges`, describes the edges of a graph:

-
  ```
  {string} Nodes ...;
  ```

-
  ```
  int edges[Nodes][Nodes] = ...;
  ```

- It can be transformed into a set of edges, with each edge represented by a tuple:

-
  ```
  tuple Edge {
        Nodes o;
        Nodes d;
  }
  ```

-
  ```
  {Edge} setEdges ={<o,d> | o,d in Nodes:edges[o][d]==1};
  ```

Combining OPL data structures

- **Using sets to instantiate arrays**

- Sometimes it is useful to use a set in order to instantiate an array in a generic way. The following code extract shows how this can be done:

- `{string} Gasolines = ...;`
- `{string} Oils = ...;`
- `{gasType} GasData = ...;`
- `{oilType} OilData = ...;`
- `gasType Gas[Gasolines] = [ g.name : g | g in GasData ];`
- `oilType Oil[Oils] = [ o.name : o | o in OilData ];`

Sparsity and slicing

Learning objective

- Learn about sparse data structures using tuple sets, implicit and explicit slicing, and performance monitoring.

Sections

- Model sparsity vs. data sparsity
- Slicing
- Example: Use tuple sets to instantiate only valid data
- Example: Create a sparse array by indexing on the tuple set
- Example: Use slicing for efficient constraint declaration
- Explicit versus implicit slicing

Sparsity and slicing

Sparsity

Sparsity refers to the characteristic where a **matrix** representation of an optimization model has **many elements** that have **0 (zero)** as a value.
The matrix on the left would be considered sparse, while the matrix on the right would be considered non-sparse or “dense”.

Sparse (many 0’s) Dense (few 0’s)

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 4 \end{bmatrix} \quad \text{vs.} \quad \begin{bmatrix} 0 & 1 & 2 & 1 \\ 2 & 1 & 0 & 3 \\ 1 & 1 & 3 & 1 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

Sparsity and slicing

- **Model sparsity vs. data sparsity**
  - Model sparsity refers to a sparse representation of the model variables and constraints.
  - Data sparsity refers to a sparse representation of the data input to the model and solver engine.
  - Sparse models and data should be exploited by allowing only the essential variables, constraints, and data.
  - Redundant data, variables, and constraints unnecessarily take up memory and solver resources.
  - Slicing is a technique used to streamline sparse models and data.

Example of model sparsity (part 1)

Consider a problem where three products are shipped between three suppliers and two customers. Let $x_{p,s,c}$ be the amount of product p shipped between supplier, s and customer, c.

The total shipments to a customer should exceed that customer's demand. This would lead to a total of 6 constraints:

$$x_{111} + x_{121} + x_{131} >= demand_{11}$$
$$x_{211} + x_{221} + x_{231} >= demand_{21}$$
$$x_{311} + x_{321} + x_{331} >= demand_{31}$$
$$x_{112} + x_{122} + x_{132} >= demand_{12}$$
$$x_{212} + x_{222} + x_{232} >= demand_{22}$$
$$x_{312} + x_{322} + x_{332} >= demand_{32}$$

When coding these constraints using software, one would not write out each constraint, but instead use a compact representation:

$$sum(s, x_{p,s,c}) >= demand_{p,c} \qquad for\,all\; p, c$$

Example of model sparsity (part 2)

Now imagine that customer 1 is only interested in products 1 and 2, and customer 2 is only interested in product 3. The data has become sparse and the redundant constraints should be removed so that the constraint set becomes:

$$x_{111} + x_{121} + x_{131} >= demand_{11}$$
$$x_{211} + x_{221} + x_{231} >= demand_{21}$$
$$x_{312} + x_{322} + x_{332} >= demand_{32}$$

The original compact representation would give the original set of 6 constraints instead of 3, because the constraint was defined for all products and customers.

To properly express the sparsity of the model, define a new object to represent all valid combinations of products and customers (*validCombinations(p,c)*), and define the constraints for valid combinations only:

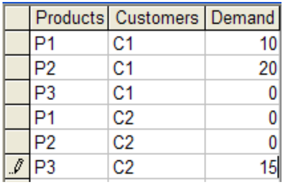
$$sum(s, x_{p,s,c}) >= demand_{p,c} \qquad for\, all\, validCombinations(p,c)$$

The result is a sparse model representation.

Sparsity and slicing

Example of data sparsity

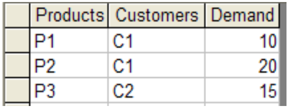
Consider the data representation of the same set of constraints. It’s possible to import demand data over all products and customers, demand(p)(c), from this table:



Products	Customers	Demand
P1	C1	10
P2	C1	20
P3	C1	0
P1	C2	0
P2	C2	0
P3	C2	15

Here, the data for the invalid combinations unnecessarily occupies database space, and adds unnecessary time and memory requirements to data import and model setup.

Instead, the more compact data set of only valid combinations should be used:



Products	Customers	Demand
P1	C1	10
P2	C1	20
P3	C2	15

Sparsity and slicing

- **Slicing**
  - The technique of using predefined sets of valid combinations (such as the set of `validCombinations(p,c)` in the example) to create a sparse model or data representation, is referred to as “slicing”.
  - Using slicing to create sparse models removes redundant constraints, thereby freeing up memory, and leading to quicker solutions.
  - The tuple data structure is fundamental in creating sparse models and data.

Sparsity and slicing

- **Example: Use tuple sets to instantiate only valid data**

  – Consider a tuple `workTask`:

  ```
  tuple workTask {
     string workerID;
     string taskID;
  };
  ```

  – The tuple set should include only valid pairs of workers and tasks:

  ```
  {workTask} WorkerTasks = ...;
  ```

  – An example of a data set with only valid combinations is:

  ```
  WorkerTasks = {
  <"GWashington" "Plumbing">
  <"GWashington" "Carpentry">
  <"NBonaparte" "Carpentry">
  <"NBonaparte" "Painting">
  <"WChurchill" "Plumbing">
  <"WChurchill" "Painting">
  };
  ```

Sparsity and slicing

- **Example: Create a sparse array by indexing on the tuple set**
  - Consider a boolean decision variable `x`, that takes a value of 1 if a worker is assigned to a task, and 0 otherwise.
  - A naïve declaration is:
    ```
    dvar boolean x[workerID] [taskID];
    ```
    This declares redundant variables when some combinations of workers and tasks are irrelevant.
  - A better declaration uses only the possible combinations as defined in the `WorkerTasks` set:
    ```
    dvar boolean x[WorkerTasks];
    ```

Sparsity and slicing

- **Example: Use slicing for efficient constraint declaration**
  - Consider constraints to assign exactly one worker to each task, and vice versa.
  - A naïve declaration is:

```
forall (t in Tasks)
   assignTasks: sum (w in Workers, t in Tasks) x[w][t] == 1
forall (w in Workers)
   assignWorkers: sum (w in Workers, t in Tasks) x[w][t] == 1
```

This declares redundant constraints when some combinations of workers and tasks are irrelevant.

  - A better declaration uses slicing to define the constraints only for valid combinations as defined in `WorkerTasks`:

```
forall (t in Tasks)
   assignTasks: sum (<w,t> in WorkerTasks) x[<w,t>] == 1
forall (w in Workers)
   assignWorkers: sum (<w,t> in WorkerTasks) x[<w,t>] == 1
```

Sparsity and slicing

- **Explicit versus implicit slicing**

  – Choose between implicit and explicit slicing to improve readibility.

  – Explicit:

```
forall(s in shippingLanes)
   sum(<p,r> in productRoutes: s == r) shipment[<p,s>] <=
limit;
```

  – Implicit:

```
forall(s in shippingLanes)
   sum(<p,s> in productRoutes) shipment[<p,s>] <= limit;
```

  – There is usually no performance advantage of implicit over explicit slicing.

Exercise: A pasta production model

Learning objective

- Practice using more advanced data structures, such as tuples, to create an optimization model for pasta production.

Exercise: A pasta production model

- **Problem description**

  – A pasta company manufactures different types of pasta, either

    - In its own factories (inside production), or
    - By contracting other companies (outside production)

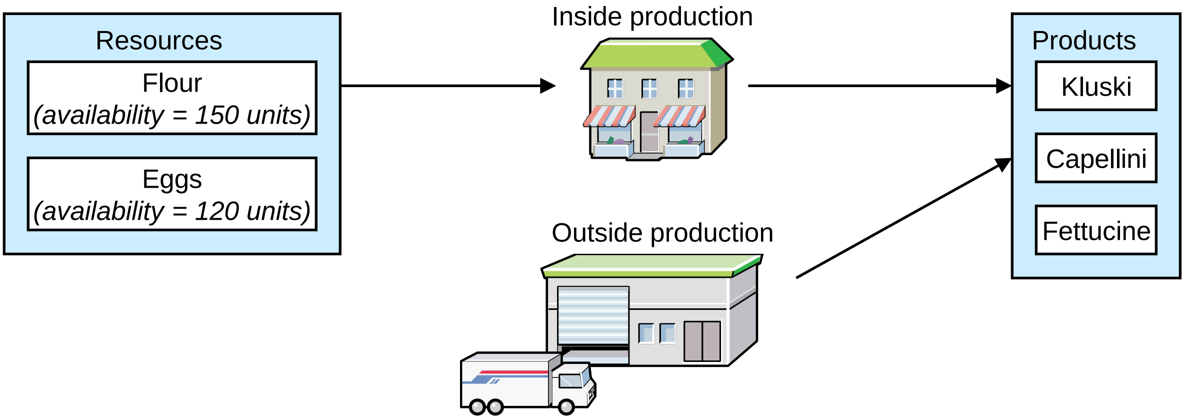
  – For inside production, each product is manufactured from resources with limited availability.

  – Outside production is only limited by contractual agreement.

  – Determine how much of each product to produce inside and outside, while:

    - Satisfying customer demand
    - Minimizing overall production costs
    - Respecting resource and contractual constraints

Pasta production - problem description



Resources

Flour *(availability = 150 units)*

Eggs *(availability = 120 units)*

Inside production

Outside production

Products

Kluski

Capellini

Fettucine

Product	Recipe (for 1 package)	Inside cost ($/package)	Outside cost ($/package)	Max outside production (packages)	Demand (packages)
Kluski	0.5 flour + 0.2 eggs	0.6	0.8	200	100
Capellini	0.4 flour + 0.4 eggs	0.8	0.9	200	200
Fettucine	0.3 flour + 0.6 eggs	0.3	0.4	200	300

Exercise: A pasta production model

Lab

- Write a descriptive model

- On paper, write a descriptive model for this problem. You will use this descriptive model in the next exercise to write the model in OPL.

- Tasks:

1. Define the data (you don't need to specify the data structures at this time)
2. Choose the decision variables
3. Define the objective function
4. Define the constraints

Exercise: A pasta production model

- **Lab**
  Model the pasta production problem in OPL

- Go to the *Pasta Production and Delivery* workshop and complete the first two steps:

Checkpoint

1. Describe tuple and tuple sets in relation to database tables.

2. Are the following declarations in the OPL .dat file correct (yes/no)?
   - `float productionCost = 2.5;`

   -
     ```
     timeInSeconds = 5000;
     timeInHours = timeInSeconds / 3600;
     ```

3. Is the following statement correct (yes/no)? “It is good practice to use arrays inside tuples to create a sparse data representation”

Unit summary

- Having completed this lesson you should be able to:
  - Describe the structure of an OPL model
  - Describe the data types, data structures, and types of variables available in OPL
  - Describe some of the constraints available in OPL
  - Understand the concept of sparsity
  - Write a simple model by using OPL syntax

Working with IBM ILOG Script: basic tasks

Unit objectives

- At the end of this lesson you should be able to:
  - Describe what IBM ILOG Script is used for
  - Initialize data in an IBM ILOG Script `execute` block
  - Perform pre- or postprocessing using IBM ILOG Script `execute` blocks
  - Prepare data using IBM ILOG Script `prepare` blocks
  - Set algorithmic parameters by using IBM ILOG Script
  - Display data by using IBM ILOG Script
  - Explain how to use IBM ILOG Script for flow control

Working with IBM ILOG Script: basic tasks

- Overview of IBM ILOG Script
- Preprocessing and postprocessing
- Data initialization and display
- Setting solver parameters
- Introduction to flow control
- Checkpoint

Overview of IBM ILOG Script

- **Learning objective**
- Gain a general understanding of IBM® ILOG® Script relative to OPL.

Sections

- What is IBM ILOG Script?
- The IBM ILOG Script keywords and functions
- Script blocks
- Script variables

Overview of IBM ILOG Script

- **What is IBM ILOG Script?**
  - A script is a sequence of commands. These commands can be of various types: declarations, simple instructions, and compound instructions.
  - IBM ILOG Script is a JavaScript™ implementation embedded in OPL to provide scripting functionality, including:
    - Preprocessing data in the `.mod` file
    - Postprocessing data in the `.mod` file
    - Processing and formatting data in the `.dat` file
    - Setting CPLEX® Optimizer algorithmic parameters
    - Specifying a search phase for CP Optimizer
    - Flow control, for example to implement decomposition schemes (where a model is decomposed into a set of smaller more manageable models)
  - All OPL model elements are available in a script statement.

Overview of IBM ILOG Script

- **The IBM ILOG Script keywords and functions**
  - IBM ILOG Script provides several keywords and functions to, for example:
    - Declare data (e.g. `Array`, `Number`)
    - Declare scripting variables (e.g. `var`)
    - Facilitate flow control (e.g. `break`, `continue`, `for`, `while`)
    - Display output (e.g. `writeln`)
  - In addition, the complete OPL API library can be accessed through IBM ILOG Script.
  - For a complete reference, see the online help at *Language Quick Reference > IBM ILOG Script Keywords and Functions.*

Overview of IBM ILOG Script

- **Script blocks**

- IBM ILOG Script provides three types of instruction blocks:

  - `execute` blocks
    - Used in the `.mod` file
    - For preprocessing, postprocessing, and setting the CP search phase

  - `main` blocks
    - Used in the `.mod` file
    - For flow control
    - Only one `main` block allowed per model file

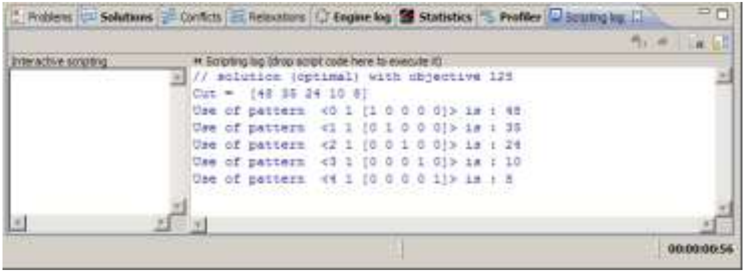
  - `prepare` blocks
    - Used in the `.dat` file
    - For additional operations related to data initialization

Overview of IBM ILOG Script

- **Script variables**
  - Use the `var` keyword to declare script variables.
  - Script variables can be declared in `main` or `execute` blocks.
  - Script variables are local to the blocks in which they are declared.

Overview of IBM ILOG Script

- **Interactive scripting**
  - The IDE contains a testing area in which you can write script code and run it to see immediate results. You don't need to wait for a model to solve after each modification of a script.
  - To enable interactive scripting, you must execute the run configuration at least once. The testing area becomes available in the **Scripting log** tab:



```
// solution (optimal) with objective 125
Cut =  [48 35 24 10 8]
Use of pattern  <0 1 [1 0 0 0 0]> is : 48
Use of pattern  <1 1 [0 1 0 0 0]> is : 35
Use of pattern  <2 1 [0 0 1 0 0]> is : 24
Use of pattern  <3 1 [0 0 0 1 0]> is : 10
Use of pattern  <4 1 [0 0 0 0 1]> is : 8
```

  - If the testing area is not visible, click the **>>** arrows in the scripting log.
  - In the **Interactive scripting** area, enter the script you want to test.
  - Right-click inside the area and select an **Execute** command.
  - The results of the execution are displayed in the **Scripting log**.

Preprocessing and postprocessing

Learning objective

- Understand how to use IBM® ILOG® Script for data pre- and postprocessing.

Sections

- Processing with an execute block in the .mod file
- Preprocessing in an execute block
- Postprocessing in an execute block
- Preprocessing with a prepare block in the .dat file

Preprocessing and postprocessing

- **Processing with an execute block in the .mod file**
  - An `execute` block is used for preprocessing and postprocessing statements in the .mod file:

    ```
    execute <block_name> {
       <statement>;
       <statement>;
       ...
       <statement>;
    }
    ```

  - `<statement>` is any legal IBM ILOG Script instruction or declaration.
  - `<block_name>` is optional. It puts a handle on the block so that it is more easily callable.
  - `execute` blocks *before* the objective or constraints are part of preprocessing.
  - Other blocks are part of postprocessing.

Preprocessing and postprocessing

- **Preprocessing in an execute block**

- The term *preprocessing* can refer to:
  1. Instructions that prepare your data for modeling and solving.
  2. The preprocessing done by the solver engines to reduce problems size.

- IBM ILOG Script is used for the first of these, for example:

```
execute {
   for (var w in Workers) {
      w.salary = w.salary * w.raise;
   }
}
```

- This execute block is used to adjust all workers' salaries before running the model.

Preprocessing and postprocessing

- **Postprocessing in an execute block**

  – Postprocessing is used to manipulate solutions and control output.

  – Some common uses of postprocessing include:

    - Data display, for example:

      ```
      {int} Storesof[w in Warehouses] = { s | s in Stores : Supply[s][w] == 1
      };
      execute STORES{
          writeln("Storesof=",Storesof);
      }
      ```

    - Obtaining the objective value of the current solution and processing it for display in the Scripting Log, for example:

      ```
       execute {
         writeln(“Solution divided by 1000 = “, cplex.getObjValue()/1000);
       }
      ```

Preprocessing and postprocessing

- **Preprocessing with a prepare block in the .dat file**
  - Data preprocessing in the .dat file is useful, for example, when source data is in a different form than required by the model.
  - Use a `prepare` block for preprocessing in the .dat file.
  - Call the preprocessing functions during data initialization by using the `invoke` keyword.
  - For example, if the .mod file contains a declaration for `t` that represents time in hours:

    ```
    float t[1..3]=...;
    ```

    and the unit of time in the source data is minutes, then this code in the .dat file will convert the source data to hours:

    ```
    prepare {
      function transformIntoHours(t) {
          for(var a=0;a<t.length;a++) t[a]=t[a]/60;
          return true;
       }
    }
    ```

    This function should be invoked at data initialization:

    ```
    t = [600,240,150] invoke transformIntoHours;
    ```

Data initialization and display

Learning objective

- Understand how IBM® ILOG® Script can be used to initialize and display data

Sections

- Data initialization
- Initializing arrays – example 1
- Initializing arrays – example 2
- Initializing arrays – example 3
- Data display

Data initialization and display

- **Data initialization**
  - IBM ILOG Script can be used to initialize data in `execute` blocks.
  - Initializing arrays using IBM ILOG Script is less efficient than using a generically initialized array, and is recommended only for complex or multiple initializations.

Data initialization and display

- **Initializing arrays – example 1**

- The tuple set of `Workers` is used to initialize the arrays of `names` and `salaries`:

```
string names[Workers];
int salaries[Workers];
execute {
       for(var w in Workers){
               names[w] = w.name;
               salaries[w] = w.salary;
}
```

Data initialization and display

- **Initializing arrays – example 2**

- The tuple set of `TableRoutes` is used to initialize the `cost` array:

```
■   tuple tableRoute {
         string product;
         string origin;
         string destination;
         float cost;}
{tableRoute} TableRoutes = ...;

tuple connection {
         string origin;
         string destination;}
{connection} Connections = ...;

tuple route {
         string product;
         connection conn;}
{route} Routes = ...;

float cost[Routes] = ...;
execute INITIALIZE {
  for( var t in TableRoutes) {
cost[Routes.find(t.product,Connections.find(t.origin,t.destination))] = t.cost;}}
```

Data initialization and display

- **Initializing arrays – example 3**

- This example shows the general method of using a tuple set to initialize an array (indexed by a tuple) in an execute block.

