 Previous: (5g) Column Generation

(5h) Column Generation 2

All LP's we have solved so far have been using **row-wise** modelling: the variables have been created and given bounds, the complete objective function has been entered on one line, and each of the constraints has been entered one line at a time (sometimes in a *for* loop) stating how the pre-created variables must relate to each other.
This means that it is easy to add a new constraint to the existing `prob` object, as was done in the Sudoku Problem. However, when a new variable (a pattern in the Cutting Stock Problem) is added, a new `prob` object must be created so that the different objective function and different constraints can be added. This was done on the first Column Generation page, but the entire process can be made much more efficient by using **column-wise modelling**.
In **column-wise modelling**, the problem data is added as columns (one column would be all the coefficients on any particular variable, across the objective function and all the constraints). This makes it much easier to add more variables.

A simple example of the familiar row-wise modelling:

```
from pulp import *
prob = LpProblem("test", LpMinimize)
x = LpVariable("x", 0, 4)
y = LpVariable("y", -1, 1)
z = LpVariable("z", 0)
prob += x + 4*y + 9*z, "obj"
prob += x+y <= 5, "c1"
prob += x+z >= 10, "c2"
prob += -y+z == 7, "c3"
prob.solve()
```

This same problem can be modelled using column-wise modelling:

```
from pulp import *
prob = LpProblem("test", LpMinimize)




obj = LpConstraintVar("obj")
prob.setObjective(obj)      # the obj variable is initialised in this standard way


a = LpConstraintVar("Ca", LpConstraintLE, 5)    # each constraint is created, but only the sign and R.H.S are specified
b = LpConstraintVar("Cb", LpConstraintGE, 10)
c = LpConstraintVar("Cc", LpConstraintEQ, 7)


prob += a   # each constraint is added to prob
prob += b
prob += c


x = LpVariable("x", 0, 4, LpContinuous, obj + a + b)      # each variable is added with it's bounds, category/type,
y = LpVariable("y", -1, 1, LpContinuous, 4*obj + a - c)   # and it's coefficient in the objective function and each
z = LpVariable("z", 0, None, LpContinuous, 9*obj + b + c) # of the constraints


prob.solve()
```

The main tricky part to columnwise modelling is entering the last parameter of the variable definitions correctly. Note that for constraint a, it has a coefficient of "+1" in the x and y variable definitions and is earlier specified to be less than or equal to 5. Therefore, all the information is specified to create "x+y <= 5" as was entered in full in row-wise modelling. Also note that a, b & c are all added to prob before the variables are created, and no more statements relating to prob need to made before the LP is solved. This is because obj, a, b & c are already added to prob.

Using this column-wise modelling, the Sponge Roll Problem can be solved more efficiently again. It is still in two files: A main file and a function file.

Main File

```
"""
The Sponge Roll Problem with Columnwise Column Generation for the PuLP Modeller



Authors: Antony Phillips,  Dr Stuart Mitchell  2008
"""


# Import Column Generation functions
from CGcolumnwise import *
```

After the standard introduction, a function called createMaster() is called. This will set up all aspects of the problem except for the pattern variables. i.e. it creates obj and assigns it to prob, it creates the constraints and adds them to prob (as above, when the constraints are created the only required specification is the R.H.S), it creates the surplus variables since they will not change. (It may be worthwhile scrolling down to see the createMaster function in the functions file)

```
# The Master Problem is created
prob, obj, constraints = createMaster()
```

A set of initial patterns which will make the problem solvable (not optimal) are required.

```
# A list of starting patterns is created
newPatterns = [[1,0,0],[0,1,0],[0,0,1]]
```

The newPatterns list will continue to be altered as more patterns are required to be added. Eventually the newPatterns list will be empty which will end the *while* loop

```
# New patterns will be added until newPatterns is an empty list
while newPatterns:
```

The addPatterns function will create the patterns in newPatterns as LpVariables.

```
    # The new patterns are added to the problem
    addPatterns(obj,constraints,newPatterns)
```

The masterSolve function is passed `prob` which is ready to solve, and the duals are passed out.

```
    # The master problem is solved, and the dual variables are returned
    duals = masterSolve(prob)
```

The subSolve function is still formulated as before (row-wise), receiving only the input of `duals` and returning the newPatterns list.

```
    # The sub problem is solved and a new pattern will be returned if there is one
    # which can reduce the master objective function
    newPatterns = subSolve(duals)
```

Once the newPatterns list is returned empty, the loop will end and one final solve of the problem is required which has non-relaxed integer constraints. The masterSolve function is designed to take the relax input and relax the variables, whilst still being able to use the same prob variable.

```
# The master problem is solved with Integer Constraints not relaxed
solution, varsdict = masterSolve(prob,relax = False)
```

The solution is printed.

```
# Display Solution
for i,j in varsdict.items():
    print i, "=", j



print "objective = ", solution
```

The main file is avialable here.

Function File CGcolumnwise.py

The start to the function file is the same as before except there is a class variable called numPatterns which is incremented each time the init function runs.

```
"""
Columnwise Column Generation Functions



Authors: Antony Phillips,  Dr Stuart Mitchell  2008
"""


# Import PuLP modeler functions
from pulp import *


class Pattern:
    """
    Information on a specific pattern in the SpongeRoll Problem
    """
    cost = 1
    trimValue = 0.04
    totalRollLength = 20
    lenOpts = ["5", "7", "9"]
    numPatterns = 0

    def __init__(self, name, lengths = None):
        self.name = name
        self.lengthsdict = dict(zip(self.lenOpts,lengths))
        Pattern.numPatterns += 1


    def __str__(self):
        return self.name


    def trim(self):
        return Pattern.totalRollLength - sum([int(i)*int(self.lengthsdict[i]) for i in self.lengthsdict])
```

createMaster

The createMaster function sets up the constraints and surplusVars so the rollData dictionary is required. The prob object is initialised.

```
def createMaster():

    rollData = {#Length Demand SalePrice
              "5":   [150,   0.25],
              "7":   [200,   0.33],
              "9":   [300,   0.40]}


    (rollDemand,surplusPrice) = splitDict(rollData)


    # The variable prob is created
    prob = LpProblem("MasterSpongeRollProblem",LpMinimize)
```

```
    # The variable obj is created and set as the LP's objective function
    obj = LpConstraintVar("Obj")
    prob.setObjective(obj)
```

Each constraint is logically named, given a sign and a R.H.S value. Each constraint is then added to prob and the constraints remain saved in a dictionary for use later when defining variables.

```
    # The constraints are initialised and added to prob
    constraints = {}
    for l in Pattern.lenOpts:
        constraints[l]= LpConstraintVar("Min" + str(l), LpConstraintGE, rollDemand[l])
        prob += constraints[l]
```

The surplus variables are created. The last parameter means that: the negative of the surplus price * surplus variable for each of the lenOpts, occurs in the objective function. It also means that the negative of the surplus variable for each of the length options is in the constraint for that length option. This column-wise variable definition is important to understand. The coefficient in front of the `obj` or the constraint is multiplied by the surplus variable and occurs in that respective constraint or the objective function. Note that this loop makes all three surplusVars in the objective function but only one in each constraint.

```
    # The surplus variables are created
    surplusVars = []
    for i in Pattern.lenOpts:
        surplusVars += [LpVariable("Surplus "+ i,0,None,LpContinuous, -surplusPrice[i] * obj - constraints[i])]

    return prob,obj,constraints
```

addPatterns

The addPatterns function is the first call inside the *while* loop. It's task is to create the patterns as LpVariable instances.