```
tuple tupleType {int a; int b; int c; };
   {tupleType} tupleSet = {<1,2,3>};
   tuple tupleIndexType {int a; int b;};
   {tupleIndexType} tupleIndex = {<a,b> | <a,b,c> in tupleSet};

int arrayIndexedByTuple[tupleIndex];
execute OPTIONAL_NAME {
  for ( var x in tupleSet )
     arrayIndexedByTuple[tupleIndex.find(x.a,x.b)] = x.c
}
```

This initialization could also be done by using a generically indexed array:

- `int arrayIndexedByTuple2[tupleIndex] = [<a,b>:c | <a,b,c> in tupleSet];`

Data initialization and display

- **Data display**
  - Use the `write` and `writeln` keywords for data display, for example:

```
execute {
   writeln("hire list = ");
   for (var h in hires)
      writeln(h);
}
```

  - To insert a blank line in the display, use `writeln()`
  - `writeln` will display all elements of a set or array on the same line unless the data structure is looped through (e.g. in a `for` loop)

Setting solver parameters

Learning objective

- Understand how to use IBM® ILOG® Script to set algorithmic parameters.

Sections

- The execute block for setting solver parameters
- Example — setting CPLEX Optimizer’s MP parameters
- Example — setting CP Optimizer parameters
- Specifying a CP Optimizer search phase

Setting solver parameters

- **The execute block for setting solver parameters**
  - IBM ILOG Script `execute` blocks can be used to set CPLEX® Optimizer parameters to control, for example:
    - Presolve behavior
    - Solution display
    - Algorithm choice
    - Termination criteria
  - These parameters can also be set in the settings (`.ops`) file.

Setting solver parameters

- **Example — setting CPLEX Optimizer’s MP parameters**

- The following code turns CPLEX Optimizer presolve off and displays iteration information for each iteration of the Simplex Optimizer.

```
execute CPX_PARAM {
       cplex.preind = 0;
       cplex.simdisplay = 2;
 }
```

- For a complete list of CPLEX Optimizer’s MP parameters, see the documentation at *Language > Parameters and settings in OPL > Mathematical programming options > IBM ILOG CPLEX Parameters Reference Manual*.

Setting solver parameters

- **Example — setting CP Optimizer parameters**

- The following code sets the CP Optimizer search time limit to 13 seconds, and the output to the search log to none.

```
execute CP_PARAM {
       cp.param.TimeLimit = 13;
       cp.param.LogVerbosity = "Quiet";
  }
```

- For a complete list of CP Optimizer parameters, see the documentation at *Language > Language User’s Manual > IBM ILOG Script for OPL > Using IBM ILOG Script in constraint programming*.

Setting solver parameters

- **Specifying a CP Optimizer search phase**

  – You can specify a search phase for CP Optimizer by using an IBM ILOG Script `execute` block.

  – This is covered in another lesson on *Solving Simple CP Problems*

Introduction to flow control

Learning objective

- Understand how to use an IBM® ILOG® Script `main` block for flow control.

Sections

- What is flow control?
- The main block

Introduction to flow control

- **What is flow control?**

- Flow control enables control over how models are instantiated and solved:
  - solve several models with different data in sequence or iteratively
  - run multiple "solves" on the same base model, modifying data, constraints, or both, after each solve
  - decompose a model into smaller more manageable models, and solve these to arrive at a solution to the original model (model decomposition)

- Some examples of IBM ILOG Script for flow control are:
  - Implementing column generation (a mathematical programming algorithm)
  - Decomposing a supply chain model into a planning model and a scheduling model and solving these in sequence

- Flow control can also be implemented using the available OPL APIs, for example for implementations in C++ or Java^{TM}.

Introduction to flow control

- **The main block**
  - Flow control statements are written within an IBM ILOG Script `main` block.
  - An example of a simple `main` block is:

```
main {
   model.generate();
   cplex.solve();
}
```

    - `model.generate()` is generates the OPL model
    - `cplex.solve()` calls one of the CPLEX® Optimizers for MP
    - `main` blocks can be written either before the objective or after the constraint block
    - Each `.mod` file can contain at most one `main` block.

Checkpoint

- True of false?

1. An `execute` block must be used for flow control.
2. Script variables declared in a prepare block have scope local to that block.
3. An `execute` block can be used to set solver parameters.

Unit summary

- Having completed this lesson you should be able to:
  - Describe what IBM ILOG Script is used for
  - Initialize data in an IBM ILOG Script `execute` block
  - Perform pre- or postprocessing using IBM ILOG Script `execute` blocks
  - Prepare data using IBM ILOG Script `prepare` blocks
  - Set algorithmic parameters by using IBM ILOG Script
  - Display data by using IBM ILOG Script
  - Explain how to use IBM ILOG Script for flow control

Solving Simple MP Problems

Unit objectives

- At the end of this lesson you should be able to:

- Use some additional OPL functionality for MP problems, such as:
  - Operators
  - Logical constraints
  - Access to reduced costs, duals and slacks
  - Ordered indices

- Apply more of the concepts learned in the preceding lessons, including:
  - Initializing data from the .dat file
  - Initializing sets in a generic way
  - Declaring decision expressions
  - Declaring and labeling constraints

- Convert an LP problem given in business terms to a mathematical model
- Write an LP model in OPL

Solving Simple MP Problems

- MP modeling with OPL
- Supermarket display problem
- Checkpoint

MP modeling with OPL

Learning objective

- Understand the OPL modeling structures and operators used for MP models. Practice writing an LP model in OPL.

Sections

- MP models in OPL
- Basic OPL functionality for MP
- Operators
- Ordered indices
- Logical constraints for MP (1)
- Logical constraints for MP (2)
- Example – logical statements in a logical constraint
- Example – logical statements in a linear constraint
- Conditional constraint declarations using if...else...
- Example – using implication with or
- Constraint labels
- Accessing reduced costs, duals and slacks

MP modeling with OPL

- **MP models in OPL**

- OPL supports several types of Mathematical Programming (MP) models, including:
  - Linear Programming (LP)
  - Integer Programming (IP)
  - Mixed-integer Programming (MIP)
  - Quadratic Programming (QP)

- This exercise at the end of this lesson focuses on the simplest of these, namely LP.

MP modeling with OPL

- **Basic OPL functionality for MP**

  – To formulate MP problems in OPL, you will use the functionality learned in the preceding lessons, as well as additional functionality covered in this lesson, for example:

    - Operators
    - Ordered indices
    - Logical constraints
    - Constraint labels

  – Some of the functionality covered in this topic is also applicable to Constraint Programming (CP) problems.

MP modeling with OPL

- **Operators**

- OPL provides a rich set of operators that can be used for MP models, including:
  - The algebraic operators (`+`, `-`, `*`, `/`)
  - Operators on type `float`, for example
    - `ceil(f)`: The smallest integer greater than or equal to `f`
    - `round(f)`: The nearest integer to `f`
  - Operators on type `int`, for example:
    - `x mod y`: The integer remainder of `x` divided by `y`
  - Operators on type `string`, for example
    - `==`: Equivalent to
    - `!=`: Different from
  - Operators on type `boolean`, for example
    - `&&`: Logical "and"
    - `||`: Logical "or"
  - Set operators, for example
    - `a union b`: The union of set `a` and set `b`

MP modeling with OPL

- **Ordered indices**

  - Use the `ordered` keyword to express the order of index pairs compactly, for example

    ```
    forall (ordered i, j in mySet)
    ```

    is equivalent to:

    ```
    forall (i,j in mySet : ord(mySet,i) < ord(mySet,j))
    ```

  - The `ordered` keyword applies to the `i,j` pair of indices, and not to the set `mySet`.

  - Ordered indices prevents double counting. For example, when calculating the total distance traversing a set of cities, you would count only the distance between city 1 and 2, and not also between city 2 and 1.

  - `ordered` is frequently used in practical problems.

MP modeling with OPL

- **Logical constraints for MP (1)**

- In MP models, a logical constraint can combine linear constraints, other logical constraints, or both, by means of logical operators, including:
  - logical “and”: `&&`
  - logical “or”: `||`
  - logical “not”: `!`
  - different from: `!=`
  - equivalence: `==`
  - implication: `=>`

MP modeling with OPL

- **Logical constraints for MP (2)**

- CPLEX® Optimizer can also extract certain nonlinear expressions internally, that would otherwise require a logical formulation, for example:
  - `min` and `minl` : the minimum of several numeric expressions
  - `abs`: the absolute value of a variable
  - `piecewise:` the piecewise linear combination of a numeric expression

MP modeling with OPL

- **Example – logical statements in a logical constraint**
  - This constraint combines logical statements in a logical constraint:
    `(Use[i] == 0) || (Use[i] >= 20);`
  - `(Use[i] == 0)` will evaluate to 1 if ingredient `i` is used, and 0 otherwise.
  - The constraint states that either none of ingredient `i` is used, or more than 20.
  - Quantities between 0 and 20 are not allowed.
  - This constraint is useful , for example, when minimum production or purchase quantities exist.

- **Example – logical statements in a linear constraint**
  - This constraint combines logical statements in a linear constraint:
    ```
    (Use["ingr1"] == 0) + (Use["ingr2"] == 0) + (Use["ingr3"] ==
    0) + (Use["ingr4"] == 0) >= 2;
    ```
  - `(Use[“ingr1”] == 0)` will evaluate to 1 if `ingr1` is excluded, and 0 otherwise.
  - The constraint states that at least 2 of the ingredients must be excluded.
  - This constraint comes from a blending problem where 2 of 4 possible ingredients must be chosen.

MP modeling with OPL

- **Conditional constraint declarations using if...else...**

  – Logical constraints should not be confused with conditional constraint declaration using if...else... statements, for example:

```
forall(p in Products){
  if (p.code == 1)
   p.production = 2*Use[p][“ingr1”] + 5*Use[p][“ingr2”];
  else
   p.production = 4 * Use[p][“ingr1”] + 3 * Use[p][“ingr3”];}
```

  – This constraint specifies two different recipes depending on the value of a product code.

  – The product `code` is a data input value, and not a decision variable.

  – To express conditional constraints that depend on the outcome of a decision variable, use the implication (=>) operator instead.

- **Example – using implication with or**
  - This constraint uses the implication and or operators:

    ```
    (Use["ingr1"] >= 20) || (Use["ingr2"] >= 20) => Use["ingr3"]
    >= 20;
    ```

  - `(Use[“ingr1”] >= 20)` will evaluate to 1 if `ingr1` is greater than or equal to 20, and 0 otherwise.
  - The constraint states that if at least 20 of either ingredient 1 or ingredient 2 is used, it implies that one should also use at least 20 of ingredient 3.
  - This constraint comes from a blending problem, and is used to enforce the blend recipe.

- **Constraint labels**

  – Constraints can be labeled to help identify them, for example:

```
forall(p in Products)
   ctDemand: Inside[p] + Outside[p] >= Demand[p];
```

  – Constraint labeling has the following advantages:

  - Labeled constraints will be displayed with the solution in the *Problem Browser*, and can be selected to display their slack and dual values.
  - The slack and dual values of labeled constraints can be accessed during post-processing.
  - Labeled constraints are considered by the relaxation and conflict search process in infeasible models.

  – The disadvantage of labeling constraints is decreased performance.

- **Accessing reduced costs, duals and slacks**
  - Reduced costs, duals, and slacks can be accessed by using the following post-processing keywords:where `var` is a variable name and `ct` is a constraint label.
    - `reducedCost(var)`
    - `dual(ct)`
    - `slack(ct)`
  - You can also access reduced costs, duals, and slacks in IBM® ILOG® Script blocks, for example:
    - `variableName.reducedCost`
    - `constraintName.dual`
    - `constraintName.slack`

Supermarket display problem

Learning objective

- Model a real-world linear programming problem in OPL.

Supermarket display problem

- **Managing supermarket display space**
  - This exercise takes you through a basic linear programming problem and shows how you can:
    - Retrieve raw data from an external data source
    - Pre-process data before building the model
    - Post-process data before returning a solution
  - In addition, you will practice:
    - Instantiating and using tuples, arrays and sets
    - Using IBM® ILOG® Script `execute` blocks for pre- and post-processing.

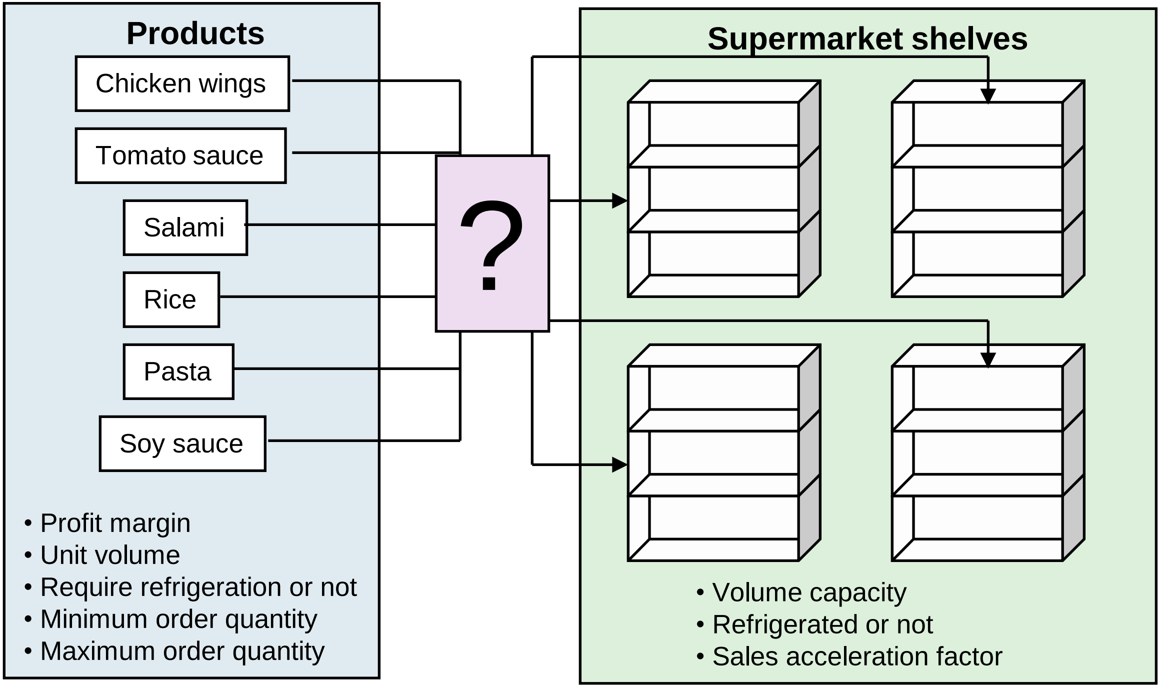
Supermarket display problem

- **Lab**
  Supermarket display problem: Problem description

- A supermarket wants to optimize the display of products on shelves in order to maximize profit.

Supermarket display - problem description

Determine the quantity of each product to display on each shelf, as well as the total quantity to order of each product, to maximize profit while respecting the storage constraints.



- Profit margin
- Unit volume
- Require refrigeration or not
- Minimum order quantity
- Maximum order quantity

- Volume capacity
- Refrigerated or not
- Sales acceleration factor

Supermarket display problem

- **Lab**

- Go to the *Supermarket Display* workshop and complete the following steps:
  - *Model the input data*
  - *Model the decision variables*
  - *Write the objective function and constraints*
  - *Display the results*

Checkpoint

1. Consider a variable, `invest[j]` to decide whether to invest in investment j (`invest[j]` = 1) or not (`invest[j]` = 0). Which of the following two code segments are correct to force investment B *not* to be chosen when investment A is chosen:

   •A:

   ```
   if (invest["A"]==1)
        then invest["B"] = 0;
   ```

   •B:

   ```
   invest["B"] + invest["A"] <= 1;
   ```

2. True or false: You should label all constraints to be able to easily track conflicts.

3. True or false: Introducing logical constraints in an LP model results in a Mixed-Integer model.

Unit summary

- Having completed this lesson you should be able to:

- Use some additional OPL functionality for MP problems, such as:
  - Operators
  - Logical constraints
  - Access to reduced costs, duals and slacks
  - Ordered indices

- Apply more of the concepts learned in the preceding lessons, including:
  - Initializing data from the .dat file
  - Initializing sets in a generic way
  - Declaring decision expressions
  - Declaring and labeling constraints

- Convert an LP problem given in business terms to a mathematical model
- Write an LP model in OPL

Solving Simple CP Problems

Unit objectives

- At the end of this lesson you should be able to:
  - Describe what Constraint Programming (CP) is
  - Describe the major benefits of using IBM ILOG CPLEX CP Optimizer
  - Model a simple CP problem in OPL
  - Specify a CP search phase in OPL

Solving Simple CP Problems

- Introduction to CP
- CP models in OPL
- Checkpoint

Introduction to CP

Learning objective

- Understand the principles of Constraint Programming and how to address CP problems by using OPL and IBM® ILOG® CPLEX® CP Optimizer.

Sections

- What is Constraint Programming?
- CP Optimizer and CPLEX Studio
- Benefits of CP Optimizer
- Scheduling with CP Optimizer
- Combinatorial optimization with CP Optimizer
- How does CP Optimizer work?
- Constructive search
- Search strategies
- CP search phases
- Search phases — general syntax
- Value and variable choosers
- A typical search phase
- Specifying multiple search phases

Introduction to CP

- **What is Constraint Programming?**
  - Constraint Programming (CP) is a discipline of computer science, closely associated with artificial intelligence.
  - It is based on logic and symbolic reasoning.
  - CP is very effective in solving, for example:
    - Detailed scheduling problems
    - Routing problems
    - Satisfiability problems
    - Certain combinatorial optimization problems not well-suited for traditional Mathematical Programming (MP) methods

Introduction to CP

- **CP Optimizer and CPLEX Studio**
  - CP Optimizer combines a state-of-the-art CP solver with CPLEX Studio's development, debugging and tuning features.
  - CP Optimizer is intended for the following two categories of problems:
    - Detailed scheduling problems
      - Decision variables are of type `interval` and describe an interval of time during which, for example, a task occurs
      - There is no discrete enumeration of time
    - Certain combinatorial optimization problems that cannot easily be linearized and solved using traditional MP methods
      - These problems contain only discrete decision variables

Introduction to CP

- **Benefits of CP Optimizer**

- The benefits of CP Optimizer as embedded in CPLEX Studio include:
  - Automatically generated advanced algorithms based on the mathematical formulation of a model
  - Easy-to-use representations for specialized constraints used in scheduling and other combinatorial problems
  - Users can “model and run” complex problems, without having to write the complex searches traditionally required for CP.
  - The advanced user can still specify a search if desired.

Introduction to CP

- **Scheduling with CP Optimizer**

  – Scheduling problems involve:
    - Optimally assigning start and end times to task intervals
    - Management of minimal or maximal capacity constraints for resources over time

  – CP Optimizer as embedded in CPLEX Studio provides elements to concisely represent complex scheduling problems, for example:
    - Variables of type `interval`, with attributes of start, end, size and intensity
    - Precedence constraints, for example `endBeforeStart` to indicate that the end of one interval must occur before the start of another
    - Cumulative expressions to define resource constraints, for example `cumulFunction, step` and `pulse`
    - Other elements to model sequencing, synchronization, and so forth
    - Gantt Chart solution representation for `interval` variables

Introduction to CP

- **Combinatorial optimization with CP Optimizer**

  – CP is well-suited for certain combinatorial optimization problems cannot easily be linearized or solved using traditional MP techniques.

  – The elements of these types of problems are:

    - A set of discrete decision variables (integer or Boolean)
    - A predefined domain for each variable designating its possible values
    - A set of constraints defined according to the rules of CP
    - Optionally, an objective function to be minimized or maximized

  – You will explore the use of CP Optimizer for combinatorial optimization, other than scheduling, in more detail later in this lesson.

Introduction to CP

- **How does CP Optimizer work?**

  – CP methodology, in general, has two phases:

    1. Write a model representation of a problem in a computer programming language
    2. Describe a search strategy for solving the problem

  – A typical CP search uses the following techniques iteratively until a solution is found:

    - Domain reduction: The process of eliminating possible values from a variable domain by considering the constraints on that variable and related variables.
    - Constraint propagation: The process of communicating the domain reduction of a decision variable to all of the constraints on this variable. This can result in more domain reductions.

Introduction to CP

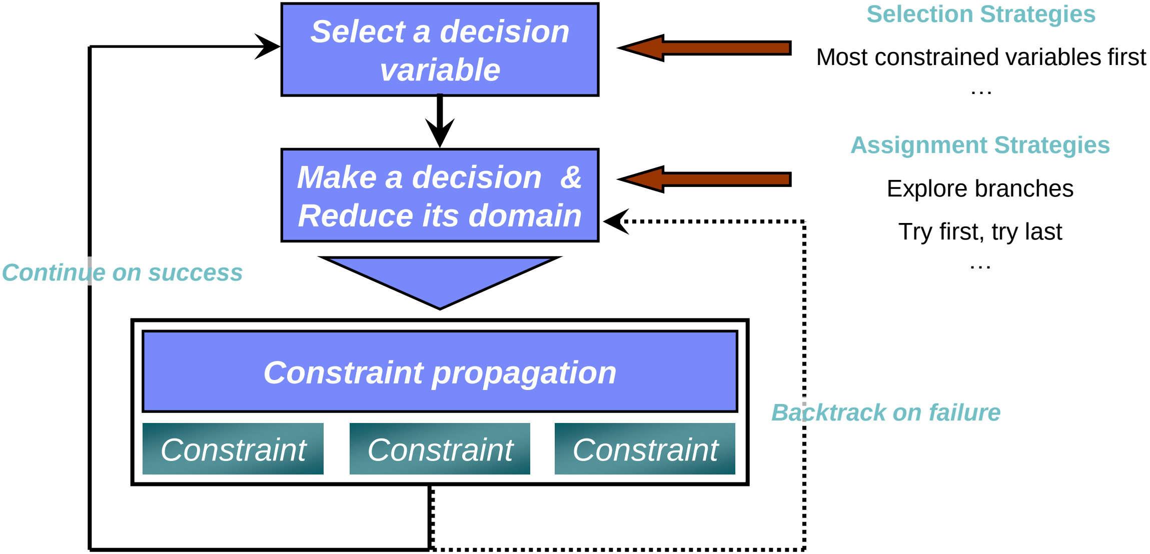
- **Constructive search**

  – CP Optimizer uses a process called *constructive search* to *construct* a solution following these steps:

    1. Select a decision variable
    2. Assign a value to the decision variable
    3. Reduce the domains of the other variables using constraint propagation
    4. If step 3 fails, backtrack to step 2 and assign a different value to the variable, otherwise return to step 1

  – The process continues until all decision variables have a value, or it is established that no solution exists.

CP constructive search process



Selection Strategies

Most constrained variables first

…

Assignment Strategies

Explore branches

Try first, try last

…

Introduction to CP

- **Search strategies**
  - With CP Optimizer, the user does not need to describe a search strategy, because CP Optimizer will automatically generate a search based on:
    - the model structure
    - constraint propagation
  - However, the user can optionally fine-tune the search strategies by:
    - modifying search parameters
    - specifying more detailed search phases

Introduction to CP

- **CP search phases**
  - In general, CP Optimizer's built in search works well without additional guidance.
  - However, a search phase can be used to tune a search strategy by:
    - specifying the criteria for the order in which decision variables are chosen to be fixed
    - specifying the values decision variables should be fixed to
    - specifying both the order and the values
  - This strategy is then used to instantiate the decision variables of the phase.
  - Specifying a search phase can in some cases have a significant influence on the processing time.

Introduction to CP

- **Search phases — general syntax**

  – The general syntax of a phase that specifies both a variable chooser and a value chooser is as follows:

  ```
  var phase1 = f.searchPhase(x,<variable chooser>, <value
  chooser>);
  ```

  – `<variable chooser>` specifies how the next decision variable to fix in the search is chosen

  – `<value chooser>` specifies how values are chosen for instantiating decision variables

Introduction to CP

- **Value and variable choosers**

  – A variable chooser is a combination of selectors and evaluators. For example,

  ```
  f.selectSmallest(f.domainSize())
  ```

  is a variable chooser that evaluates the domain size (the evaluator is `f.domainSize()`) of each variable and selects the one having the smallest size (the selector is `f.selectSmallest()`).

  – A value chooser is defined according to the same template. For example,

  ```
  f.selectLargest(f.value())
  ```

  selects the largest value of the domain to instantiate the variable chosen.

  – The search phase using these choosers is then

  ```
  var phase3 = f.searchPhase(z, f.selectSmallest(f.domainSize()),
  f.selectLargest(f.value()));
  ```

Introduction to CP

- **A typical search phase**
  - In the OPL model, search phases are written using IBM® ILOG Script `execute` blocks located after the decision variable declarations and before the objective function definition.
  - A typical search phase definition, where `x` is a decision variable, is as follows:

```
execute {
   var f = cp.factory;
   var phase1 = f.searchPhase(x);
   cp.setSearchPhases(phase1);
}
```

Introduction to CP

- **Specifying multiple search phases**

  – You can also specify a sequence of search phases, for example:

```
execute {
  var f = cp.factory;
  var phase1 = f.searchPhase(x);
  var phase2 = f.searchPhase(y);
 cp.setSearchPhases(phase1, phase2);
}
```

  – This tells CP Optimizer to search first on the `x` decision variable and then on the `y` decision variable.

CP Optimizer custom search

- The OPL API for C++ includes the `IloCP` class for implementing a custom search, beyond the search phase functionality available in CPLEX Studio.

- An instance of `IloCP` represents a search engine for CP Optimizer.

- `IloSearchPhase` allows coding a search phase in C++.

- `IlcGoal` is the lowest level search primitive, and allows complete freedom in coding the search.

- Only use goals (`IlcGoal`) after exhausting other possibilities, because a custom search increases maintenance.

- For a complete description of the `IloCP` class, see the documentation at **Interfaces > C++ interface reference manual > optim.cpoptimizer > Classes > IloCP**

CP models in OPL

Learning objective

Learn how to use IBM® ILOG® CPLEX® CP Optimizer and OPL to solve certain combinatorial optimization problems, other than scheduling problems.

Sections

- CP models for combinatorial optimization
- Invoking CP Optimizer
- Types of CP constraints in OPL
- Arithmetic expressions and constraints
- Logical constraints (1)
- Logical constraints (2)
- Conditional constraint declarations
- Compatibility constraints
- Compatibility constraints – example
- Specialized constraints
- The all quantifier
- The all quantifier – example

CP models in OPL

- **CP models for combinatorial optimization**
  - CP Optimizer effectively solves certain combinatorial optimization problems, other than scheduling problems.
  - These problems are not well-suited for traditional MP methods, but effectively solved using CP.
  - The focus of this topic is on syntax and basic modeling constructs.
  - The content in this topic is valid for any CP application written in OPL.

CP models in OPL

- **Invoking CP Optimizer**
  - The OPL keyword `using` at the beginning of a model file indicates whether an MP or CP solution approach should be used.
  - The default is to use one of CPLEX Optimizer’s MP algorithms (this can be forced with `using CPLEX;` at the beginning of a model file).
  - To use CP Optimizer, type `using CP;` as the first line of the `.mod` file.
  - If a model contains CP-specific constraints and `using CP` is not present, you will get an error messages.

CP models in OPL

- **Types of CP constraints in OPL**

- CP constraints in OPL can be classified as:
  - Arithmetic expressions and constraints
  - Logical constraints
  - Compatibility constraints
  - Scheduling constraints
  - Specialized constraints

- For a full list, refer to the documentation: Language > Language Reference Manual > OPL, the modeling language > Constraints > Types of Constraints > Constraints available in Constraint Programming.

- Scheduling constraints are not covered in this topic; they are the subject of another (optional) lesson entitled *Scheduling with CP Optimizer*.

CP models in OPL

- **Arithmetic expressions and constraints**

- A CP modeler has access to a large number of arithmetic operators in OPL, grouped as:

  - Arithmetic operations, for example:
    - The usual arithmetic operators (`+, -, *, /`)
    - `div` to return the integer portion of a division

  - Arithmetic expressions, for example:
    - `element(<int_array>,n)` to return the `n`th element of an `int` array, where n is an `int` decision variable.
    - `count (<arg_1>,<arg_2>)` to count how many of the elements in the `int` array `<arg_1>` are equal to the `int` value `<arg_2>`

  - Arithmetic constraints, for example:
    - `!=`: not equal to
    - `>=`: greater than or equal to

CP models in OPL

- **Logical constraints (1)**

- The following logical operators, also valid for MP models, are available for CP models in OPL:
  - And (`&&`)
  - Or (`||`)
  - Not (`!`)
  - Imply (`=>`)
  - Different from (`!=`)
  - Equivalent (`==`)

CP models in OPL

- **Logical constraints (2)**
  - The `and` and `or` aggregate constraints are valid in CP constraint blocks.
  - `and` has the same effect as the `&&` operator, except that it can be used with indices to aggregate several "logical-and" constraints, for example:

    ```
    subject to {
      and(i in 1..5:i % 2==1) x[i]==1;
    }
    ```
  - On a high level, `and` is equivalent to `forall`:

    ```
    subject to {
      forall(i in 1..5:i % 2==1) x[i]==1;
      }
    ```
  - `or` has the same effect as the `||` operator, except that it can be used with indices to aggregate several "logical-or" constraints, for example:

    ```
    or(i in 1..5 : i mod 2 == 0) x[i] == 2;
    ```
  - `and` and `or` can be combined to create complex logical constraints, for example:

    ```
        or(i in 1..9) and(j in i..i + 1) x[j]==1;
    ```
  - `and` and `or` are not allowed in MP constraint blocks.

CP models in OPL

- **Conditional constraint declarations**

  – The conditional constraint declarations, if and if-else, are valid for both MP and CP models, for example:

  ```
  subject to
  { if  (<condition>) <constraint1>;
   else <constraint2>;
  }
  ```

  – Only data items are allowed in `<condition>` (no decision variables)

  – To use decision variables in conditional constraint declarations, use the logical implication (`=>`) operator instead.

CP models in OPL

- **Compatibility constraints**

  – Compatibility constraints define combinations of allowed or forbidden values for any number of integer decision variables.

  – Examples of compatibility constraints allowed in CP constraint blocks are:
    - `allowedAssignments({tuple-type},int, ...)`
    - `forbiddenAssignments({tuple-type},int, ...)`

  – The arguments of these constraints are:
    - `{tuple-type}`: A tuple type where the elements define an allowed (or forbidden) combination.
    - `int,...`: Any number of integer decision variables for which combinations are considered.

  – These constraints can also be used with integer arrays outside constraint blocks for either CP or MP models.

CP models in OPL

- **Compatibility constraints – example**

- The following example illustrates these compatibility constraints:
-
```
  using CP;

tuple pair {
   int a;
   int b;
};

{pair} possible = {<1,1>, <2,4>};
{pair} forbidden = {<3,5>};

dvar int+ x;
dvar int+ y;

constraints {
  allowedAssignments(possible, x, y);
  forbiddenAssignments(forbidden, x, y);
}
```

CP models in OPL

- **Specialized constraints**
  - A specialized constraint is a concise equivalent of a set of arithmetic or logical constraints.
  - Examples of specialized constraints are:
    - `allDifferent`: Constrains decision variables in a `dvar` array to all take different values
    - `lex`: States that the values of a first array of decision variables is less than or equal to the values of a second array of decision variables, in lexical order.
    - `pack`: Maintains the load of a set of containers or bins, given a set of weighted items and an assignment of items to containers.
  - In most cases, a specialized constraint achieves more domain reduction than the equivalent set of basic constraints.
  - Specialized constraints can also be used outside constraint blocks with integer arrays in CP and MP models.

CP models in OPL

- **The all quantifier**

  – `all` is similar to `forall`, but is used in a different part of the model:

    - `forall` is used to create instances of constraints in a model.
    - `all` is used to create a set of objects that are used as arguments for certain CP constructs.

  – When used with a specialized constraint, `all` allows selection of a part of an array. For example:

    ```
    allDifferent(all(i in 1..3) c[i]);
    ```

    Here, only the first three items of the array `c` are required to be different.

  – `all` preserves the order when an index is present.

  – This is very useful in conjunction with specialized constraints that use the order of the array of variables passed as arguments.

CP models in OPL

- **The all quantifier – example**

  – The following code illustrates the use of `all`, as well as some specialized constraints:

  ```
  using CP;
  dvar int a[1..3] in 1..10;
  dvar int b[1..3] in 1..10;
  dvar int c[1..6] in 1..10;
  constraints{
    lex(a,b);
    allDifferent(append(a,b));
    lex(all(i in 1..3) c[i], all(i in 4..6) c[i]);
    allDifferent(c);}
  ```

  – The result from these constraints is:

  ```
  CP found a solution:
  a = [1 5 3];
  b = [2 4 6];
  c = [1 6 5 2 3 4];
  ```

CP models in OPL

- **Lab**
  The steel mill inventory matching problem

- In this exercise, you will practice:
  - Deriving a CP model from a problem description
  - Writing the CP model in OPL
  - Using search phases in OPL
  - Using alternative formulations to improve solution speed (optional)

CP models in OPL

- **Lab**
  Steel mill problem description

  - A steel mill inventories steel slabs of different sizes
  - Steel slabs are used to manufacture steel coil, with a given amount of slab required per coil order
  - During production, some steel from slabs is lost
  - Production manager should match coil orders with steel slabs so as to minimize losses
  - The complete problem description is given in the workshop

- Go to the *Steel mill inventory matching* workshop and complete all the steps.

Checkpoint

- True or false?

1. CP involves a fine-grained enumeration of time in order to determine start and end times of tasks to be scheduled.

2. In order to solve a CP problem, you must specify a search phase.

3. The `all` operator has the same effect as the `&&` operator, except that it can be used with indices to aggregate several “logical-and” constraints.

Unit summary

- Having completed this lesson you should be able to:
  - Describe what Constraint Programming (CP) is
  - Describe the major benefits of using IBM ILOG CPLEX CP Optimizer
  - Model a simple CP problem in OPL
  - Specify a CP search phase in OPL

Scheduling with CP Optimizer

Unit objectives

At the end of this lesson you should be able to:

- Explain what an interval variable is.
- Use OPL keyword to express scheduling functions and constraints, including:
  - calendars
  - alternative resources
  - resource pools
  - precedence constraints
  - cumulative functions
  - sequencing constraints
  - no overlap constraints
  - synchronize constraints
  - isomorphism constraint
- Write a simple scheduling model in OPL.

Scheduling with CP Optimizer

- Introduction to scheduling
- A simple scheduling problem
- Scheduling constraints
- Putting everything together - a staff scheduling problem
- A house building calendar problem
- Matters of State: Understanding State Functions
- Applying state functions: a wood cutting problem
- Checkpoint

Introduction to scheduling

Learning objective

- At the end of this lesson, you will be able to write a simple scheduling model in OPL, and solve it by using IBM® ILOG® CPLEX® CP Optimizer.

Sections

- What is a detailed scheduling problem?
- Defining a typical scheduling problem
- Scheduling models in OPL

Introduction to scheduling

What is a detailed scheduling problem?

- Detailed scheduling involves:
  - Assigning start and end times to intervals
  - Deciding which alternative will be used if an activity can be performed in different modes
  - Management of minimal or maximal capacity constraints for resources over time

Introduction to scheduling

Defining a typical scheduling problem

- A typical scheduling problem is defined by:
  - A set of time intervals -- definitions of activities, operations, or tasks to be completed, that might be optional or mandatory
  - A set of temporal constraints – definitions of possible relationships between the start and end times of the intervals
  - A set of specialized constraints – definitions of the complex relationships on a set of intervals due to the state and finite capacity of resources
  - A cost function – for instance, the time required to perform a set of tasks, non execution cost of some optional tasks, or the penalty costs of delivering some tasks past a due date

Introduction to scheduling

Scheduling models in OPL

- A scheduling model has the same format as other models written in OPL:
  - Data structure declarations
  - Decision variable declarations
  - Objective function
  - Constraint declarations

- OPL provides specialized variables, constraints and keywords designed for modeling scheduling problems.

A simple scheduling problem

Learning objective

- Understand the scheduling framework as it is declared in OPL, for IBM® ILOG® CPLEX® CP Optimizer.

Sections

- Task declaration in OPL
- Precedence constraints in OPL
- Intervals – the tasks to schedule
- Functions on intervals
- Intensity (calendar functions)

A simple scheduling problem

- The example that follows, a simple house building problem, declares a series of tasks (`dvar interval`) of fixed time duration (`size`) that need to be scheduled (assigned start and end times).

- These tasks have *precedence constraints*. This means that one task must be completed before another can start. For example, in the example code, the `carpentry` task must be complete before the `roofing` task can start.

A simple scheduling problem

Task declaration in OPL

-
  ```
  using CP;
  dvar interval masonry size 35;
  dvar interval carpentry size 15;
  dvar interval plumbing size 40;
  dvar interval ceiling size 15;
  dvar interval roofing size 5;
  dvar interval painting size 10;
  dvar interval windows size 5;
  dvar interval facade size 10;
  dvar interval garden size 5;
  dvar interval moving size 5;
  ```

A simple scheduling problem

Precedence constraints in OPL

```
subject to {
endBeforeStart(masonry, carpentry);
endBeforeStart(masonry, plumbing);
endBeforeStart(masonry, ceiling);
endBeforeStart(carpentry, roofing);
endBeforeStart(ceiling, painting);
endBeforeStart(roofing, windows);
endBeforeStart(roofing, facade);
endBeforeStart(plumbing, facade);
endBeforeStart(roofing, garden);
endBeforeStart(plumbing, garden);
endBeforeStart(windows, moving);
endBeforeStart(facade, moving);
endBeforeStart(garden, moving);
endBeforeStart(painting, moving);
}
```

A simple scheduling problem

Intervals – the tasks to schedule

- In OPL, tasks, such as the activities involved in house building problem, are modeled as *intervals*, represented by the decision variable type `interval`. An interval has the following attributes:
  - A start
  - An end
  - A size
  - Can be optional
  - An intensity (calendar function)

- The time elapsed between the start and the end is the *length* of an interval.

- The *size* of an interval is the time required to perform the task without interruptions.

- An interval decision variable allows these attributes to vary in the model, subject to constraints.

A simple scheduling problem

Syntax:

- `dvar interval <taskName> <switches>`

- where `<switches>` represents one or more different modifying conditions to be applied to the interval.

A simple scheduling problem

Functions on intervals

- A number of functions are available with intervals. These are normally used in decision expressions (using the keyword, `dexpr`) to access an aspect of the interval. Some of these include:
  - `endOf` – integer expression used to access the end time of an interval
  - `startOf` – integer expression used to access the start time of an interval
  - `lengthOf` – integer expression used to access the length of an interval
  - `sizeOf` – integer expression used to access the size of an interval
  - `presenceOf` – integer expression returning 1 if an optional interval is present, and 0 otherwise

A simple scheduling problem

Intensity (calendar functions)

- A calendar (or *intensity* function) can be associated with an interval decision variable.
- Intensity is a function that applies a measure of usage or utility over an interval length.
- For example, it can be used to specify the availability of a person or physical resource (such as a machine) during the interval.
- Syntax:

```
dvar interval <taskName> intensity F;
```

- where `F` is a stepwise function with integer values
  - The intensity is 100% by default, and can not exceed this value (granularity of 100).
  - If a task cannot be processed at all during a certain time window, such as a break or holiday, then the intensity for that time period is set to 0.
  - When a task is processed part-time (this can be due to worker time off, interaction with other tasks, etc.) the intensity is expressed as a positive percentage.

A simple scheduling problem

- Consider a task, for example, `decoration`, that is performed during an interval one week in length.

- In this interval a worker works five full days, one half day, and has one day off; the intensity function would be:
- 100% for five days,
- 50% for one day, and
- zero for the last day.

- You declare the intensity values using a linear stepwise function, via the OPL keyword `stepFunction`.

A simple scheduling problem

- Interval size, length, and intensity are always related by the following:
- *size multiplied by granularity is equal to the integral of the intensity over the length of the interval*

- Intensity can not exceed 100%, so interval size can never exceed the interval length.

- Therefore, in the proceeding example, the interval length is seven days and size equals 5.5 work days, and would be declared as follows:

-
```
stepFunction F = stepwise(0—>1; 100–>5; 50–>6; 0–>7);
dvar interval decoration size 5..5 in 1..7 intensity F;
```

Scheduling constraints

Learning objective

- Understand how to use specialized constraints for scheduling with IBM® ILOG® CPLEX® CP Optimizer

Sections

- Precedence constraints
- Cumulative constraints
- Sequence decision variable and no overlap constraints
- The Sequence decision variable
- No overlap constraints
- Alternative and span constraints
- Synchronize constraint

Scheduling constraints

Precedence constraints

- *Precedence constraints* are common scheduling constraints used to restrict the relative position of interval variables in a solution.

- These constraints are used to specify when one interval variable must start or end with respect to the start or end time of another interval.

- A delay, fixed or variable, can be included.

- For example a precedence constraint can model the fact that an activity `a` must end before activity `b` starts (optionally with some minimum delay `z`).

  List of precedence constraints in OPL:

  - `endBeforeStart`
  - `startBeforeEnd`
  - `endAtStart`
  - `endAtEnd`
  - `startAtStart`
  - `startAtEnd`

Scheduling constraints

Example syntax:

```
startBeforeEnd (a,b[,z]);
```

- Where the end of a given time interval `a` (modified by an optional time value `z`) is less than or equal to the start of a given time interval `b`:
- `s(a) + z ≤ s(b)`
- Thus, if the ceiling had to dry for two days before the painting could begin, you would write:
- `endBeforeStart(ceiling, painting, 2);`

Scheduling constraints

Cumulative constraints

- In some cases, there may be a restriction on the number of intervals that can be processed at a given time, perhaps because there are limited resources available.

- Additionally, there may be some types of *reservoirs* in the problem description (cash flow or a tank that gets filled and emptied).

- These types of constraints on resource usage over time can be modeled with constraints on *cumulative function* expressions.

- A cumulative function expression is a step function that can be incremented or decremented in relation to a fixed time or an interval.

- A cumulative function expression is represented by the OPL keyword `cumulFunction`.

Scheduling constraints

Syntax:

- `cumulFunction <functionName> = <elementary_function_expression>;`
- where `<elementary_function_expression>` is a cumulative function expression that can legally modify a `cumulFunction`. These expressions include:
  - `step`
  - `pulse`
  - `stepAtStart`
  - `stepAtEnd`

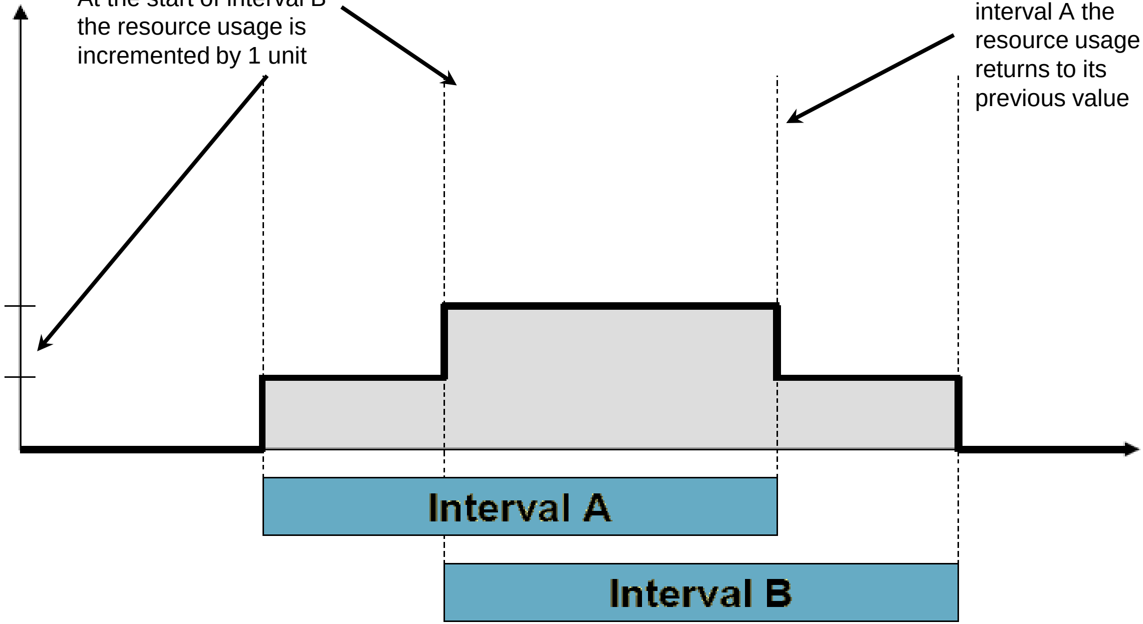
Scheduling constraints

- A cumulative function expression can be constrained to model limited resource capacity by constraining that the function be less than the capacity:

- `workersUsage <= NbWorkers;`

The Pulse cumulative function

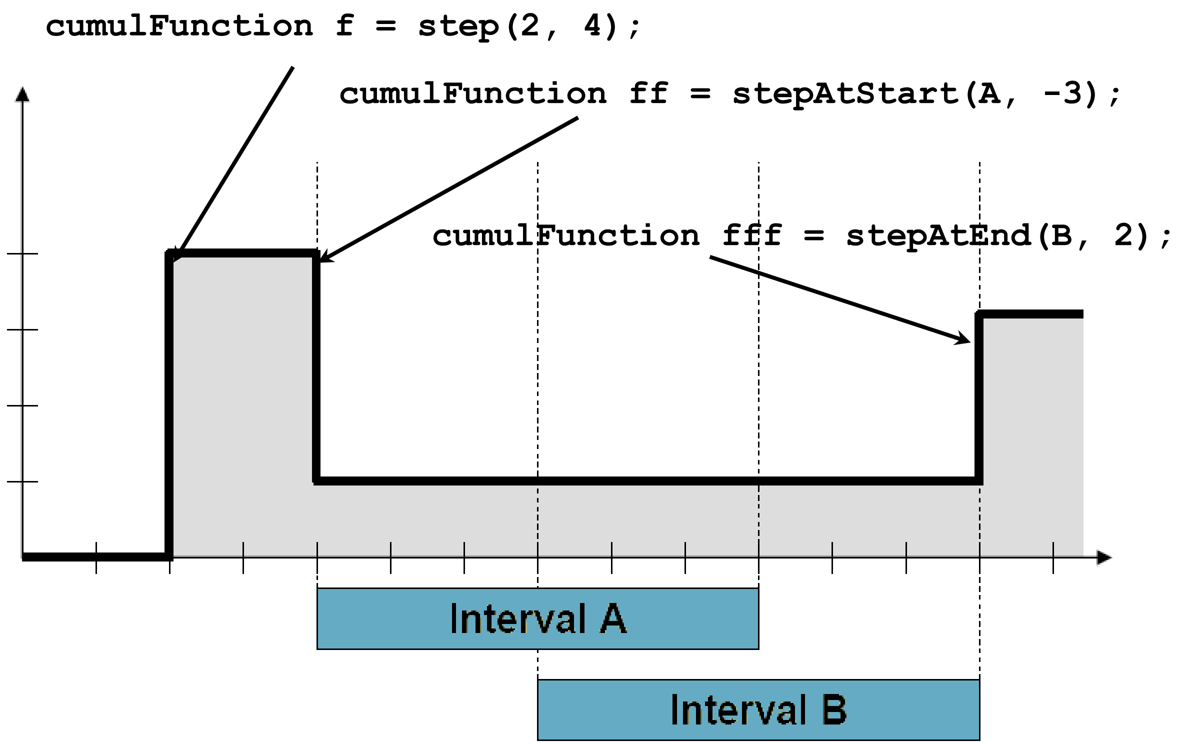
```
cumulFunction f = pulse(A, 1);
cumulFunction ff = pulse(B, 1);
```



At the start of interval B the resource usage is incremented by 1 unit

At the end of interval A the resource usage returns to its previous value

Step cumulative functions



```
cumulFunction f = step(2, 4);
cumulFunction ff = stepAtStart(A, -3);
cumulFunction fff = stepAtEnd(B, 2);
```

Scheduling constraints

Sequence decision variable and no overlap constraints

- A scheduling model can contain tasks that must not overlap, for example, tasks that are to be performed by a given worker cannot occur simultaneously.

- To model this, you use two constructs:
  - The `sequence` decision variable
  - The `noOverlap` scheduling constraint

- Unlike precedence constraints, there is no restriction on relative position of the tasks. In addition, there may be transition times between tasks.

Scheduling constraints

The Sequence decision variable

- Sequences are represented by the decision variable type `sequence`.

- Syntax:

```
dvar sequence <sequenceName> in <intervalName> [types T];
```

- where `T` represents a non-negative integer.

Scheduling constraints

No overlap constraints

- To constrain the intervals in a sequence such that they:
  - Are ordered in time corresponding to the order in the sequence
  - Do not overlap
  - Respect transition times
- OPL provides the constraint `noOverlap`.
- Syntax:

```
noOverlap (<sequenceName> [,M]);
```

- where `<sequenceName>` is a previously declared sequence decision variable, and `M` is an optional transition matrix (in the form of a tuple set) that can be used to maintain a minimal distance between the end of one interval and the start of the next interval in the sequence.

Scheduling constraints

-
  ```
  {int} Types = { T[i] | i in 1..n };
  tuple triplet { int id1; int id2; int value; };
  {triplet} M = { <i,j,ftoi(abs(i-j))> | i in Types, j in Types };

  dvar interval A[i in 1..n] size d[i];
  dvar sequence p in A types T;

  subject to {
    noOverlap(p, M);
  };
  ```

Scheduling constraints

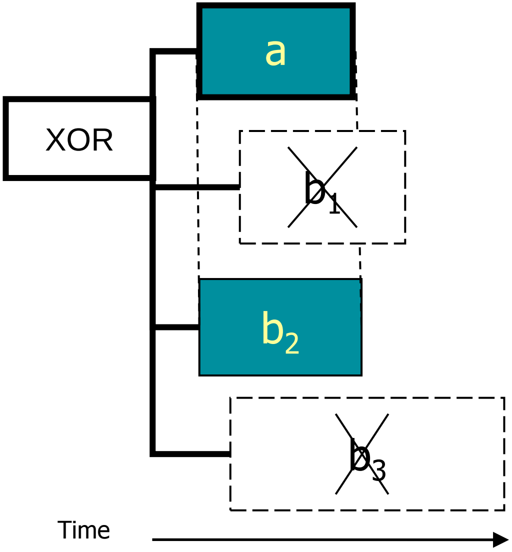
Alternative and span constraints

- The two keywords `alternative` and `span` provide important ways to control the execution and synchronization of different tasks.

- An `alternative` constraint between an interval decision variable `a` and a set of interval decision variables `B` states that interval `a` is executed if and only if exactly one of the members of `B` is executed. In that case, the two tasks are synchronized.

- That is, interval `a` starts together with the first present interval from set `B` and ends together with it. No other members of set `B` are executed, and interval `a` is absent if and only if all intervals in the set `B` are absent, as shown in the following diagram:

Alternative constraints



Example:
`alternative(a,[b1..bn]);`
where **b_2** is the first present interval in the set.

a and **b_2** are executed simulaneously. No other intervals are executed.

Scheduling constraints

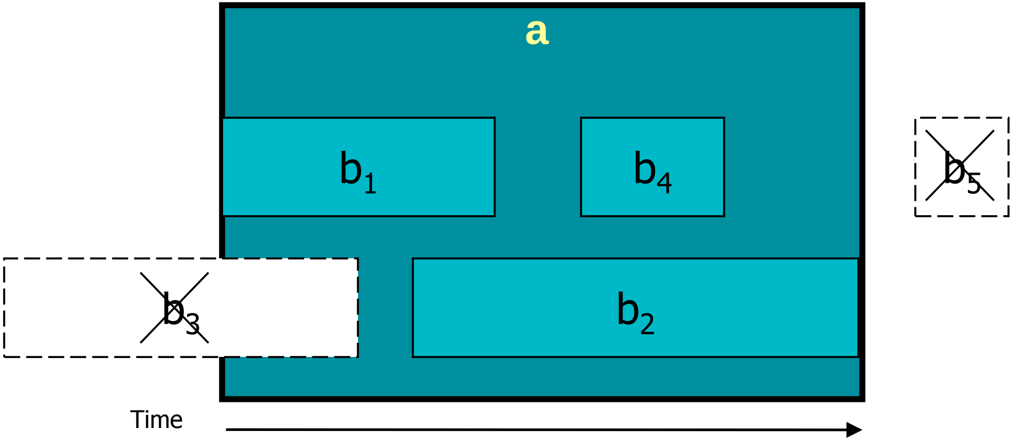
- A span constraint between an interval decision variable `a` and a set of interval decision variables `B` states that interval `a` spans over all intervals present in the set.

- That is: interval `a` starts together with the first present interval from set `B` and ends together with the last one.

- Interval `a` is absent if and only if all intervals in the set `B` are absent, as shown in the following diagram:

Span constraints

Example: `span(a,[b1..bn]);`



a starts together with **b_1** and ends together with **b_2.**
Intervals **b_3** and **b_5** are not executed.

Scheduling constraints

- Examples:

```
alternative(tasks[h] [t], all(s in Skills: s.task==t) wtasks[h]
[s]);

span(house[i], all(t in tasks : t.house == i) tasks[t]);
```

Scheduling constraints

Synchronize constraint

- A synchronization constraint (keyword `synchronize`) between an interval decision variable `a` and a set of interval decision variables `B` makes all present intervals in the set `B` start and end at the same times as interval `a`, if it is present.

- Example:

```
synchronize(task[i], all(o in opers : o.task == i) tiopers[o]);
```

Scheduling constraints

Isomorphism constraint

- An isomorphism constraint states that in a solution there is a one-to-one correspondance between the intervals present in two arrays.

- Example:

```
forall (m in 2..3)
  presenceOf(spans[m]) == true;
forall (s in 3..4)
  presenceOf(intervals[s]) == true;
isomorphism(all[1..12](i in 1..12) spans[i],
            all(i in 1..15) intervals[i],
            all(i in 1..15) indices[i], 0);
```

Putting everything together - a staff scheduling problem

Learning objective

- Use OPL scheduling keywords and syntax to model a simple staff scheduling problem, and solve it with IBM® ILOG® CPLEX® CP Optimizer

Sections

- The business problem
- What is the objective?
- What are the unknowns?
- Modeling alternative resources
- Modeling resource committed to at most one task at a time
- Surrogate constraints

Putting everything together - a staff scheduling problem

The business problem

- A telephone company must schedule customer requests for installation of different types of telephone lines:
  - First (or principal) line
  - Second (or additional) line
  - ISDN (digital) line
- Each request has a requested due date; a due date can be missed, but the objective is to minimize the number of days late.
- These three request types each have a list of tasks that must be completed in order to complete the request.
- There are precedence constraints associated with some of the tasks.
- Each task has a fixed duration and also may require certain fixed quantities of specific types of resources.

Putting everything together - a staff scheduling problem

- The resource types are
  - Operator
  - Technician
  - CherryPicker (a type of crane)
  - ISDNPacketMonitor
  - ISDNTechnician.

Tasks to perform – Staff scheduling problem

Task type	Duration	Required resources
MakeAppointment	1	Operator
FlipSwitch	1	Technician
InteriorSiteCall	1	Technician X 2
ISDNInteriorSiteCall	3	Technician ISDNTechnician ISDNPacketMonitor
ExteriorSiteCall	2	Technician x 2 CherryPicker
TestLine	1	Technician

Each task has a fixed duration.
It may also require fixed quantities of specific types of resources.

Precedences – Staff scheduling problem

Request type	Task type	Preceding task
FirstLineInstall	FlipSwitch	-
	TestLine	FlipSwitch
SecondLineInstall	MakeAppointment	-
	InteriorSiteCall	MakeAppointment
	TestLine	InteriorSiteCall
ISDNInstall	MakeAppointment	-
	ISDNInteriorSiteCall ExteriorSiteCall	MakeAppointment
	TestLine	ISDNInteriorSiteCall ExteriorSiteCall

Each request type has a set of task types that must be executed.
Some of the tasks must be executed before other tasks can start.

Putting everything together - a staff scheduling problem

What is the objective?

- In business terms, the objective is “to improve on-time performance.”
- This is a qualitative statement that is difficult to model.
- In order to find a modeling representation (i.e. quantifiable) of the idea, things should be turned around; instead of maximizing something abstract, we find a number that needs to be kept to a minimum.
- Thus, the objective becomes: To *minimize the total number of late days* (days beyond the due date when requests are actually finished).

Putting everything together - a staff scheduling problem

What are the unknowns?

- The unknowns are:
  - When each task will start
  - Which resource will be assigned to each task

Putting everything together - a staff scheduling problem

Modeling alternative resources

- In this problem there is more than one resource, in some cases, capable of doing a given task.

- For example, the task type `FlipSwitch` requires a technician. There are two technicians, that is, two *alternative resources*, available to do the same task.

- The model should optimize how each of these is used relative to the objective.

- The key idea in representing a scheduling problem with alternative resources is:
  - Model each possible task-resource combination with an *optional* interval decision variable.
  - Link these with an interval decision variable that represents the entire task itself (using a `synchronize` constraint).

Modeling resource committed to at most one task at a time

- To constrain a resource so that it cannot be used by more than one task at a given time, you should ensure that there is no overlap in time amongst the intervals associated with a given resource:
  - Create a `sequence` decision variable from those intervals
  - Place a `noOverlap` constraint on the `sequence` decision variable.

Putting everything together - a staff scheduling problem

Surrogate constraints

- At times it is beneficial to introduce additional constraints that improve the CP Optimizer search. Along with treating the resources individually, you can also treat the set of resources of a given type as a *resource pool*. A resource pool can be modeled using a cumulative function expression.

- Each resource type has one `cumulFunction` associated with it. Between the start and end of a task, the `cumulFunction` for any required resource is increased by the number of instances of the resource that the task requires, using the `pulse` function.

- A constraint that the `cumulFunction` never exceeds the number of resources of the given type is added to the model.

- These surrogate constraints on the `cumulFunction` expressions are crucial as they enforce a stronger constraint when the whole set of resources of the tasks is not chosen.

- You'll practice using a surrogate constraint in the last step of the workshop that follows.

Putting everything together - a staff scheduling problem

Lab

Model the staff scheduling problem

- Go to the *Staff Scheduling* workshop and complete all the steps:
  - *Declare task interval and precedences*
  - *Compute the end of a task and define the objective*
  - *Define the resource constraints*
  - *Add a surrogate constraint to accelerate search*

A house building calendar problem

Learning objective

Use IBM® ILOG® CPLEX® CP Optimizer and OPL to model and solve a house building calendar problem.

Sections

- The business problem
- What are the unknowns?
- What are the constraints?
- Modeling the workers' calendars
- Modeling the noOverlap constraint
- Modeling forbidden start/end periods

A house building calendar problem

The business problem

- There are five houses to be built. As usual, some tasks must take place before other tasks, and each task has a predefined size.

- There are two workers, each of whom must perform a given subset of the necessary tasks.

- A worker can be assigned to only one task at a time.

- Each worker has a calendar detailing the days on which he does not work, such as weekends and holidays, with the following constraints:
  - On a worker’s day off, he does no work on his tasks.
  - A worker's tasks may not be scheduled to start or end on a day off.
  - Tasks that are in process by the worker are suspended during his days off.

- The objective is to minimize the overall completion date, that is, the total number of days required to build five houses.

A house building calendar problem

What are the unknowns?

- The unknowns are when each task will start.
- The actual length of a task depends on its position in time and on the calendar of the associated worker.

A house building calendar problem

What are the constraints?

- The constraints specify that:
  - A particular task may not begin until one or more given tasks have been completed
  - A worker can be assigned to only one task at a time
  - Tasks that are in process are suspended during the associated worker’s days off
  - A task cannot start or end during the associated worker’s days off.

A house building calendar problem

Modeling the workers' calendars

- A `stepFunction` is used to model the availability (`intensity`) of a worker with respect to his days off.

- This function has a range of [0..100], where the value 0 represents that the worker is not available and the value 100 represents that the worker is available for a full work period with regard to his calendar.

- While not part of this model, any value in 0..100 can be used as the intensity. For instance, the function could take the value 50 for a time window in which a resource works at half-capacity.

- For each worker, a sorted tuple set is created. At each point in time where the worker’s availability changes, a tuple is created. The tuple has two elements; the first element is an integer value that represents the worker’s availability (0 for on a break, 100 for fully available to work, 50 for a half-day), and the other element represents the date at which the availability changes to this value. This tuple set, sorted by date, is then used to create a `stepFunction` to represent the worker’s intensity over time. The value of the function after the final step is set to 100.

A house building calendar problem

Modeling the noOverlap constraint

- To add the constraints that a worker can perform only one task at a time, the interval variables associated with that worker are constrained to not overlap in the solution using the specialized constraint `noOverlap`:
-
  ```
  forall(w in WorkerNames)
    noOverlap( all(h in Houses, t in TaskNames: Worker[t]==w)
  itvs[h][t]);
  ```

A house building calendar problem

Modeling forbidden start/end periods

- When an intensity function is set on an interval variable, the tasks which overlap weekends and/or holidays will be automatically prolonged.

- An option could be available to start or end a task on a weekend day, but in this problem, a worker’s tasks cannot start or end during the worker’s days off.

- The constraints `forbidStart` and `forbidEnd` respectively constrain an interval variable to not end and not overlap where the associated step function has a zero value.

- You will use these constraints in the next exercise.

A house building calendar problem

Lab

Model the house building calendar problem

- Go to the *House building calendar* workshop and complete all the steps:
  - Read the *Problem description* and *Problem data* sections.
  - *Declare data and decision variables*
  - *Define constraints*

Matters of State: Understanding State Functions

Learning objective

- Learn to use state functions and constraints in a IBM® ILOG® CPLEX® CP Optimizer scheduling model

Matters of State: Understanding State Functions

- In some cases, there may be a restriction on what types of tasks can be processed simultaneously. For instance, in the house building problem, a "clean task" like painting cannot occur at the same time as a "dirty" task like sanding the floors.

- Moreover, some transition may be necessary between intervals with different states, such as needing to wait for the paint to dry before floor sanding can take place.

- This type of situation is called a *state function*. A state function represents the changes in state over time, and can be used to define constraints.

- OPL provides the keyword `stateFunction` to model this.

Matters of State: Understanding State Functions

Syntax:

```
stateFunction <functionName> [with M];
```

- where <functionName> is a label given to the function, and M is an optional transition matrix that needs to be defined as a set of integer triplets (just as for the noOverlap constraint). Thus this matrix is a tuple set.

Matters of State: Understanding State Functions

- For example, for an oven with three possible temperature levels identified by indexes 0, 1 and 2 you could have:
  - `[start=0, end=100): state=0`
  - `[start=150, end=250): state=1`
  - `[start=250, end=300): state=1`
  - `[start=320, end=420): state=2`
  - `[start=460, end=560): state=0,`

Matters of State: Understanding State Functions

To model the oven example:

```
tuple triplet {
      int start;
      int end;
      int state;
};
{ triplet } Transition = ...;
//...
stateFunction ovenTemperature with Transition;
```

Matters of State: Understanding State Functions

State constraints

- You can use constraints to restrict the evolution of a state function. These constraints can specify:
  - That the state of the function must be defined and should remain equal to a given state everywhere over a given fixed or variable interval (`alwaysEqual`).
  - That the state of the function must be defined and should remain constant (no matter its value) everywhere over a given fixed or variable interval (`alwaysConstant`).
  - That intervals requiring the state of the function to be defined cannot overlap a given fixed or variable interval (`alwaysNoState`).
  - That everywhere over a given fixed or variable interval, the state of the function, if defined, must remain within a given range of states `[vmin, vmax] (alwaysIn).`

Matters of State: Understanding State Functions

- Additionally, `alwaysEqual` and `alwaysConstant` can be combined with synchronization constraints to specify that the given fixed or variable interval should have its start and/or end point synchronized with the start and/or end point of the interval of the state function that maintains the required state (notions of start and end alignment).

Applying state functions: a wood cutting problem

Learning objective

- Apply a state function to a real-world scheduling problem, modeled in OPL and solved with IBM® ILOG® CPLEX® CP Optimizer.

Sections

- The business problem
- What is the objective?
- What are the unknowns?
- What are the constraints?

Applying state functions: a wood cutting problem

The business problem

- A wood factory machine cuts stands (processed portions of log) into chips.
- Each stand is characterized by a length, diameter, and species of wood.
-
- The following restrictions apply:
  - The machine can cut a limited number of stands at a time with some restriction on the sum of the diameters that it can accept.
  - The truck fleet can handle a limited number of stands at a given time.
  - Stands processed simultaneously must all be of the same species.
  - Each stand has a fixed delivery date and a processing status of either standard or rushed.
    Any delay on a rush stand will cost a penalty.

Applying state functions: a wood cutting problem

What is the objective?

- In business terms, the objective is to minimize the combined total of:
  - cutting costs (expressed as a function of time spent on the machine)
  - cost for any penalties due to late delivery of rushed order
- In mathematical terms, this objective translates to minimizing the sum of:
  - the product of the maximum cutting time per stand and the cost per time unit
  - the product of the length of rushed stands that are late and the cost per unit of length for being late

Applying state functions: a wood cutting problem

What are the unknowns?

- The unknowns are the completion date of the cutting of the stands.
- An interval variable is associated with each of the stands.
- The size of an interval variable is the product of the length of the stand and the time it takes to cut one unit of the stand's species.

Applying state functions: a wood cutting problem

What are the constraints?

- The constraints are:
  - At a given time, the machine can cut:
    - A limited number of stands
    - A limited sum of stand diameters
    - Only one species.
  - The trucks can carry a limited number of stands.

Applying state functions: a wood cutting problem

Lab

- Go to the *Wood Cutting* workshop and complete all the steps.

Checkpoint

True or false?

1. If a task is completed with some interruptions, its `size` will be greater than its `length`.
2. To model a calendar, define an intensity function for an interval variable, and declare this intensity function by using a `stepFunction`.
3. You must use a `sequence` variable to create a sequence of non-overlapping tasks.

Unit summary

Having completed this lesson you should be able to:

- Explain what an interval variable is.
- Use OPL keyword to express scheduling functions and constraints, including:
  - calendars
  - alternative resources
  - resource pools
  - precedence constraints
  - cumulative functions
  - sequencing constraints
  - no overlap constraints
  - synchronize constraints
  - isomorphism constraint
- Write a simple scheduling model in OPL.

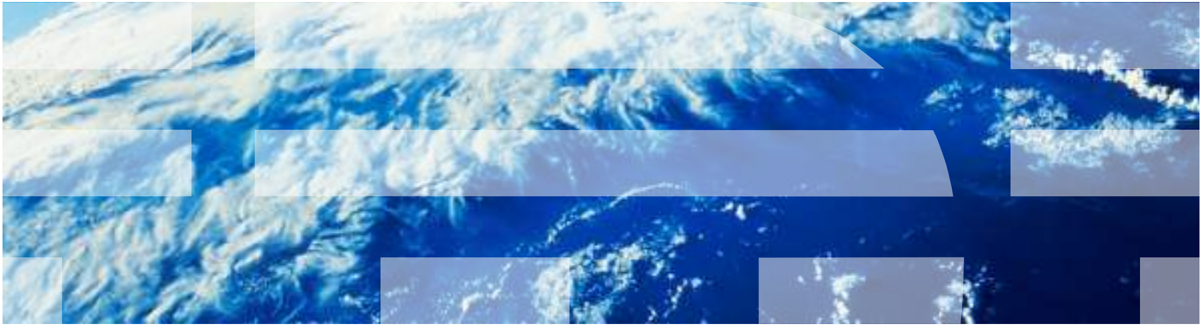
Industry Solutions

Supply Chain Network Design with LogicNet Plus XE

Lesson 9: Scheduling with CP Optimizer

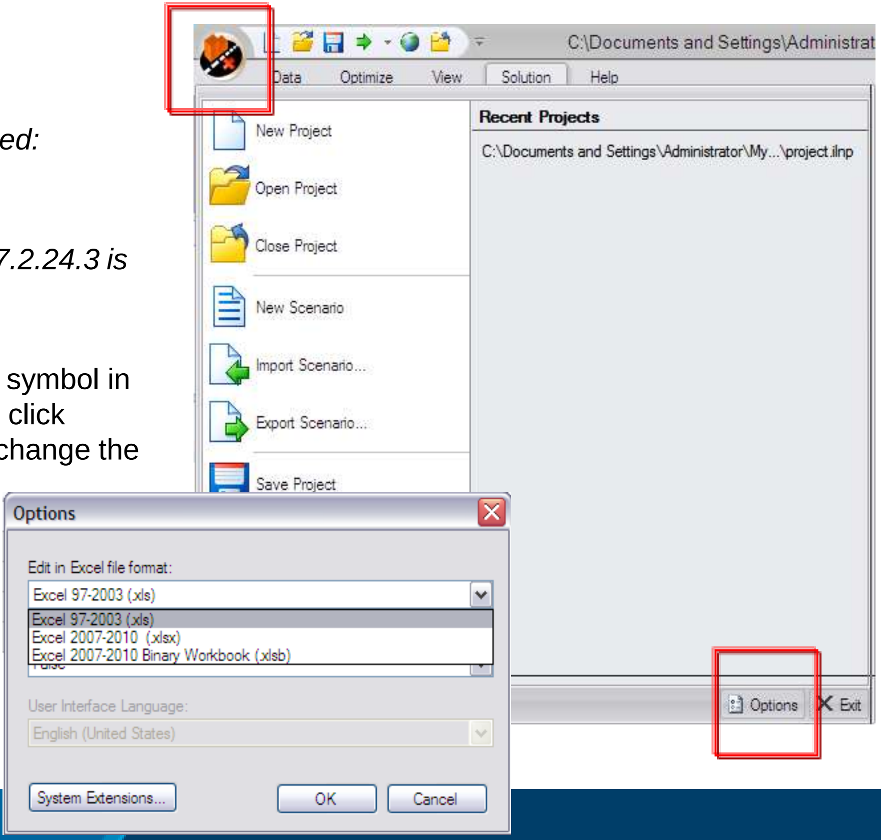
IBM ILOG LogicNet Plus XE 7.2
Self Paced Training Exercises

Introduction to Center of Gravity, Multi-Objective Optimization, and High Service Percentage



Material Needed

- Files needed:
  - *Excel file called: Germany.xls*
- *LNP XE Version 7.2.24.3 is the latest*
- If you click on the symbol in the upper left and click options, you can change the Excel settings



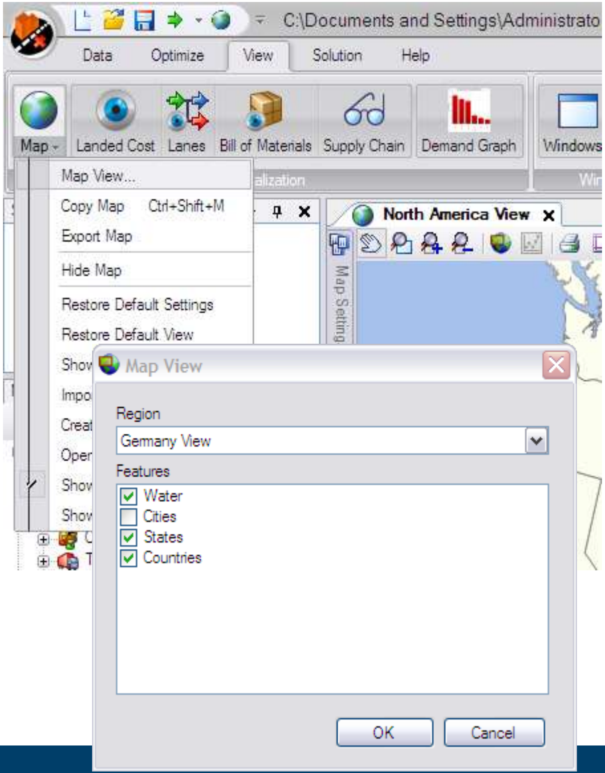
Edit in Excel file format: Excel 97-2003 (xls)

- Excel 97-2003 (xls)
- Excel 2007-2010 (xlsx)
- Excel 2007-2010 Binary Workbook (xlsb)

User Interface Language: English (United States)

Let’s Load Data from Germany’s 200 Largest Cities

- Create a new project
- Change the map view by clicking on the Ribbon for View, then select Map and then select Map View