A *for* loop is used so that `i` becomes each of the pattern lists in newPatterns. (Since newPatterns is a list of lists)

```
def addPatterns(obj,constraints,newPatterns):

    # A list called Patterns is created to contain all the Pattern class
    # objects created in this function call
    Patterns = []
    for i in newPatterns:
```

Each pattern is checked that it does not use more cms of roll than are available

```
        # The new patterns are checked to see that their length does not exceed
        # the total roll length
        lsum = 0
        for j,k in zip(i,Pattern.lenOpts):
            lsum += j * int(k)
        if lsum > Pattern.totalRollLength:
            raise "Length Options too large for Roll"
```

Each pattern that is about to be added, is printed along with it's name. `Pattern.numPatterns` is a class variable that will increment each time the Pattern.__init__ function is run (i.e. when a new pattern is created as an instance of the Pattern class). Each of the patterns in newPatterns is created as a Pattern instance and added to the `Patterns` list.

```
        # The number of rolls of each length in each new pattern is printed
        print "P"+str(Pattern.numPatterns),"=",i


        # The patterns are instantiated as Pattern objects
        Patterns += [Pattern("P" + str(Pattern.numPatterns),i)]
```

The pattern variables are created. The lpSum term here just saves us from writing 'i.lengthsdict["5"]*constraints["5"] + i.lengthsdict["7"]*constraints["7"] + i.lengthsdict["9"]*constraints["9"]'. The last parameter of the LpVariable init function works the same as before, it just has more terms in this case. This is the end of the function - there is no output since the creation of LpVariable instances, referenced to `obj` and the `constraints`, adds the variables.

```
    # The pattern variables are created
    pattVars = []
    for i in Patterns:
        pattVars += [LpVariable("Pattern "+i.name,0,None,LpContinuous, (i.cost - Pattern.trimValue*i.trim())  obj\
         + lpSum([constraints[l]*i.lengthsdict[l] for l in Pattern.lenOpts]))]
```

masterSolve

The masterSolve works the same as in Column Generation, except it is already passed prob and so does not have to do so many steps.

If the relax parameter is passed as False, the variables will be set to Integer. This will only occur on the last run.

```
def masterSolve(prob,relax=True):

    # Unrelaxes the Integer Constraint
    if not relax:
        for v in prob.variables():
            v.cat = LpInteger
```

```
    # The problem is solved using CPLEX, with no message output and rounded
    prob.solve(CPLEX(msg=0))
    prob.roundSolution()
```

If the problem is relaxed then the call is from within the *while* loop and a duals dictionary is returned.

```
    if relax:
        # A dictionary of dual variable values is returned
        duals = {}
        for i,name in zip(Pattern.lenOpts,["Min5","Min7","Min9"]):
            duals[i] = prob.constraints[name].pi
        return duals
```

If the problem is not relaxed then the variable names, and their values (in varsdict), and the objective function value are returned.

```
    else:
        # A dictionary of variable values and the objective value are returned
        varsdict = {}
        for v in prob.variables():
            varsdict[v.name] = v.varValue

        return value(prob.objective), varsdict
```

subSolve

The subSolve function searches for more patterns that would reduce the objective function of the masterSolve further.

Most of this function is the same as it was in Column Generation.

```
def subSolve(duals):

    # The variable prob is created
    prob = LpProblem("SubProb",LpMinimize)


    # The problem variables are created
    vars = LpVariable.dicts("Roll Length", Pattern.lenOpts, 0, None, LpInteger)


    trim = LpVariable("Trim", 0 ,None,LpInteger)


    # The objective function is entered: the reduced cost of a new pattern
    prob += (Pattern.cost - Pattern.trimValue*trim) - lpSum([vars[i]*duals[i] for i in Pattern.lenOpts]), "Objective"


    # The conservation of length constraint is entered
    prob += lpSum([vars[i]*int(i) for i in Pattern.lenOpts]) + trim == Pattern.totalRollLength, "lengthEquate"


    # The problem is solved using CPLEX
    prob.solve(CPLEX(msg=0))


    # The variable values are rounded
    prob.roundSolution()
```

This is again similar to before, except the output is a list containing another list. The *if* statement is set to `< -10**-5` since otherwise some very small negative values which are meant to be zero (except became slightly negative due to floating point error) will cause a new pattern to be found.

```
    newPatterns = []
    # Check if there are more patterns which would reduce the master LP objective function further
    if value(prob.objective) < -10**-5:
        varsdict = {}
        for v in prob.variables():
            varsdict[v.name] = v.varValue
        # Adds the new pattern to the newPatterns list
        newPatterns += [[int(varsdict["Roll_Length_5"]),int(varsdict["Roll_Length_7"]),int(varsdict["Roll_Length_9"])]]

    return newPatterns
```

The full function file is available here.

 Previous: (5g) Column Generation

Start Optimisation with PuLP

Optimisation with PuLP Wiki

Welcome!

You have just entered the Optimisation with PuLP Wiki. Using this wiki you can:

1. Learn how to use Python and PuLP to build mathematical models for optimisation
2. Look at existing case studies that use PuLP for optimisation
3. Learn the syntax of the Python language

We have assumed that students have completed either ENGSCI255 or STATS255 or their equivalent before using this wiki.

This wiki is used for learning/teaching in two courses in the Department of Engineering Science:

1. ENGSCI761 Computational Optimisation in Operations Research
2. OPSRES392 Optimisation in Management Science

How to Use this Wiki

This wiki contains case studies that begin with a description of an optimisation problem, followed by a "walk-through" formulation of the mathematical model in PuLP and finally the results from solving the model as well as some analysis and discussion. For each of these case studies you will have to complete a management summary of the problem, solution and some analysis.

Within each wiki page there will be several links to concepts as they arise. If you follow these links you will come to a page with a detailed description of the concept, perhaps some dicsussion and (if appropriate) some notes on the necessary Python syntax.

As well at the "walk-through" cases studies there are several "do-it-yourself" (DIY) case studies. These case studies each have a problem description, with some discussion about its formulation, and some questions to be answered. For each of these case studies you will have to formulate a mathematical model in PuLP, solve the model and complete the management summary (be sure to answer the case study questions).

Adding Comments

Unlike traditional wikis you will not be able to edit these wiki pages directly. However, we encourage you to add your own comments at the bottom of each wiki page. We will continually monitor each page's comments for improvements to our wiki. We appreciate your help in improving this learning resource.

Start Optimisation with PuLP

Optimisation with PuLP

Getting Started

Now you know how to use this wiki, you can begin learning Python and using PuLP by looking at the content below. We recommend that you read The Optimisation Process, Optimisation Concepts, the Introduction to Python and the Introduction to PuLP before beginning the case-studies. The full PuLP function documentation is available here, and the useful functions will be explained in the case studies. The case studies are in order, so the later case studies will assume you have (at least) read the earlier case studies. However, we will provide links to any relevant information you will need.

1. **The Optimisation Process**

2. **Optimisation Concepts**

3. **Introduction to Python**
   a. Installing Python at Home
   b. Basic Python Coding

4. **Introduction to PuLP**
   a. Installing PuLP at Home

5. **"Walk Through" Case Studies**
   a. A Blending Problem
   b. A Transportation Problem
   c. A Transshipment Problem
   d. A Facility Location Problem
   e. A Cutting Stock Problem
   f. Sudoku As An LP
   g. Column Generation
   h. Column Generation 2

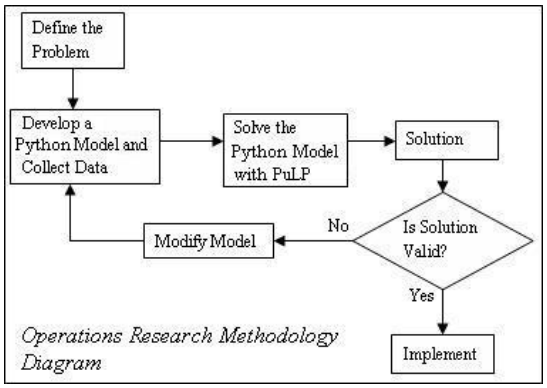
Next: (2) Optimisation Concepts

(1) The Optimisation Process

Solving an optimisation problem is not a linear process, but the process can be broken down into five general steps:

1. Getting the problem description
2. Formulating the mathematical programme
3. Solving the mathematical programme
4. Performing some post-optimal analysis
5. Presenting the solution and analysis

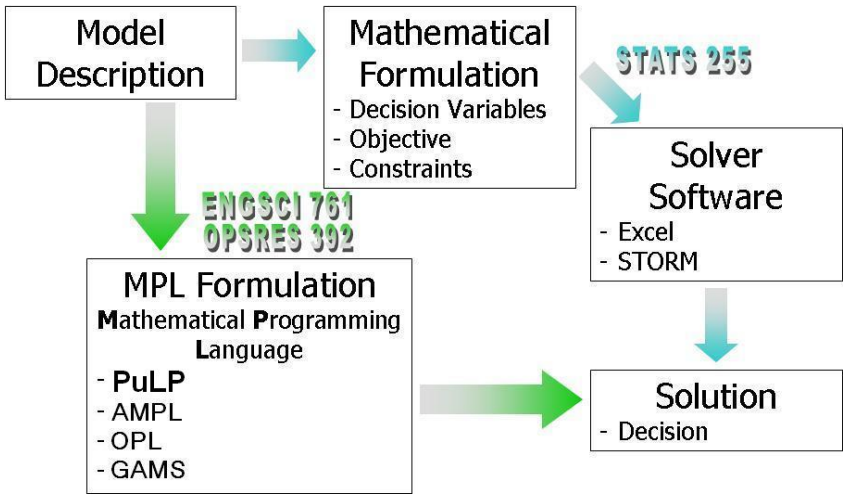
However, there are often "feedback loops" within this process. For example, after formulating and solving an optimisation problem, you will often want to consider the validity of your solution (often consulting with the person who provided the problem description). If your solution is invalid you may need to alter or update your formulation to incorporate your new understanding of the actual problem. This process is shown in the *Operations Research Methodology Diagram*. Note that we have altered the original diagram (from STATS 255) to reflect the use of Python and PuLP.



In ENGSCI 255 and STATS 255 you are taught mostly about the *Modeling Process*. The modeling process starts with a well-defined model description, then uses mathematics to formulate a mathematical programme. Next, the modeler enters the mathematical programme into some solver software, e.g., Excel or Storm, and solves the model. Finally, the solution is translated into a decision in terms of the original model description.

Using Python gives you a "shortcut" through the modeling process. By formulating the mathematical programme in Python you have already put it into a form that can be used easily by PuLP the modeller to call many solvers, e.g., CPLEX, GLPK, so you don't need to enter the mathematical programme into the solver software. However, you usually don't put any "hard" numbers into your formulation, instead you "populate" your model using data files, so there is some work involved in creating the appropriate data file. The advantage of using data files is that the same model may used many times with different data sets.

The Modeling Process



The modeling process is a "neat and tidy" simplification of the optimisation process. Let's consider the five steps of the optimisation process in more detail:

1. **Getting the Problem Description** The aim of this step is to come up with a formal, rigourous model description. Usually you start an optimisation project with an abstract description of a problem and some data. Often you need to spend some time talking

with the person providing the problem (usually known as the *client*). By talking with the client and considering the data available you can come up with the more rigourous model description you are used to. Sometimes not all the data will be relevant or you will need to ask the client if they can provide some other data. Sometimes the limitations of the available data may change your model description and subsequent formulation significantly.

2. **Formulating the mathematical programme** In this step we identify the key quantifiable decisions, restrictions and goals from the problem description, and capture their interdependencies in a mathematical model. We can break the formulation process into 4 key steps:
    1. *Identify the Decision Variables* paying particular attention to units (for example: we need to decide how many *hours per week* each process will run for).

    2. *Formulate the Objective Function* using the decision variables, we can construct a *minimise* or *maximise* objective function. The objective function typically reflects the total cost, or total profit, for a given value of the decision variables.

    3. *Formulate the Constraints*, either logical (for example, we cannot work for a negative number of hours), or explicit to the problem description. Again, the constraints are expressed in terms of the decision variables.

    4. *Identify the Data* needed for the objective function and constraints. To solve your mathematical programme you will need to have some "hard numbers" as variable bounds and/or variable coefficients in your objective function and/or constraints.

3. **Solving the mathematical programme** For relatively simple or well understood problems the mathematical model can often be solved to optimality (i.e., the best possible solution is identified). This is done using algorithms such as the Revised Simplex Method (see ENGSCI 391) or Interior Point Methods (see ENGSCI 768). However, many industrial problems would take too long to solve to optimality using these techniques, and so are solved using heuristic methods (such as Tabu search and Simulated Annealing - see ENGSCI 760) which do not guarantee optimality.

4. **Performing some post-optimal analysis** Often there is uncertainty in the problem description (either with the accuracy of the data provided, or with the value(s) of data in the future). In this situation the robustness of our solution can be examined by performing post-optimal analysis. This involves identifying how the optimal solution would change under various changes to the formulation (for example, what would be the effect of a given cost increasing, or a particular machine failing?). This sort of analysis can also be useful for making tactical or strategic decisions (for example, if we invested in opening another factory, what effect would this have on our revenue?).

    Another important consideration in this step (and the next) is the validation of the mathematical programme's solution. You should carefully consider what the solution's variable values mean in terms of the original problem description. Make sure they make sense to you and, more importantly, your client (which is why the next step, presenting the solution and analysis is important).

5. **Presenting the solution and analysis** A crucial step in the optimisation process is the presentation of the solution and any post-optimal analysis. The translation from a mathematical programme's solution back into a concise and comprehensible summary is as important as the translation from the problem description into the mathematical programme. Key observations and decisions generated via optimisation must be presented in an easily understandable way for the client or project stakeholders.

    Your presentation is a crucial first step in the implementation of the decisions generated by your mathematical programme. If the decisions and their consequences (often determined by the mathematical programme constraints) are not presented clearly and intelligently your optimal decision will never be used.

    This step is also your chance to suggest other work in the future. This could include:
    - Periodic monitoring of the validity of your mathematical programme;
    - Further analysis of your solution, looking for other benefits for your client;
    - Identification of future optimisation opportunities.


Next: (2) Optimisation Concepts

Previous: (1) The Optimisation Process

Next: (3a) Installing Python at Home

(2) Optimisation Concepts

Linear Programing

The simplest type of mathematical programme is a *linear programme*. For your mathematical programme to be a linear programme you need the following conditions to be true:

1. The decision variables must be real variables;
2. The objective constraint must be a linear expression;
3. The constraints must be linear expressions.

Linear expressions are any expression of the form

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n \{\leq = \geq\} b$$

where $a_1, a_2, , a_n$ and b are known quantities and $x_1, x_2, , x_n$ are variables. The process of solving a linear programme is called *linear programming*. Linear programming is done via the Revised Simplex Method (also known as the Primal Simplex Method), the Dual Simplex Method or an Interior Point Method. CPLEX allows you to specify which method you use, but we won't go into further detail here.