- Select Germany from the drop down menu



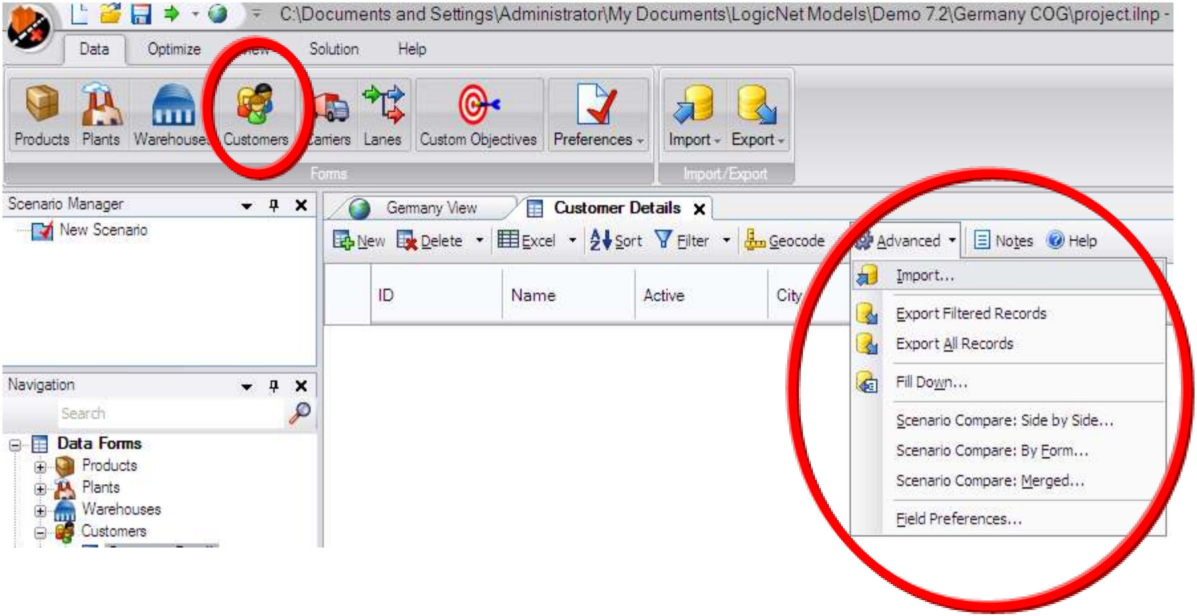
Region

Germany View

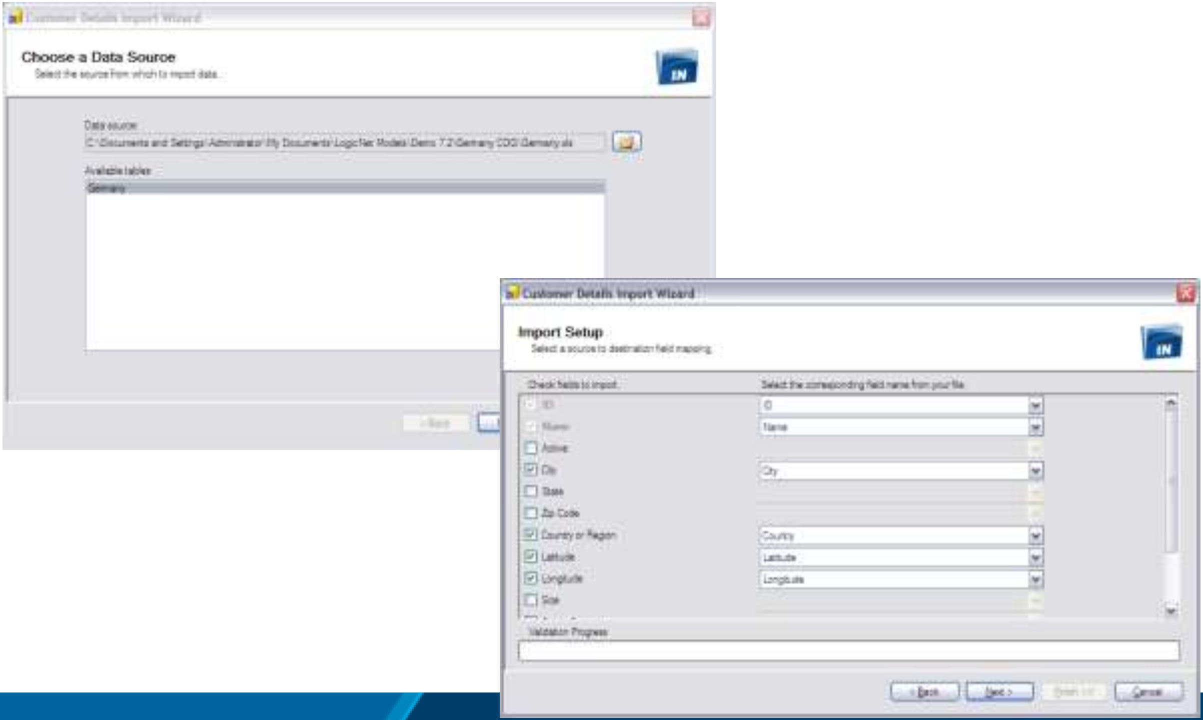
Features

☑ Water
☐ Cities
☑ States
☑ Countries

Open the tab for the customer form and select Import from the advanced tab



Find the folder where you stored the *Germany.xls* file. It only has one tab so select that tab in the "Available tables" section. Then, click next. The fields you need should match up. Click Next and Finish.



Customer Details Import Wizard

Choose a Data Source

Select the source from which to import data.

Data source

C:\Documents and Settings\Administrator\My Documents\LogicNet Models\Demo 7.2\Germany COG\Germany.xls

Available tables

Germany

Customer Details Import Wizard

Import Setup

Select a source to destination field mapping.

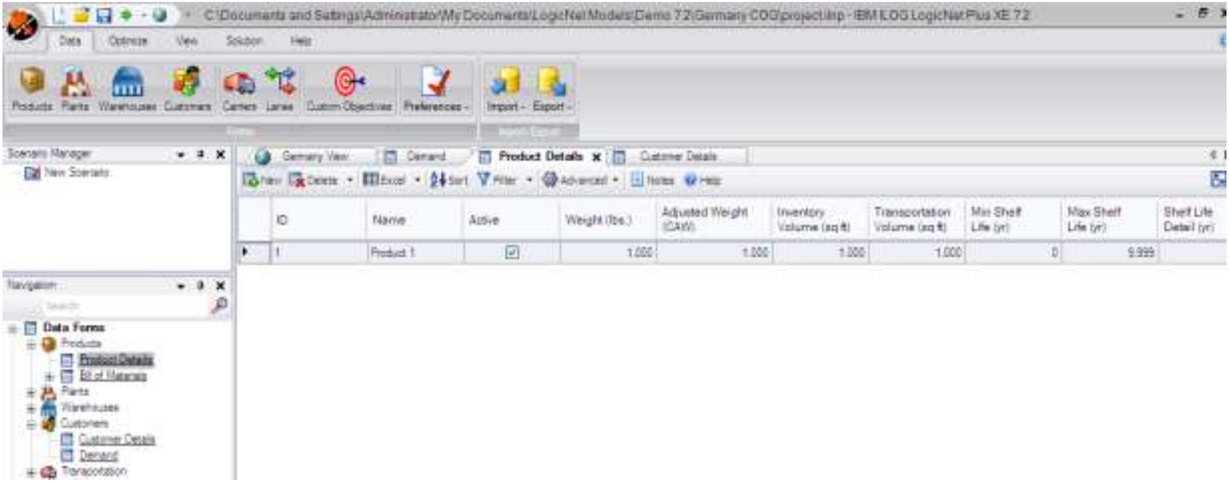
Check fields to import.	Select the corresponding field name from your file.
☑ ID	ID
☑ Name	Name
□ Active	
☑ City	City
□ State	
□ Zip Code	
☑ Country or Region	Country
☑ Latitude	Latitude
☑ Longitude	Longitude
□ Size	

Validation Progress

In the Data Ribbon (or in the Navigation tree), click on Products.

~~Within the products table, click on New to create one product.~~

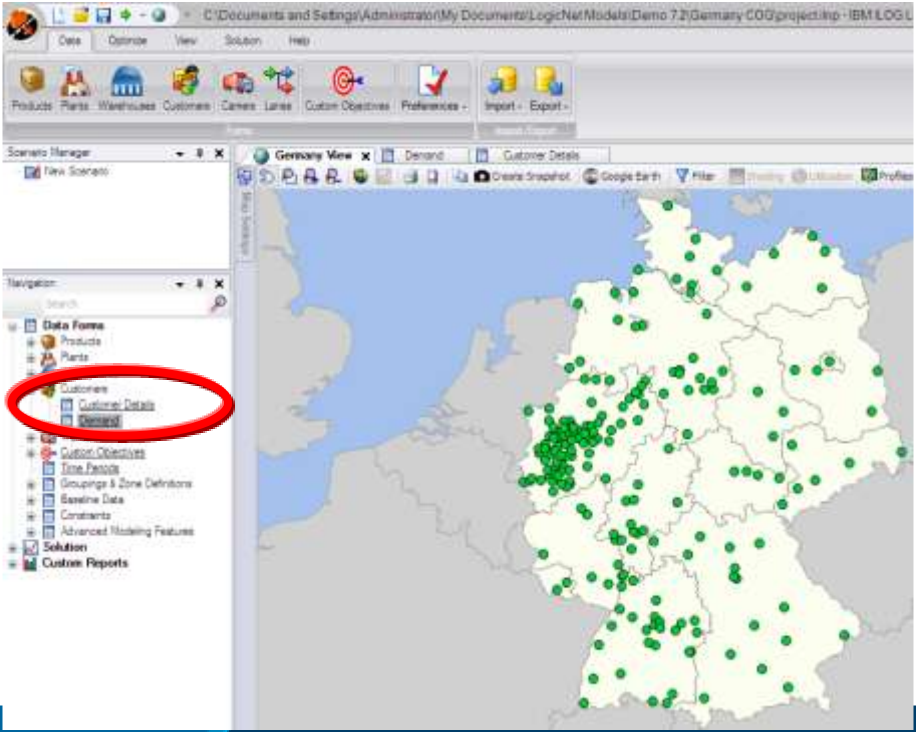
That is all we'll do with products for now.



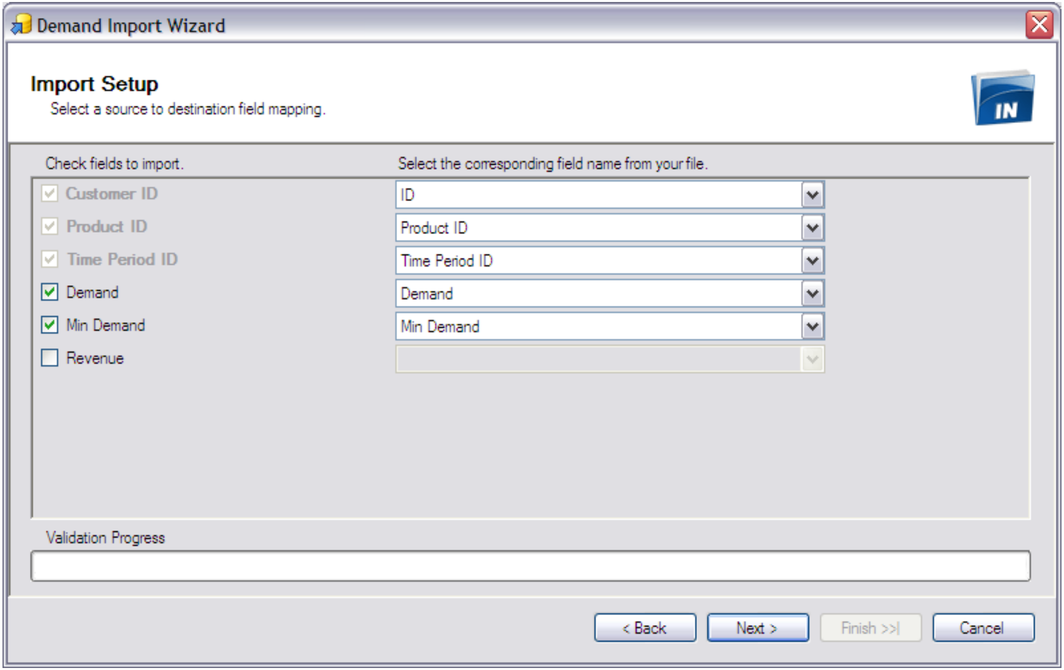
	ID	Name	Active	Weight (lbs.)	Adjusted Weight (CAW)	Inventory Volume (sq ft)	Transportation Volume (sq ft)	Min Shelf Life (yr)	Max Shelf Life (yr)	Shelf Life Detail (yr)
▸	1	Product 1	☑	1.000	1.000	1.000	1.000	0	9.999	

Now, check your work by looking at the map view.

~~Click on the demand table in the menu tree to open the demand form~~



Import the demand data by going to the Demand tab, clicking on Advanced, and then Import. Select the same Germany File and use the default field names that matched as shown below. Click Next and Finish.



Demand Import Wizard

Import Setup

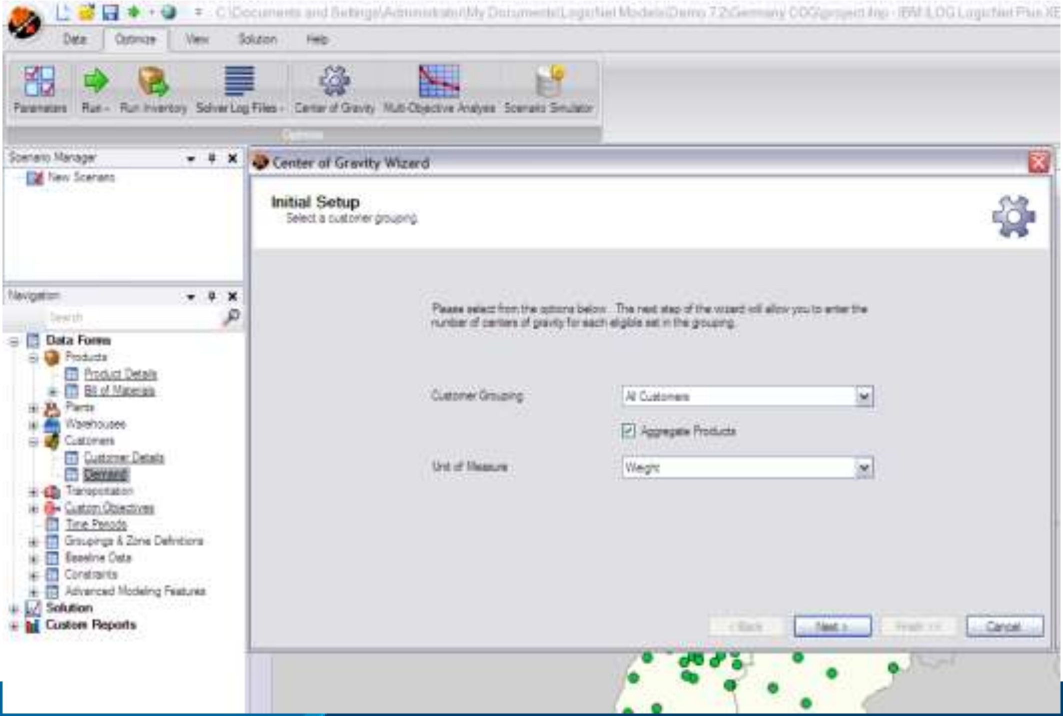
Select a source to destination field mapping.

Check fields to import. | Select the corresponding field name from your file.

Check fields to import.	Select the corresponding field name from your file.
☑ Customer ID	ID
☑ Product ID	Product ID
☑ Time Period ID	Time Period ID
☑ Demand	Demand
☑ Min Demand	Min Demand
□ Revenue	

Validation Progress

We are now ready to start our Center of Gravity Analysis. In the Optimize Ribbon, click on Center of Gravity. Keep the default setting for Initial Setup and click next



Initial Setup

Select a customer grouping

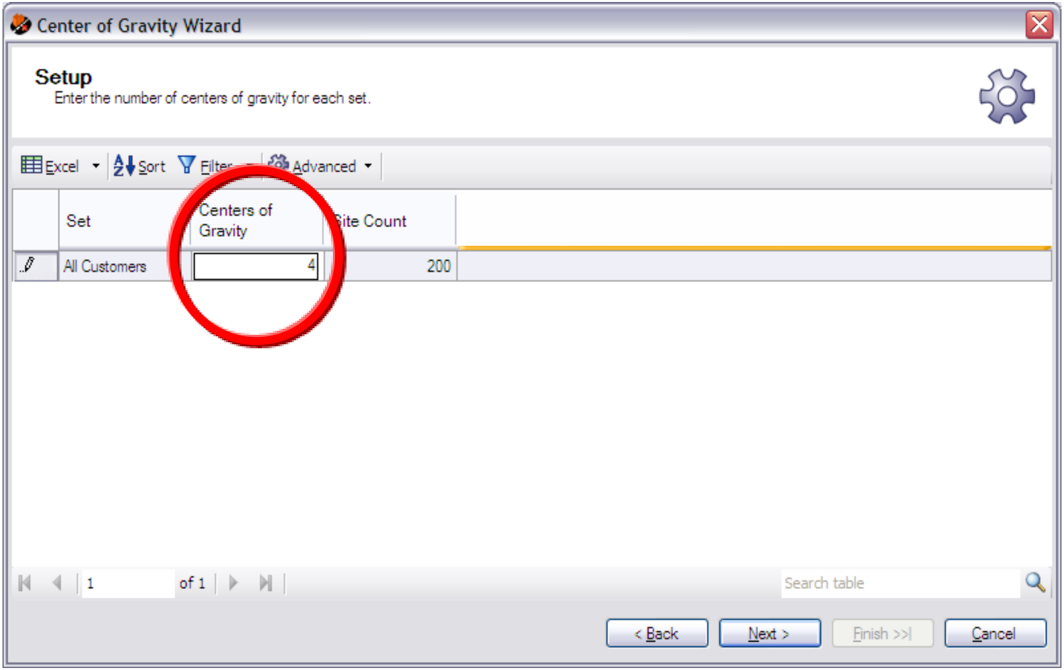
Please select from the options below. The next step of the wizard will allow you to enter the number of centers of gravity for each eligible set in the grouping.

Customer Grouping: All Customers

☑ Aggregate Products

Unit of Measure: Weight

Let's select the best 4 locations to start. Click Next and Finish. It will now generate a new scenario that is ready to run.

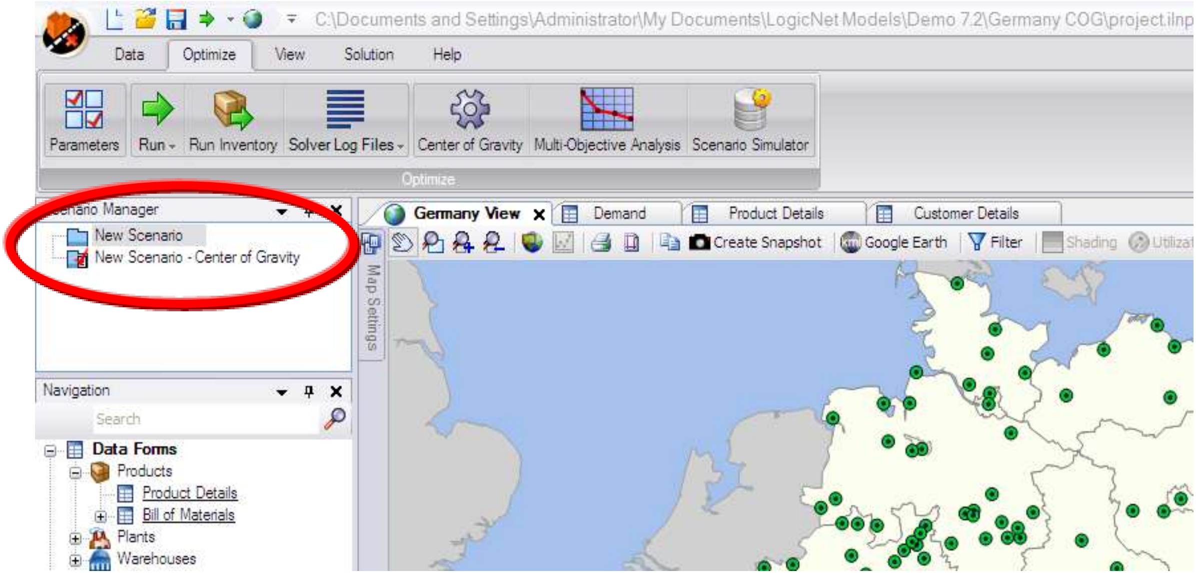


Setup

Enter the number of centers of gravity for each set.

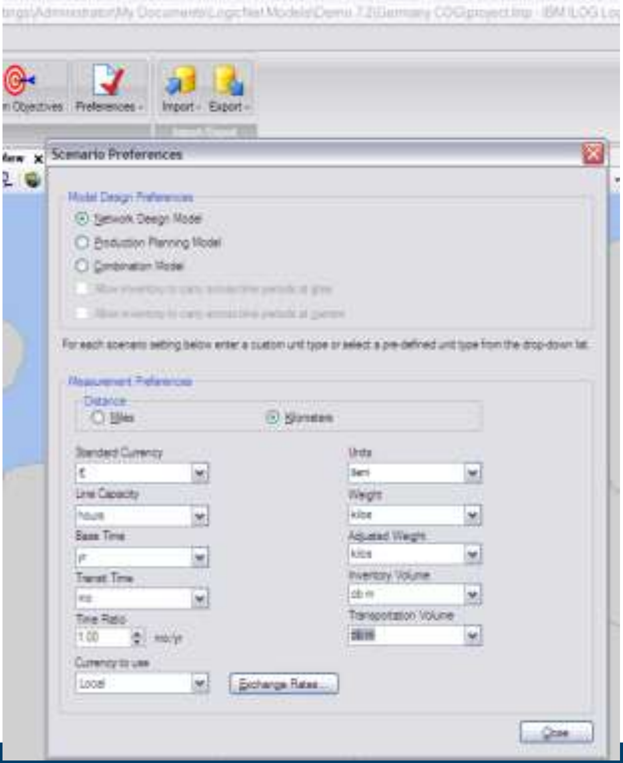
Set	Centers of Gravity	Site Count
All Customers	4	200

Now, move the red check mark to the Center of Gravity Scenario by right clicking on the scenario and selecting “Set as Active.”



Let’s Set the Right Preferences. In the data ribbon, click on Preferences and Select Scenario Preferences.

- Change currency to Euros
- Change Weight to Kilos
- Change Adjusted Weight to Kilos by typing it in the box
- Change the Inventory and Transportation Volume to “cb m”



Scenario Preferences

Model Design Preferences

● Network Design Model
○ Production Planning Model
○ Combination Model
□ Allow inventory to carry across time periods at [illegible]
□ Allow inventory to carry across time periods at [illegible]

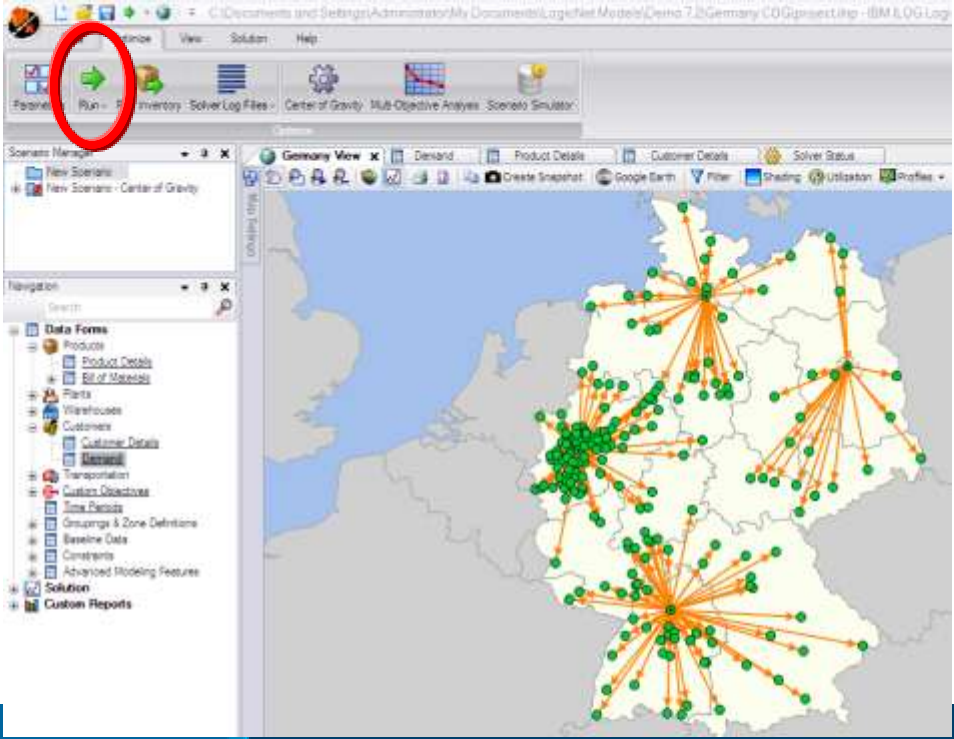
For each scenario setting below enter a custom unit type or select a pre-defined unit type from the drop-down list.

Measurement Preferences

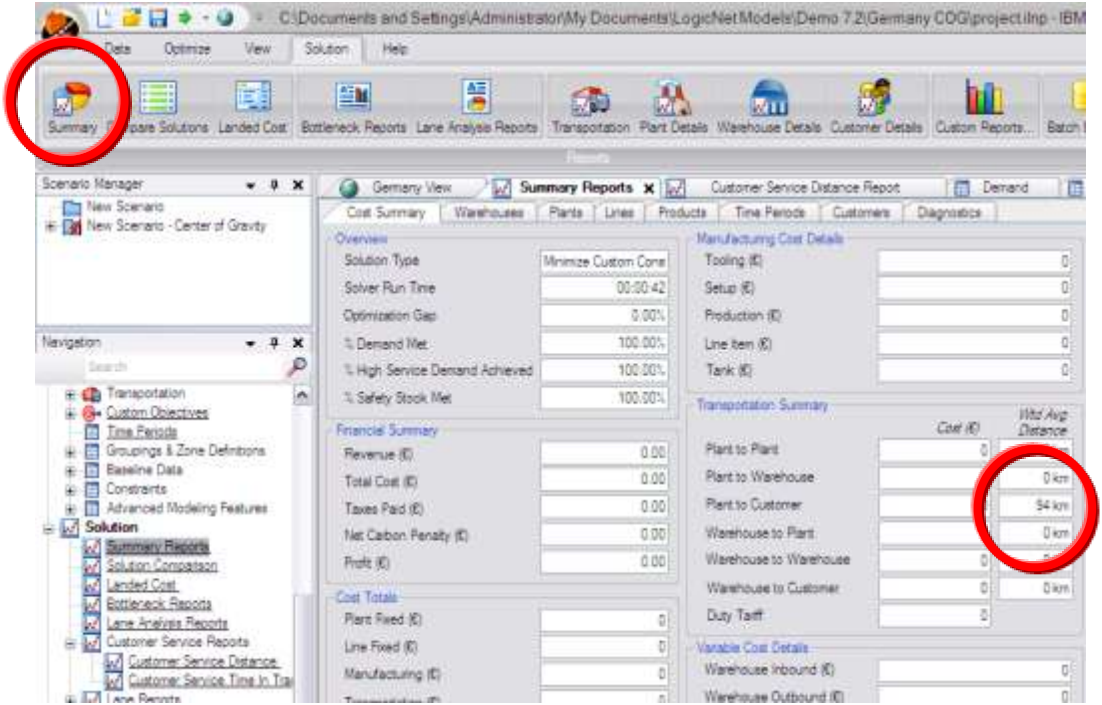
Distance: ○ Miles ● Kilometers

Standard Currency: €
Line Capacity: hours
Base Time: yr
Transit Time: hrs
Time Ratio: 1.00 hrs/yr
Currency to use: Local
Units: item
Weight: kilos
Adjusted Weight: kilos
Inventory Volume: cb m
Transportation Volume: [illegible]

Go to the Optimize window and click on Run. Once the run is complete, you will see the map of the solution



By definition, the Center of Gravity models run with a minimum set of data. We can fill more in later, but the nice thing is that we get some information quickly. Go to the Solution Ribbon and click on Summary to see the Weighted Average Distance that we minimized



Overview

Solution Type	Minimize Custom Cons
Solver Run Time	00:00:42
Optimization Gap	0.00%
% Demand Met	100.00%
% High Service Demand Achieved	100.00%
% Safety Stock Met	100.00%

Financial Summary

Revenue (€)	0.00
Total Cost (€)	0.00
Taxes Paid (€)	0.00
Net Carbon Penalty (€)	0.00
Profit (€)	0.00

Cost Totals

Plant Fixed (€)	0
Line Fixed (€)	0
Manufacturing (€)	0
Transportation (€)	0

Manufacturing Cost Details

Tooling (€)	0
Setup (€)	0
Production (€)	0
Line Item (€)	0
Tank (€)	0

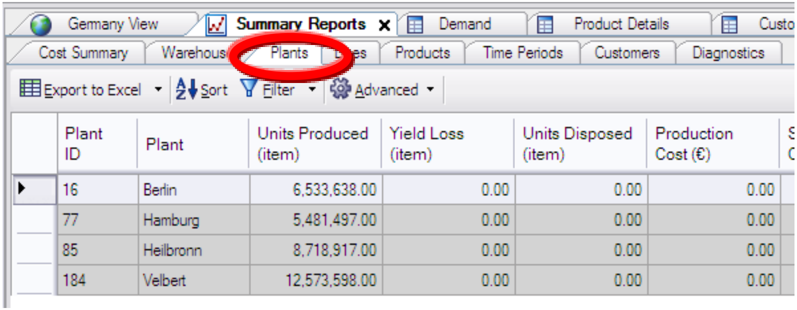
Transportation Summary

	Cost (€)	Wtd Avg Distance
Plant to Plant	0	[illegible]
Plant to Warehouse	[illegible]	0 km
Plant to Customer	[illegible]	54 km
Warehouse to Plant	[illegible]	0 km
Warehouse to Warehouse	0	[illegible]
Warehouse to Customer	0	0 km
Duty Tariff	0	

Variable Cost Details

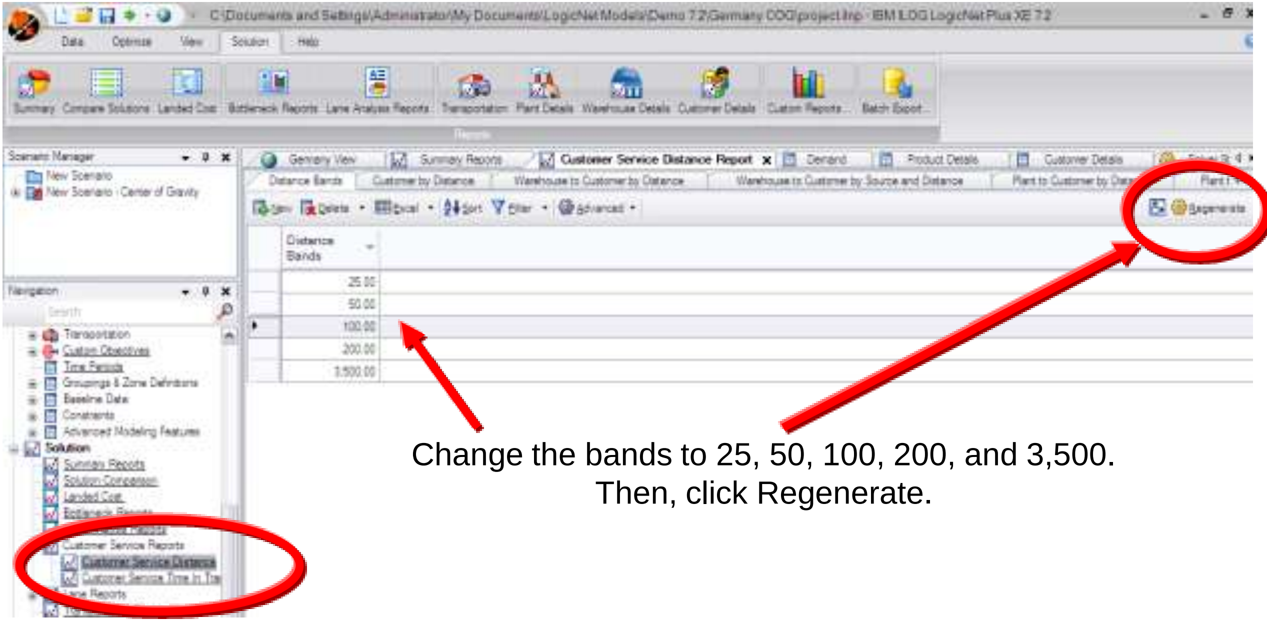
Warehouse Inbound (€)	0
Warehouse Outbound (€)	0

If you go to the Plant tab on the same summary report, you will see which cities it picked



Plant ID	Plant	Units Produced (item)	Yield Loss (item)	Units Disposed (item)	Production Cost (€)
16	Berlin	6,533,638.00	0.00	0.00	0.00
77	Hamburg	5,481,497.00	0.00	0.00	0.00
85	Heilbronn	8,718,917.00	0.00	0.00	0.00
184	Velbert	12,573,598.00	0.00	0.00	0.00

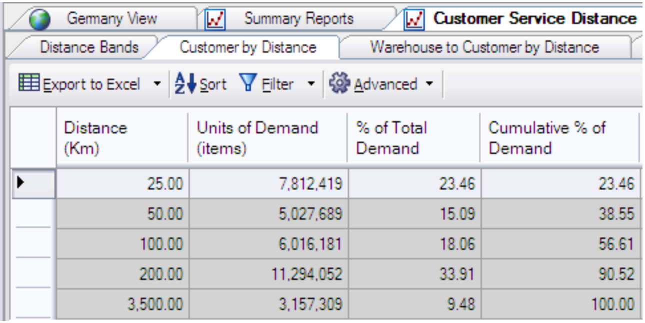