Integer Programing

Integer programmes are almost identical to linear programmes with one very important exception. Some of the decision variables in integer programmes may need to have only integer values. The variables are known as integer variables. Since most integer programmes contain a mix of real variables and integer variables they are often known as *mixed integer programmes*. While the change from linear programming is a minor one, the effect on the solution process is enormous. Integer programmes can be very difficult problems to solve and there is a lot of current research finding "good" ways to solve integer programmes. Integer programming (the process of solving a (mixed) integer programme) was originally done using the branch-and-bound process. The *branch* part of the process eliminated non-integer values for integer variables in the following way:

- Initially, all variables are left as real variables. The problem is solved using linear programming;
- If one of the integer variables in the linear programming solution has a fractional value, e.g., $x_i = 4.5$, then the linear programme is split in two and the fractional region eliminated. This is done by *branching* on the variable value, e.g., adding the constraint $x_i \leq 4$ to form one linear programme and $x_i \geq 5$ to form the other.
- By finding the optimal solution in each of these new linear programmes and comparing we can find the optimal solution for the original problem.
- If either of the new linear programmes has a fractional value for an integer variable then a new branch is needed.

This branching process results in the formation of a *branch-and-bound tree* (we will discuss the bounding next). Each node in this tree represents a linear programme consisting of the original linear programme and the extra branches added. Eventually all the *leaf nodes* in the tree will

contain solutions where all the integer variables have integer values and no further branching is needed. All these values can be compared and the best one is the solution to the original integer programme.
Note For MIPs of any reasonable size this tree could be huge, in fact it *grows exponentially* as the number of integer variables increases. The bounding process allows sections of the branch-and-bound tree to be removed from consideration before all the leaf nodes have integer solutions. It relies on the following optimization principle:

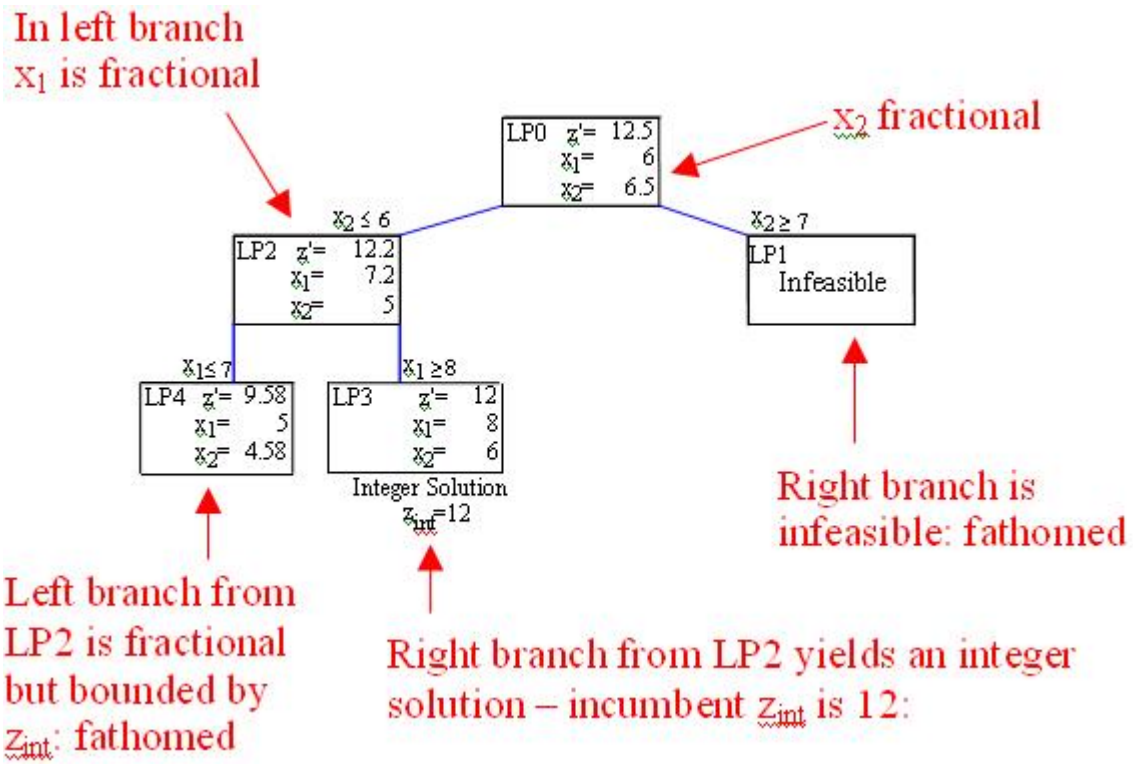
Adding constraints to a mathematical programming will result in a deterioration of the optimal objective value.

This means that adding the branching constraints to the linear programmes at the branch-and-bound tree nodes will mean the resulting nodes will have an optimal objective function value that is equal to or worse than the optimal objective function value of the original linear programme. Thus the objective function values get worse the deeper into the tree you look. Since we are finding the integer solution in the branch-and-bound tree with the best objective value, we can use any integer solutions to bound the tree. The current best *integer* solution is called the *incumbent*. After solving a linear programme at a leaf node of the branch-and-bound tree one of the following conditions holds:

- The linear programme is infeasible (no more branching is possible);
- The linear programme solution is an integer solution with a better objective value than the incumbent. The incumbent is replaced with this new solution;
- The linear programme solution has a worse objective than the incumbent. Any nodes created from this node will also have a worse objective than the incumbent. This node is *bounded* by the incumbent objective;
- The linear programme solution is fractional and has a better objective value than the incumbent. Further branching from this node is necessary to ensure an optimal solution is found.

Only the last condition requires more branching, all the other conditions result in the node becoming *fathomed* and no more branching is required from that node.

Example



The LP Relaxation

The Linear Programming (LP) relaxation is the same as the integer programme, except we "relax" the integer variables to allow them to take fractional values. The integer programme's feasible region lies within the feasible region of the LP relaxation (at points where the integer variables have integer values). Therefore the integer restrictions cause the optimal objective function value to be worse in the integer programme as compared to the LP relaxation. However, if a solution x of the LP relaxation has integer values for the integer variables, then x also solves the integer programme. In some cases, if the solution values for the integer variables are large, then rounding the LP relaxation solution may give a good solution to the integer programme. However, you need to make sure that the rounded solution is not infeasible! For some classes of problem the LP relaxation gives naturally integer solutions:

- An $m \times m$ matrix M is *unimodular* if and only if its determinant $|M|$ is equal to 1 or –1;
- An $m \times n$ matrix M is *totally unimodular* if and only if every $m \times m$ non-singular submatrix of M is unimodular;
- If the constraint matrix A and the right-hand side vector b of a mixed-integer programme are totally unimodular and integer respectively, then the mixed-integer programme is naturally integer and the LP relaxation solution is the optimal solution
    - The transportation problem is an example of one such problem;
    - Most network flow problems are also naturally integer;
    - Some scheduling problems are naturally integer.

When using PuLP, naturally integer variables are defined when the variables are created with the `LpVariable` function. A parameter for this function is either LpContinuous or *LpInteger*.

Master-Slave Constraints

Using zero/one variables, we can control the range of values that other variables take. Suppose that x_{AB} (the amolunt shipped from A to B) is either 0 (we don't ship from A to B) or between 20 and 100 (we ship from A to B with limits specified by the transportation company). We introduce a new 0/1-variable z_{AB} that is 1 if there is a shipment from A to B and 0 otherwise. Then we can use a *master-slave constraints* to let z_{AB} control x_{AB}

$x_{AB} amp;\geq 20 z_{AB}$

$x_{AB} amp;\leq 100 z_{AB}$

Sensitivity Analysis

In STATS/ENGSCI 255 you will have seen the following *sensitivity analysis* output from Excel and/or Storm.

Excel Output

After solving a problem, Excel can generate a sensitivity table as one of its output report sheets.