Another nice report for Center of Gravity Studies is the Customer Service Report. In the Navigation Tree, go to Solutions | Customer Service Reports | Customer Services Distance.



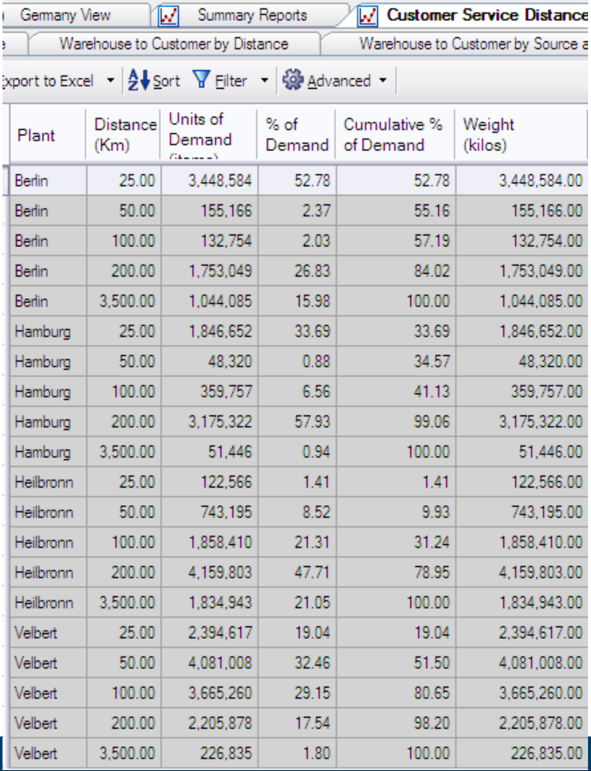
Distance Bands
25.00
50.00
100.00
200.00
3,500.00

Change the bands to 25, 50, 100, 200, and 3,500. Then, click Regenerate.

The two most interesting tabs in this report are the “Customer by Distance and the “Plant to Customer by Source and Destination” reports.



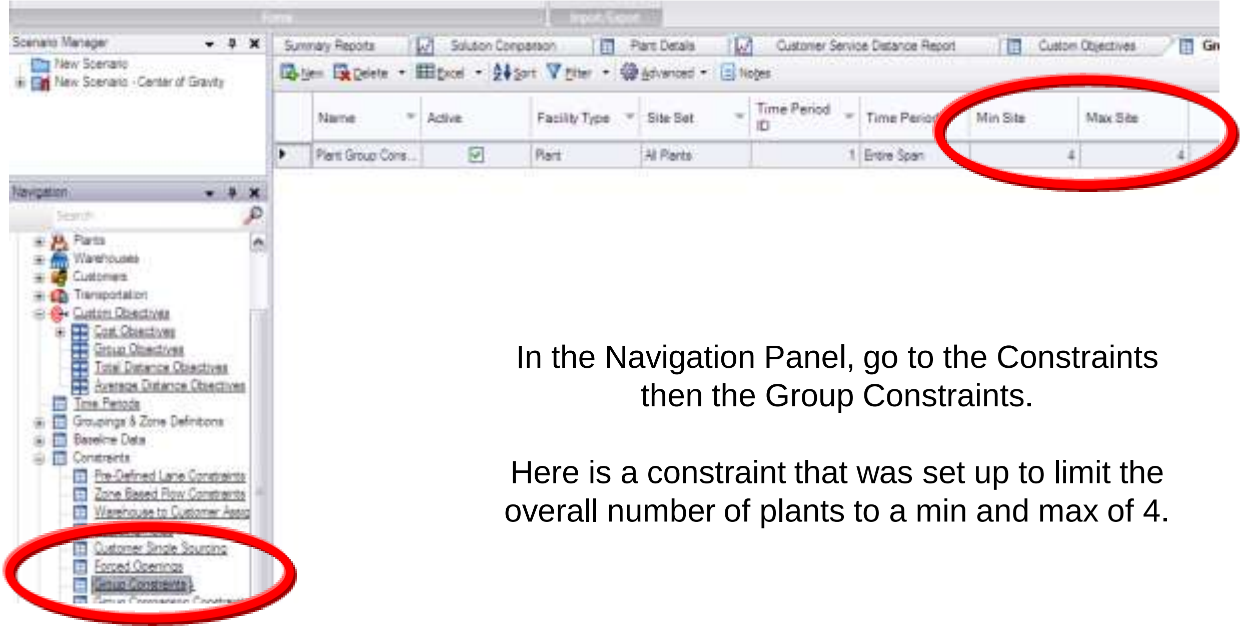
Distance (Km)	Units of Demand (items)	% of Total Demand	Cumulative % of Demand
25.00	7,812,419	23.46	23.46
50.00	5,027,689	15.09	38.55
100.00	6,016,181	18.06	56.61
200.00	11,294,052	33.91	90.52
3,500.00	3,157,309	9.48	100.00


Germany View
Summary Reports
Customer Service Distance
Warehouse to Customer by Distance
Warehouse to Customer by Source a
xport to Excel
Sort
Filter
Advanced

Plant	Distance (Km)	Units of Demand	% of Demand	Cumulative % of Demand	Weight (kilos)
Berlin	25.00	3,448,584	52.78	52.78	3,448,584.00
Berlin	50.00	155,166	2.37	55.16	155,166.00
Berlin	100.00	132,754	2.03	57.19	132,754.00
Berlin	200.00	1,753,049	26.83	84.02	1,753,049.00
Berlin	3,500.00	1,044,085	15.98	100.00	1,044,085.00
Hamburg	25.00	1,846,652	33.69	33.69	1,846,652.00
Hamburg	50.00	48,320	0.88	34.57	48,320.00
Hamburg	100.00	359,757	6.56	41.13	359,757.00
Hamburg	200.00	3,175,322	57.93	99.06	3,175,322.00
Hamburg	3,500.00	51,446	0.94	100.00	51,446.00
Heilbronn	25.00	122,566	1.41	1.41	122,566.00
Heilbronn	50.00	743,195	8.52	9.93	743,195.00
Heilbronn	100.00	1,858,410	21.31	31.24	1,858,410.00
Heilbronn	200.00	4,159,803	47.71	78.95	4,159,803.00
Heilbronn	3,500.00	1,834,943	21.05	100.00	1,834,943.00
Velbert	25.00	2,394,617	19.04	19.04	2,394,617.00
Velbert	50.00	4,081,008	32.46	51.50	4,081,008.00
Velbert	100.00	3,665,260	29.15	80.65	3,665,260.00
Velbert	200.00	2,205,878	17.54	98.20	2,205,878.00
Velbert	3,500.00	226,835	1.80	100.00	226,835.00

Let's see how the model is set up….

First, we'll start with the constraint we put in of 4 sites (we modeled the sites as plants)

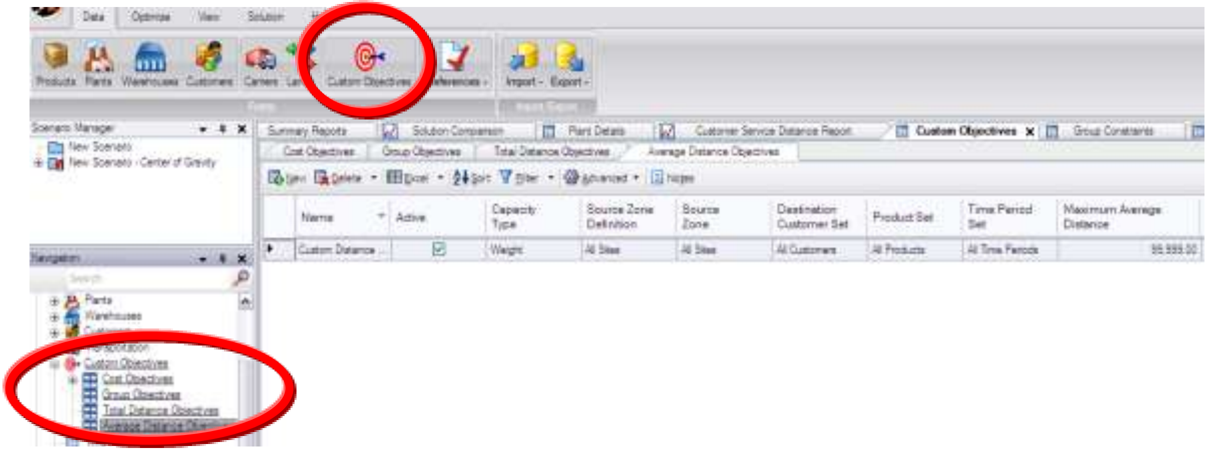


Name	Active	Facility Type	Site Set	Time Period ID	Time Period	Min Site	Max Site
Plant Group Cons...	☑	Plant	All Plants	1	Entire Span	4	4

In the Navigation Panel, go to the Constraints then the Group Constraints.

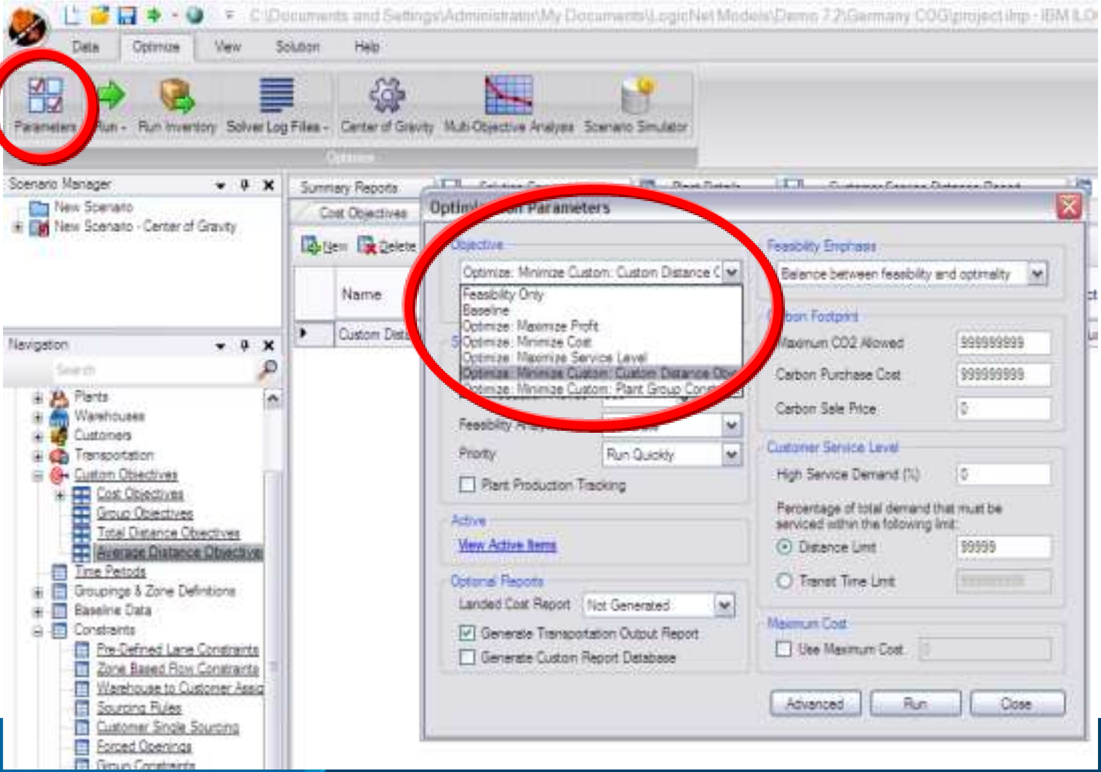
Here is a constraint that was set up to limit the overall number of plants to a min and max of 4.

Now go to the Custom Objectives and the Average Distance Objective. Here, an objective was automatically set up for us. But, as you can see, we are free to create as many custom objectives as we want.



Name	Active	Capacity Type	Source Zone Definition	Source Zone	Destination Customer Set	Product Set	Time Period Set	Maximum Average Distance
Custom Distance ...	☑	Weight	All Sites	All Sites	All Customers	All Products	All Time Periods	99,999.00

Then, we need to guide the optimization to use the custom objective. Go to the Optimize Ribbon and the Parameters form. From the drop down menu, you can see all the objectives we have set up (and the standard ones)



C:\Documents and Settings\Administrator\My Documents\LogicNet Models\Demo 7.2\Germany COG\project.lnp - IBM ILOG

Optimization Parameters

Objective

Optimize: Minimize Custom: Custom Distance O

- Feasibility Only
- Baseline
- Optimize: Maximize Profit
- Optimize: Minimize Cost
- Optimize: Maximize Service Level
- Optimize: Minimize Custom: Custom Distance Obj
- Optimize: Minimize Custom: Plant Group Cons

Feasibility Analysis

Priority: Run Quickly

□ Plant Production Tracking

Active

View Active Items

Optional Reports

Landed Cost Report: Not Generated

☑ Generate Transportation Output Report

□ Generate Custom Report Database

Feasibility Emphasis

Balance between feasibility and optimality

Carbon Footprint

Maximum CO2 Allowed: 999999999

Carbon Purchase Cost: 999999999

Carbon Sale Price: 0

Customer Service Level

High Service Demand (%): 0

Percentage of total demand that must be serviced within the following limit:

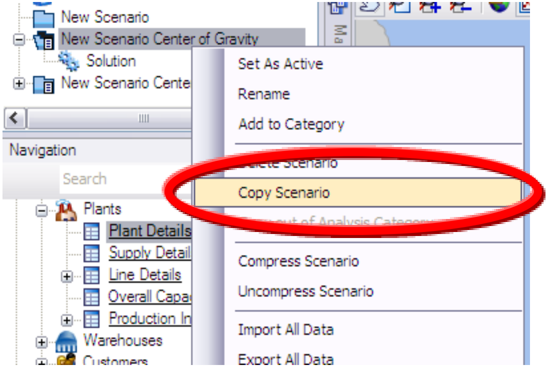
● Distance Limit: 99999

○ Transit Time Limit

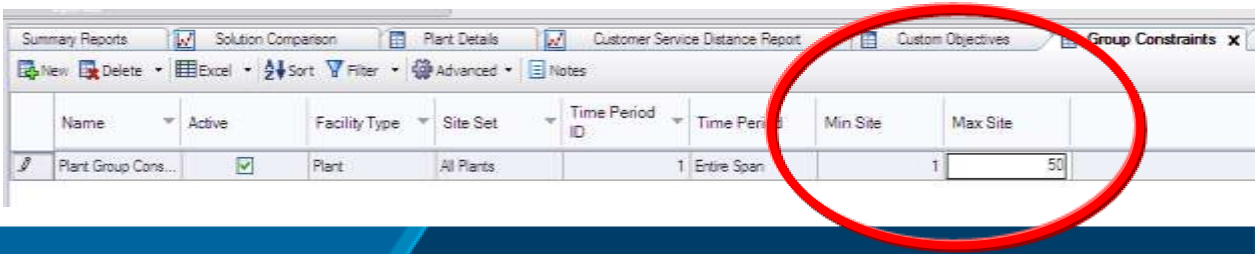
Maximum Cost

□ Use Maximum Cost

We know the Average Distance is 94km with 4 facilities. Let's use the Multi-~~Objective Optimization to Determine the trade-off between number of facilities~~ and the weighted average distance.

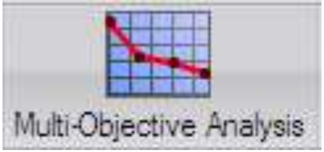


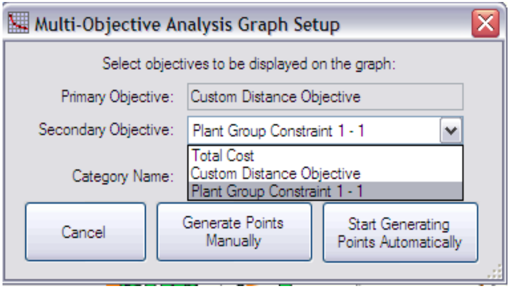
To get ready to run a multiple-objective optimization, make a copy of the scenario we just ran. Set the copy to active. Then, in the group constraints, set the min-max sites to 1 to 50. This will allow the multi-objective to have a good range of choices. Then run.



Name	Active	Facility Type	Site Set	Time Period ID	Time Period	Min Site	Max Site
Plant Group Cons...	☑	Plant	All Plants	1	Entire Span	1	50

Once the model is complete, click on the Multi-Objective Analysis button in the Optimize Ribbon. The primary objective is the one you ran with during the initial optimization. It will ask you to select the secondary objective. Select the Plant Group Constraint. If you want to specify the category name, type in the name you want in that field. Then, click “Start Generating Points Automatically”




Multi-Objective Analysis Graph Setup

Select objectives to be displayed on the graph:

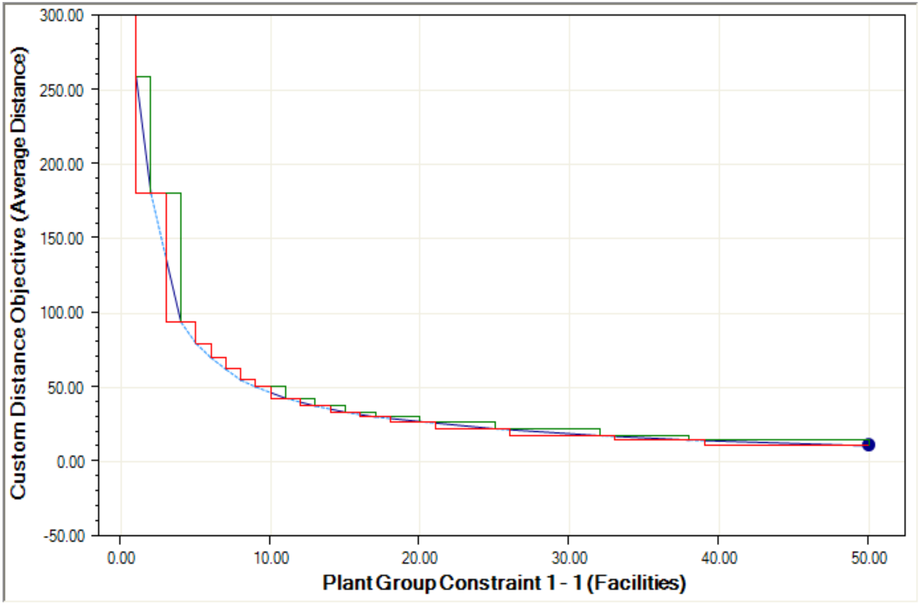
Primary Objective: Custom Distance Objective

Secondary Objective: Plant Group Constraint 1 - 1

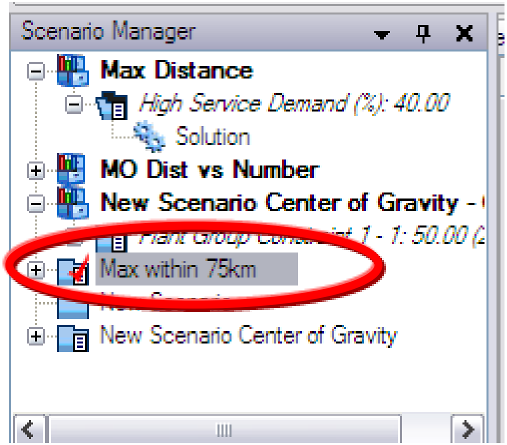
- Total Cost
- Custom Distance Objective
- Plant Group Constraint 1 - 1

Category Name:

You can then watch the optimization automatically draw this graph. This now shows you the complete trade-off between the number of facilities and average distance to customers.

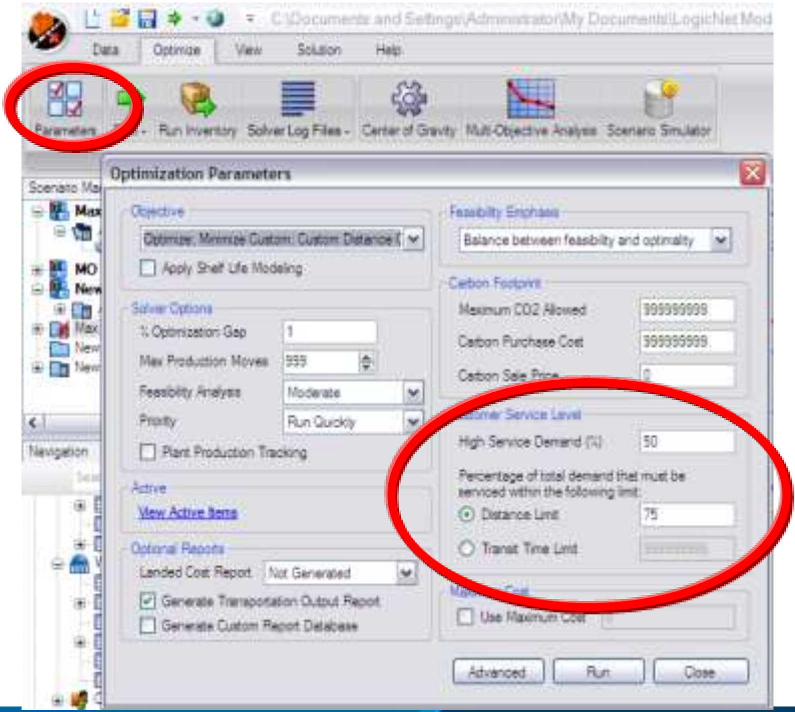


Let’s now look at maximizing the demand within a certain distance band. This is another important measure of service. It is a way to maximize the number of customers you can reach in 4 hours, 1 day, and so on. For the example in Germany, we want to maximize the demand within 75 km.