The sensitivity report for Case Chemicals is shown below. Note that to get all the items shown below, you must tell Excel to 'Assume Linear Model' under Solver Options. Unlike Storm, which gives ranges for values (eg 250 to 1000 for CS-01), Excel gives this information in terms of allowable increases (e.g. 700 on CS-01, i.e. up to 300+700=1000) and decreases (50 on CS-01, i.e. down to 300-50=250).

Changing Cells

Cell	Name	Final Value	Reduced Cost	Objective Coefficient	Allowable Increase	Allowable Decrease
B2	CS-01	70	0	300	700	50
C2	CS-02	90	0	500	100	350

Constraints

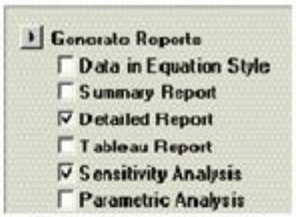
Cell	Name	Final Value	Shadow Price	Constraint R.H. Side	Allowable Increase	Allowable Decrease
D6	BLENDHRS VALUE	230	33.33333333	230	270	90
D7	PURIHRS VALUE	250	233.3333333	250	45	133
D8	CS02LIM VALUE	90	0	120	1E+30	30

Case Chemicals - Excel Sensitivity Analysis Table

SENSITIVITY ANALYSIS COMPUTER OUTPUT

Storm Output

Once we have found the optimal solution to the Case Chemical problem in Storm, we can choose the Detailed Report and Sensitivity Analysis reports. This gives the output shown below. (Irrelevant sections of the detailed report are not shown.) We explain how to use these tables in the next section.



Storm Report Options

STORM OUTPUT
CASE CHEMICALS
OPTIMAL SOLUTION - DETAILED REPORT

	Constraint	Type	RHS	Slack	Shadow price
1	BLENDHRS	<=	230.0000	0.0000	33.33
2	PURIHRS	<=	250.0000	0.0000	233.33
3	CS2LIM	<=	120.0000	30.0000	0.0000

Objective Function Value = 66000

Case Chemicals - Storm Detailed Report (shadow price information)

STORM OUTPUT
CASE CHEMICALS

SENSITIVITY ANALYSIS OF COST COEFFICIENTS

	Variable	Current Coeff.	Allowable Minimum	Allowable Maximum
1	CS01	300.00	250.00	1000.00
2	CS02	500.00	150.00	600.00

SENSITIVITY ANALYSIS OF RIGHT-HAND SIDE VALUES

	Constraint	Type	Current Value	Allowable Minimum	Allowable Maximum
1	BLENDHRS	<=	230.0000	140.0000	500.0000
2	PURIHRS	<=	250.0000	115.0000	295.0000
3	CS2LIM	<=	120.0000	90.0000	Infinity

Case Chemicals - Storm Sensitivity Report

What do these numbers mean? First, let's look at the Excel sensitivity analysis. For the variables, the *Allowable Increase* and *Allowable Decrease* show how much the objective coefficient of that variable can change without changing the optimal solution (although the objective function will change!). For the constraints the Allowable Increase and Allowable Decrease show how much the right-hand side of the constraint (i.e., the part of the constraint that does **not** involve variables) can increase or decrease without changing which variables are non-zero (although the variable values will change!). The *shadow price* gives the amount the objective function changes for each **unit** change in the right-hand side. If the constraint gets *tighter* then the objective function will *deteriorate*. The following table gives the various combinations for constraints and objective functions:

Constraint Relation	Change in Constraint RHS	Objective Type	Change in Objective Function
$\leq$	$-$ (harder)	min	+ (worse)
$\leq$	+ (easier)	min	$-$ (better)
$\leq$	$-$ (harder)	max	$-$ (worse)
$\leq$	+ (easier)	max	+ (better)
$\geq$	$-$ (easier)	min	$-$ (better)
$\geq$	+ (harder)	min	+ (worse)
$\geq$	$-$ (easier)	max	+ (better)
$\geq$	+ (harder)	max	$-$ (worse)

The Storm sensitivity analysis provides the same information, but gives the Allowable Minimum and the Allowable Maximum instead. **Note** that the Allowable Minimum = Current Value -

Allowable Decrease and the Allowable Maximum = Current Value + Allowable Increase.

Parametric Analysis

In STATS/ENGSCI 255 you will have seen the following *parametric analysis*:

```
Case Chemicals
PARAMETRIC ANALYSIS OF RIGHT-HAND SIDE VALUE - BLENDHRS
COEF = 230.000     LWR LIMIT = -1.000E+37  UPR LIMIT = 1.000E+37
     ------- Range -------        Shadow    ---- Variable ----
          From          To         Price       Leave      Enter
RHS      230.000      500.000       33.333       CS-02  SLACK   1
Obj    66000.000    75000.000

RHS      500.000    1.000E+37        0.000  ----  No change  ----
Obj    75000.000    75000.000

RHS      230.000      140.000       33.333   SLACK   3  SLACK   2
Obj    66000.000    63000.000

RHS      140.000      120.000      150.000       CS-01  SLACK   3
Obj    63000.000    60000.000

RHS      120.000        0.000      500.000       CS-02
Obj    60000.000        0.000

RHS        0.000    -Infinity  ---- Infeasible in this range ----
```

This analysis shows how the objective and shadow price change as a right-hand side value increases (from 230 to 500, then 500 upwards. It also shows how the objective and shadow price change as the right-hand side value decreases (from 230 to 140, 140 to 120, 120 to 0, then 0 downwards). Similar tables are also available in STORM for changes in the cost coefficients. These tables assume that only one quantity (right-hand side, cost) is changing.

 Previous: (1) The Optimisation Process

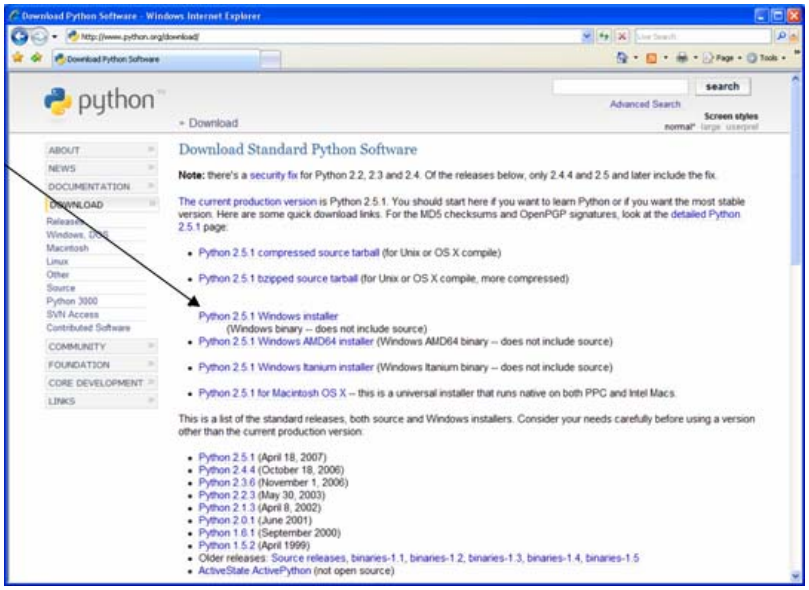
 Next: (3a) Installing Python at Home

Previous: (2) Optimisation Concepts

Next: (3b) Basic Python Coding

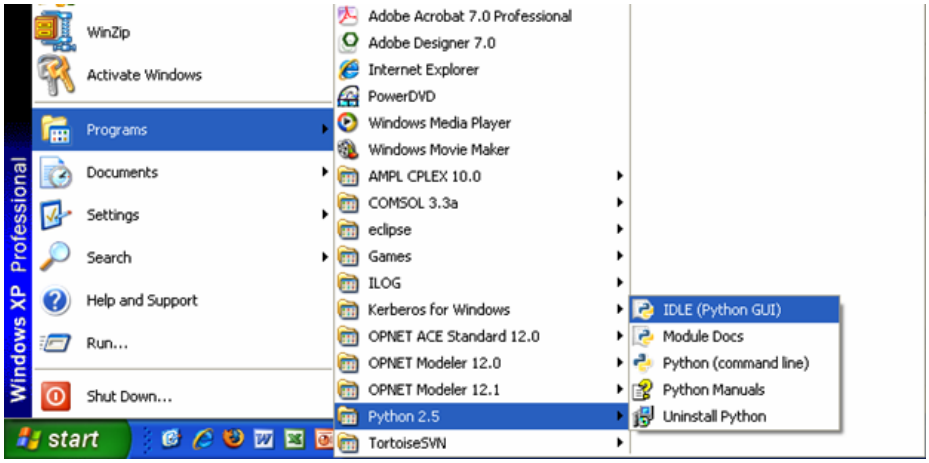
(3a) Installing Python at Home

You can easily set up Python on your home computer. First navigate to the Python Downloads Website and download the latest version (> 2.5.1) in a format suitable to you, depending on whether you use Windows or Unix. For Windows users, select the format indicated in the diagram. After selecting this version, you may need to select your processor type. For most people this will be x86. The .msi file will be around 11MB.



Once the file is downloaded it can be installed on your computer and shortcuts will be made in Start > Programs > Python 2.5. The useful installed components include:

- IDLE - The most simple Python GUI (Graphic User Interface). Python files can be written in this GUI and as you type, the text is higlighted automatically in different colours for improved readability.
- Python (Command Line) - The command line for Python. Can be useful to test small segments of code.
- Python Manuals - Extensive Python Documentation including a Beginners Guide to Programming Tutorial. There are many good guides to programming in Python which will be linked to in the next section on Basic Python Coding.

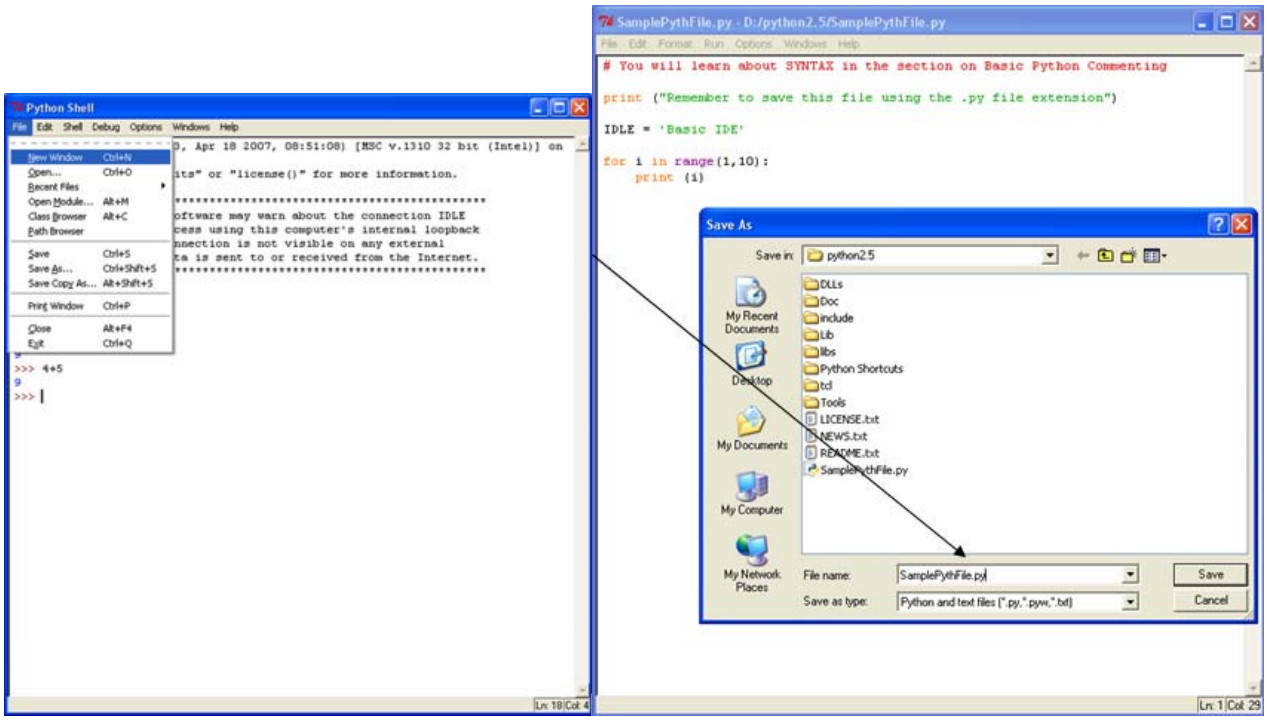


For your coding in this course, you will not actually *need* an advanced Python Graphic User Interface, however the recommended development environment for your Python code is Eclipse (www.eclipse.org). This is primarily used for Java programming but with the PyDev Plug-in, it can be used very effectively for Python.

It is your choice in this course whether to use IDLE (comes with Python) or Eclipse with PyDev for creating your .py (Python Extension) files. As a Development Environement, IDLE is similar in layout to MATLAB and Eclipse is similar in layout to Visual Studio.

Using IDLE

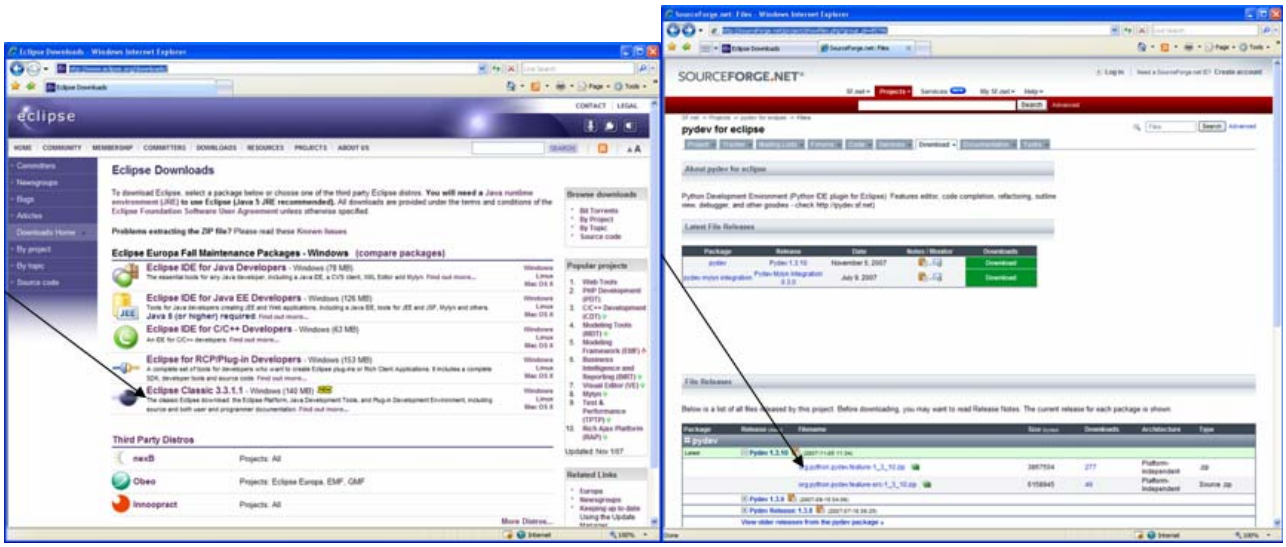
The Python Shell can be opened through the Start Menu. In the Python Shell, there is a command line which can be used for testing code. "File > New Window" will open a new empty Python code file where you can write your code and save it as a .py extension to be recognised. Writing .py is very important (as shown below).