Scenario Manager
Max Distance
High Service Demand (%): 40.00
Solution
MO Dist vs Number
New Scenario Center of Gravity -
Plant Group Constraint 1 - 1: 50.00 (2
Max within 75km
New Scenario
New Scenario Center of Gravity

Right-click on the “New Scenario Center of Gravity” and make a copy of the scenario. Rename the new scenario to “Max within 75km” and set it as active. Active scenarios have a red check mark next to them.

First, we will set up a constraint that specifies the minimum amount of demand that must be satisfied within a certain distance.



Optimization Parameters

Objective
Optimize: Minimize Custom: Custom Distance (
□ Apply Shelf Life Modeling

Solver Options
% Optimization Gap: 1
Max Production Moves: 999
Feasibility Analysis: Moderate
Priority: Run Quickly
□ Plant Production Tracking

Active
View Active Items

Optional Reports
Landed Cost Report: Not Generated
☑ Generate Transportation Output Report
□ Generate Custom Report Database

Feasibility Emphasis
Balance between feasibility and optimality

Carbon Footprint
Maximum CO2 Allowed: 999999999
Carbon Purchase Cost: 999999999
Carbon Sale Price: 0

Customer Service Level
High Service Demand (%): 50
Percentage of total demand that must be serviced within the following limit:
● Distance Limit: 75
○ Transit Time Limit: 99999999

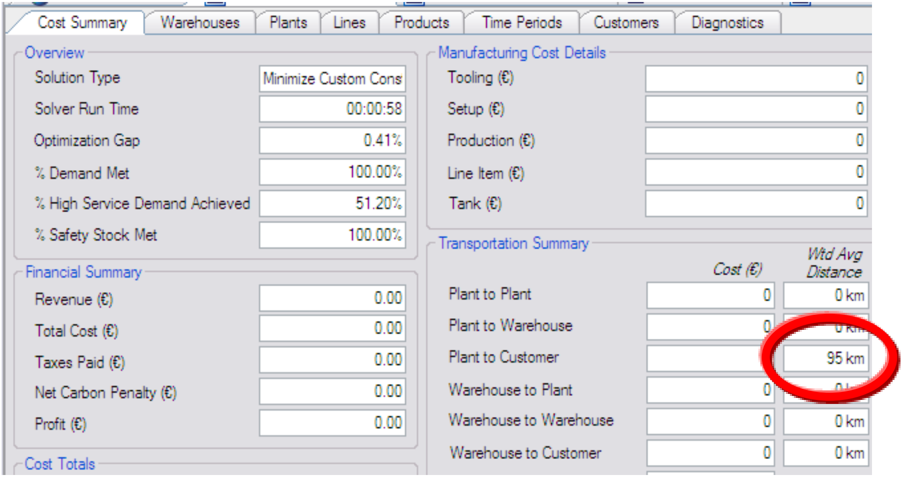
Maximum Cost
□ Use Maximum Cost

We are going to specify that 40% of the demand must be within 75km. To do this, go to the Optimize Ribbon, click on the parameters button and then set the High Service Demand (%) to 50 and the Distance Limit to 75.

We are still running with just 4 sites and an objective to minimize average weighted distance.

Now click "Run"

To see what is different, we can go to the Solution | Summary report. We first note that the average distance went up from 94 to 95 (not that big of a change.



Overview

Solution Type	Minimize Custom Cons
Solver Run Time	00:00:58
Optimization Gap	0.41%
% Demand Met	100.00%
% High Service Demand Achieved	51.20%
% Safety Stock Met	100.00%

Financial Summary

Revenue (€)	0.00
Total Cost (€)	0.00
Taxes Paid (€)	0.00
Net Carbon Penalty (€)	0.00
Profit (€)	0.00

Cost Totals

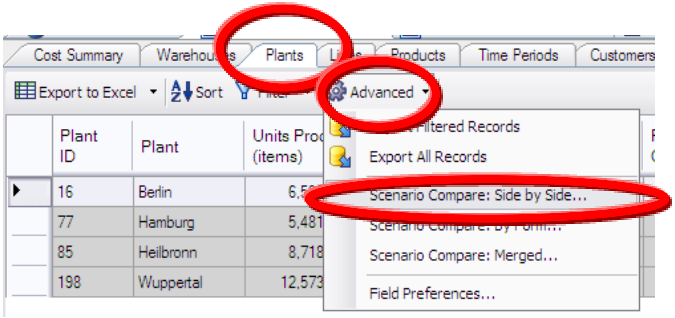
Manufacturing Cost Details

Tooling (€)	0
Setup (€)	0
Production (€)	0
Line Item (€)	0
Tank (€)	0

Transportation Summary

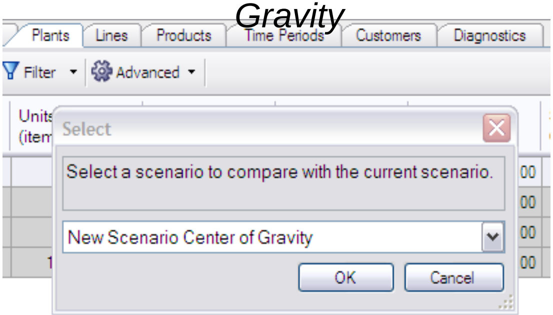
	Cost (€)	Wtd Avg Distance
Plant to Plant	0	0 km
Plant to Warehouse	0	0 km
Plant to Customer		95 km
Warehouse to Plant	0	0
Warehouse to Warehouse	0	0 km
Warehouse to Customer	0	0 km

To see which locations are different go to the Plants tab in the summary ~~report, then click on Advanced and select “Scenario Compare: Side by Side”~~



Plant ID	Plant	Units Prod (items)
16	Berlin	6,5
77	Hamburg	5,481
85	Heilbronn	8,718
198	Wuppertal	12,573

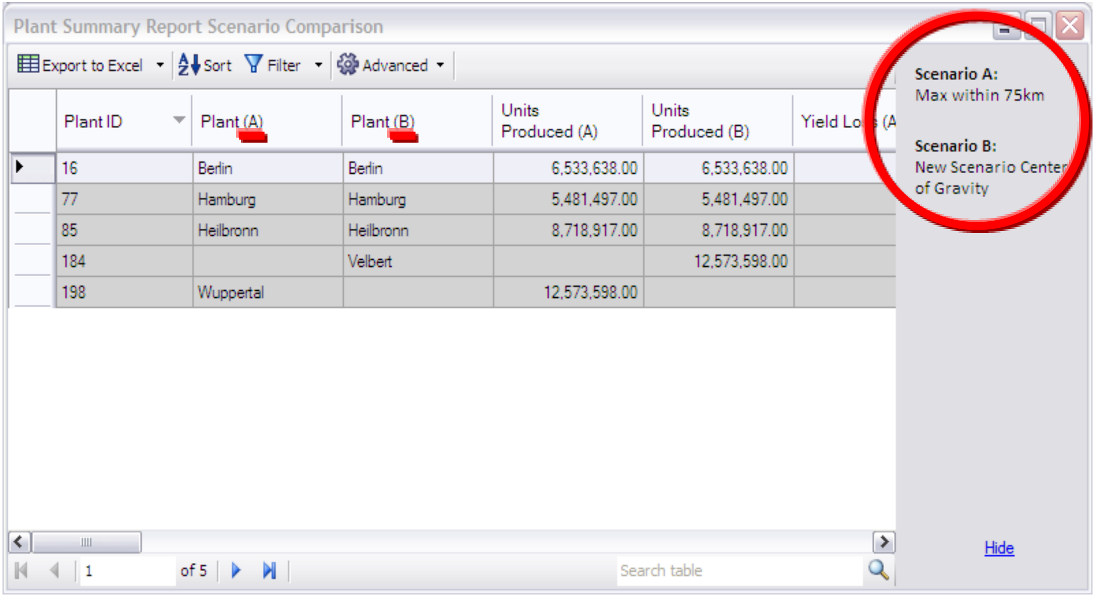
Then, select the original scenario to compare– *New Scenario Center of Gravity*



Select a scenario to compare with the current scenario.

New Scenario Center of Gravity

Now, the report that comes up shows how the solutions differ. In this case, the first three sites stay the same. The 4th site switches locations. The legend on the shows scenario A and B.

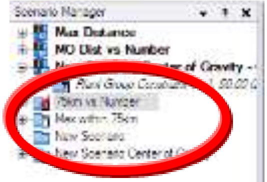


Plant ID	Plant (A)	Plant (B)	Units Produced (A)	Units Produced (B)	Yield Lo (A
16	Berlin	Berlin	6,533,638.00	6,533,638.00	
77	Hamburg	Hamburg	5,481,497.00	5,481,497.00	
85	Heilbronn	Heilbronn	8,718,917.00	8,718,917.00	
184		Velbert		12,573,598.00	
198	Wuppertal		12,573,598.00		

Scenario A:
Max within 75km

Scenario B:
New Scenario Center of Gravity

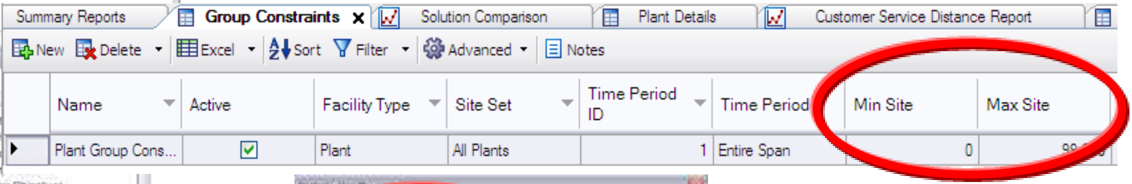
Instead of setting a constraint for demand within 75km, let's do a multi-objective optimization to determine the trade-off between number of sites and the percent with 75km Here is how we set up the model.



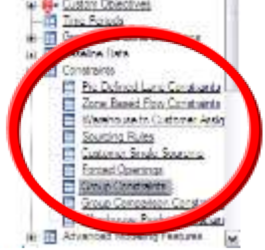
Scenario Manager

- Max Distance
- MO Dist vs Number
- New Scenario Center of Gravity - ...
  - *Plant Group Constraints ... 1, 99,999 ...*
- 75km vs Number
- Max within 75km
- New Scenario
- New Scenario Center of G...

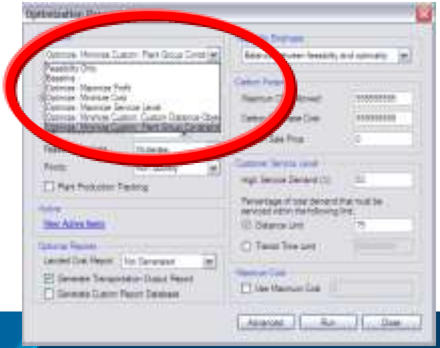
First, copy the scenario *Max within 75km*, set the new scenario as active, and rename to *75km vs Number.* Then, go to the Group Constraints and change the min to 0 and the max to 99,999.



	Name	Active	Facility Type	Site Set	Time Period ID	Time Period	Min Site	Max Site
▶	Plant Group Cons...	☑	Plant	All Plants	1	Entire Span	0	99,999



Custom Objectives
Time Periods
Constraints
Pre-Defined Lane Constraints
Zone Based Flow Constraints
Warehouse to Customer Assignment
Sourcing Rules
Customer Single Sourcing
Forced Openings
Group Constraints
Group Conversion Constraints
Advanced Modeling Features



Optimization Parameters

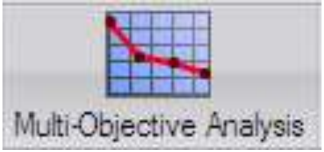
Optimize: Minimize Custom: Plant Group Constr...

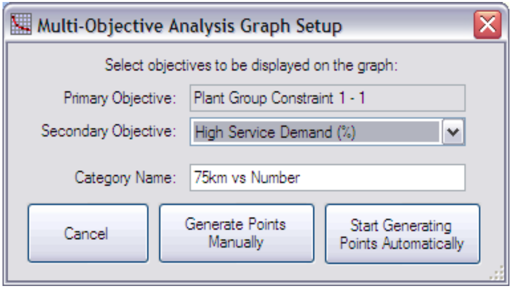
- Feasibility Only
- Baseline
- Optimize : Maximize Profit
- Optimize : Minimize Cost
- Optimize : Maximize Service Level
- Optimize: Minimize Custom: Custom Distance-Obje...
- Optimize: Minimize Custom: Plant Group Constraints

Third, go to the optimization parameters form and select the Minimize Custom: Plant Group as the objective.

Hit Run.

Now, the model is ready to run in Multi-Objective Mode. Click on Multi-Objective Analysis. Set "High Service Demand (%)" as the Secondary Objective and click "Start Generating Points Automatically"




Multi-Objective Analysis Graph Setup

Select objectives to be displayed on the graph:

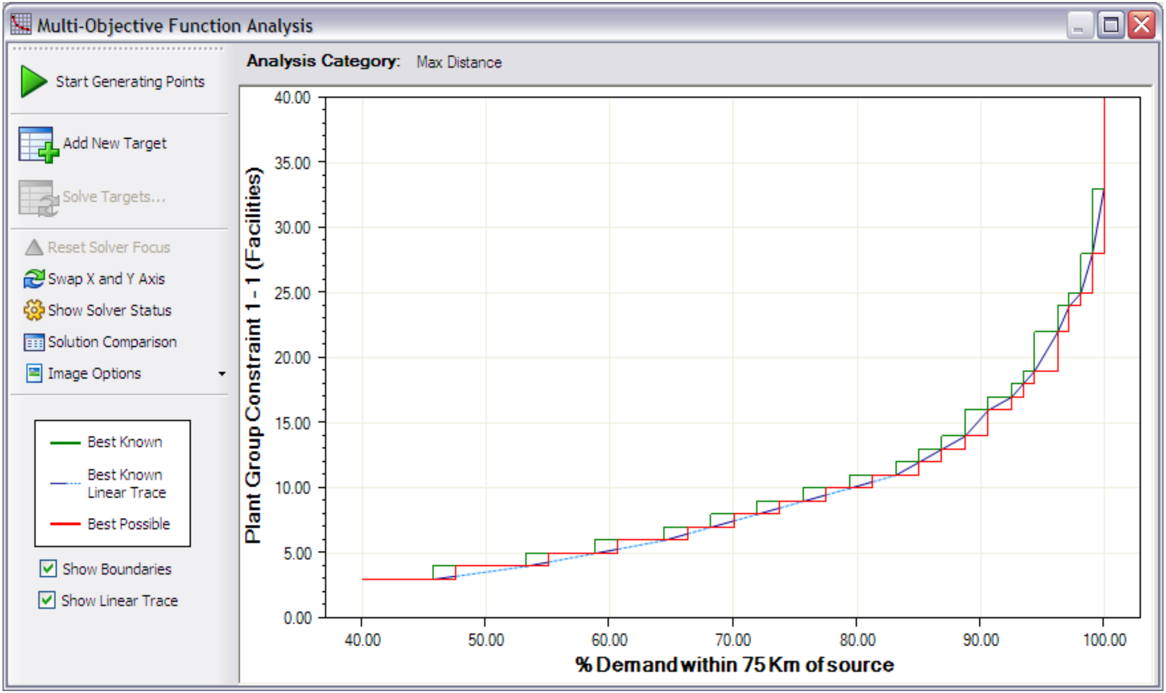
Primary Objective: Plant Group Constraint 1 - 1

Secondary Objective: High Service Demand (%)

Category Name: 75km vs Number

Cancel
Generate Points Manually
Start Generating Points Automatically

Here is the trade-off between number of facilities and % of Demand with 75 KM.



For more information on Multi-Objective Optimization see the October 2011 Virtual Users Group File:

IBM ILOG LNP XE 7.2 Oct Virtual Users' Group v1.pdf