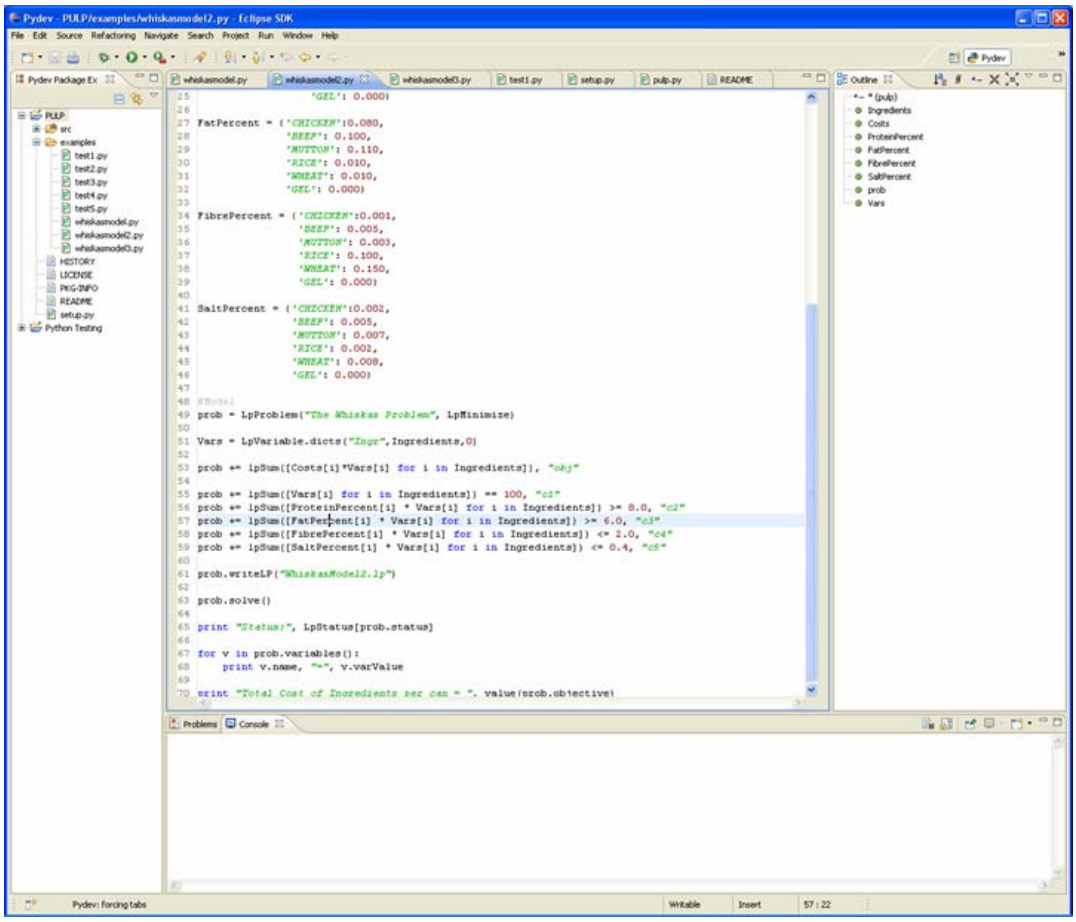
Using Eclipse with PyDev

This must be downloaded from the Eclipse Website. Download the latest version of Eclipse Classic (we will use 3.3.1.1) from any of the mirror sites listed after following the link. Once downloaded, it can be easily extracted and installed onto your computer.

PyDev can be downloaded from here. Information about the PyDev plug-in can be found on The PyDev Site.



After installing the PyDev plug-in succesfully, the Eclipse IDE should look like:



A new PyDev project can be created with "File>New>Project" and new files can similarly be created with "File>New>File". It is important to save the files with the .py extension.

Finally, before the code will run, Eclipse needs to be given the link to the Python Interpreter. In the taskbar, follow "Window>Preferences>PyDev>Interpreter-Python" and locate and add Python.exe into the Interpreter box.

Previous: (2) Optimisation Concepts

Next: (3b) Basic Python Coding

 Previous: (3b) Basic Python Coding

 Next: (5a) A Blending Problem

(4) Installing PuLP at Home

PuLP is a free open source software written in Python. It is used to describe optimisation problems as mathematical models. PuLP can then call any of numerous external LP solvers (GLPK, CPLEX, XPRESS etc) to solve this model and finally display the solution.

In your course you will solve many mathematical programmes using PuLP. These mathematical programmes will tell you such varied things as:

- The mix of ingredients to use to provide a nutritious can of cat food whilst minimising costs
- The price and production quantity for a surfboard shop to maximise profit
- The number of crates of beer to ship from breweries to pubs to minimise the shipping cost

The old version of PuLP is available on the internet, but an upgraded version of PuLP can be freely obtained from here, which is the version you will be using in this course.
To install:

1. Extract the PuLP-1.10.zip folder to a suitable location (such as the desktop - the folder will be no longer required after installation)
2. Open a command prompt by clicking "Run" in the Start Menu, and type `cmd` in the window and push enter.
3. Navigate to the extracted PuLP-1.10 folder with the setup file in it. [Do this by typing `cd foldername` at the prompt, where `cd` stands for current directory and the `foldername` is the name of the folder to open in the path already listed to the left of the prompt. To return back to a root drive, type 'cd C:\']
4. Type `setup.py install` at the command prompt. This will install all the PuLP functions into Python's callable modules.

Since PuLP is still under development, the following procedures also need to be carried out:

1. Open the `src` folder in the PuLP-1.10 folder and copy the `pulp.cfg` config file. (Two files may appear to be called `pulp`, but one is a python file and one is a config file, as shown by the file extensions)
2. Find the folder which the pulp package files are located inside, within Pythons files : C:/ ... /Python25/Lib/site-packages/pulp
3. Paste the `pulp.cfg` inside this directory, with the other pulp files

The PuLP function library is now able to be imported from any python command line! Go to IDLE or PyDev and type `from pulp import *` to load in the functions. (You need to re-import the functions each time after you close the GUI)

PuLP is written in a programming language called Python, and to use PuLP you must write Python code to describe your optimisation problem.

 Previous: (3b) Basic Python Coding


Next: (5a) A Blending Problem

Previous: (3a) Installing Python at Home

Next: (4a) Installing PuLP at Home

(3b) Basic Python Coding

In this course you will learn basic programming in Python, but there is also excellent Python Language reference material available on the internet freely. You can download the book "Dive Into Python" or there are a host of Beginners Guides to Python on the Python Website. Follow the links to either "BeginnersGuide/NonProgrammers" or "BeginnersGuide/Programmers" depending on your level of current programming knowledge. The code sections below assume a knowledge of fundamental programming principles and mainly focus on Syntax specific to programming in Python.
Note: >>> represents a Python command line prompt.

Commenting in Python

Commenting at the top of a file is done using """ to start and to end the comment section. Commenting done throughout the code is done using the hash # symbol at the start of the line.

Lists

A list is simply a sequence of variables grouped together. The *range* function is often used to create lists of integers, with the general format of range(start,stop,step). The default for start is 0 and the default for step is 1.

```
>>> range(3,8)
[3,4,5,6,7]
```

This is a list/sequence. As well as integers, lists can also have strings in them or a mix of integers, floats and strings. They can be created by a loop (as shown in the next section) or by explicit creation (below). Note that the `print` statement will display a string/variable/list/... to the user

```
>>> a = [5,8,"pt"]
>>> print a
[5,8,'pt']
>>> print a[0]
5
```

Loops in Python

for Loop

The general format is:

```
for variable in sequence:
     #some commands
#other commands after for loop
```

Note that the formatting (indents and new lines) governs the end of the *for* loop, whereas the start of the loop is the colon `:`.
Observe the loop below, which is similar to loops you will be using in the course. The variable `i` moves through the string list becoming each string in turn. The top section is the code in a .py file and the bottom section shows the output

```
#The following code demonstrates a list with strings
ingredientslist = ["Rice","Water","Jelly"]



for i in ingredientslist:
    print i


print "No longer in the loop"
```

```
>>>
Rice
Water
Jelly
No longer in the loop
```

while Loop

These are similar to *for* loops except they continue to loop until the specified condition is no longer true. A while loop is not told to work through any specific sequence.

```
i = 3
while i <= 15:
    # some commands
    i = i + 1 # a command that will eventually end the loop is naturally required
# other commands after while loop
```

For this specific simple *while* loop, it would have been better to do as a for loop but it demonstrates the syntax. *while* loops are useful if the number of iterations before the loop needs to end, is unknown.

The *if* statement

This is performed in much the same way as the loops above. The key identifier is the colon ： to start the statements and the end of indentation to end it.

```
if j in testlist:
    # some commands
elif j == 5:
    # some commands
else:
    # some commands
```

Here it is shown that "elif" (else if) and "else" can also be used after an *if* statement. "else" can in fact be used after both the loops in the same way.

Tuples

Tuples are basically the same as lists, but with the important difference that they cannot be modified once they have been created. They are assigned by:

```
>>> x = (4,1,8,"strng",[1,0],("j",4,"o"),14)
```

Tuples can have any type of number, strings, lists, other tuples, functions and objects, inside them. Also note that the first element in the tuple is numbered as element "zero". Accessing this data is done by:

```
>>> x[0]
4
>>> x[3]
"strng"
```

Dictionaries

A Dictionary is a list of reference keys each with associated data, whereby the order does not affect the operation of the dictionary at all. With dictionaries, the keys are not consecutive integers (unlike lists), and instead could be integers, floats or strings. This will become clear:

```
>>>x = {} # creates a new empty dictionary - note the curly brackets denoting the creation of a dictionary
>>>x[4] = "programming" # the string "programming" is added to the dictionary x, with "4" as it's reference
>>>x["games"] = 12
>>>print x["games"]
12
```

In a dictionary, the reference keys and the stored values can be any type of input. New dictionary elements are added as they are created (with a list, you cannot access or write to a place in the list that exceeds the initially defined list dimensions).

```
costs = {"CHICKEN": 1.3, "BEEF": 0.8, "MUTTON": 12}



print "Cost of Meats"
for i in costs:
    print i
    print costs[i]


costs["LAMB"] = 5


print "Updated Costs of Meats"
for i in costs:
    print i
    print costs[i]
```

```
>>>
Cost of Meats
CHICKEN
1.3
MUTTON
12
BEEF
0.8
Updated Costs of Meats
LAMB
5
CHICKEN
1.3
MUTTON
12
BEEF
0.8
```

In the above example, the dictionary is created using curly brackets and colons to represent the assignment of data to the dictionary keys. The variable i is assigned to each of the keys in turn (in the same way it would be for a list with "for i in range(1,10)"). Then the dictionary is called with this key, and it *returns the data stored under that key name*. These types of for loops using dictionaries will be highly relevant in using PuLP to model LPs in this course.

List/Tuple/Dictionary Syntax Note

Note that the creation of a: list is done with square brackets [], tuple is done with round brackets(), dictionary is done with parentheses{}.

After creation however, when accessing elements in the list/tuple/dictionary, the operation is always performed with square brackets (i.e a [3]). If a was a list or tuple, this would return the fourth element. If a was a dictionary it would return the data stored with a reference key of 3.

List Comprehensions

Python supports "List Comprehensions," which is a fast and concise way to create lists without using multiple lines. They are easily understandable when simple, and you will be using them in your code for this course.

```
a = [i for i in range(5)]
```

This statement above will create the list [0,1,2,3,4] and assign it to the variable "a".

```
odds = [i for i in range(25) if i%2==1]
```

This statement above uses the if statement and the modulus operator(%) so that only odd numbers are included in the list: [1,3,5,...,19,21,23]. (Note: The modulus operator calculates the remainder from an integer division.

```
fifths = [i for i in range(25) if i%5==0]
```

This will create a list with every fifth value in it [0,5,10,15,20]. Existing lists can also be used in the creation of new lists below:

```
a = [i for i in range(25) if (i in odds and i not in fifths)]
```

Note that this could also have been done in one step from scratch:

```
a = [i for i in range(25) if (i%2==1 and i%5==0)]
```

For a challenge you can try creating (a) a list of prime numbers up to 100, and (b) a list of all "perfect" numbers.

More List Comprehensions Examples Wikipedia: Perfect Numbers

Import Statement

At the top of all your Python coding that you intend to use PuLP to model, you will need the *import* statement. This statement makes the contents of another module (file of program code) available in the module you are currently writing i.e. functions and values defined in pulp.py that you will be required to call, become useable. In this course you will use:

```
from pulp import *
```

The asterisk represents that you are importing all names from the module of pulp. Calling a function defined in pulp.py now can be done as if they were defined in your own module.

Functions

Functions in Python are defined by: (`def` is short for define)

```
def name(inputparameter1, inputparameter2, . . .):
    #function body
```

For a real example, note that if inputs are assigned a value in the function definition, that is the default value, and will be used only if no other value is passed in. The order of the input parameters (in the definition) does not matter at all, as long as when the function is called, the parameters are entered in the corresponding correct order:

```
def listappender(list1=[0],endmessage="EOL",list2=[9,8,7]):
    list1.append(list2)
    list1.append(endmessage)
    return list1



print listappender([3,5,6],ListOver)
```

```
>>>
[3,5,6,[9,8,7],'ListOver"]
```

In the above example, the output from the function call is printed. The default value for "list1" is [0], but this was overwritten by the input of [3,5,6]. List2 of [9,8,7] was not overwritten and so the default value was used and appended, along with the overwritten "endmessage" string.

Classes

To demonstrate how classes work in Python, look at the following class structure.

The class name is Pattern, and it contains several class variables which are relevant to any instance of the Pattern class (i.e a Pattern). The first function is the `__init__` function which creates an instance of the Pattern class and assigns the attributes of name and lengthsdict using `self`..
The `__str__` function defines what to return if the class instance is printed.
The `trim` function acts like any normal function, except as with all class functions, `self` must be in the input brackets.

```
class Pattern:
    """
    Information on a specific pattern in the SpongeRoll Problem
    """
    cost = 1
    trimValue = 0.04
    totalRollLength = 20
    lenOpts = [5, 7, 9]

    def __init__(self,name,lengths = None):
        self.name = name
        self.lengthsdict = dict(zip(self.lenOpts,lengths))


    def __str__(self):
        return self.name


    def trim(self):
        return Pattern.totalRollLength - sum([int(i)*self.lengthsdict[i] for i in self.lengthsdict])
```

This class could be used as follows:

```
>>> Pattern.cost # The class variables can be accessed without making an instance of the class
1
>>> a = Pattern("PatternA",[1,0,1])
>>> a.cost # a is now an instance of the Pattern class and is associated with Pattern class variables
1
>>> print a # This calls the Pattern.__str__() function
"PatternA"
>>> a.trim() # This calls the Pattern.trim() function. Note that no input is required. The self in the function definition is an implied input
6
```

Previous: (3a) Installing Python at Home
Next: (4a) Installing PuLP at Home

Previous: (4a) Installing PuLP at Home

Next: (5b) A Transportation Problem

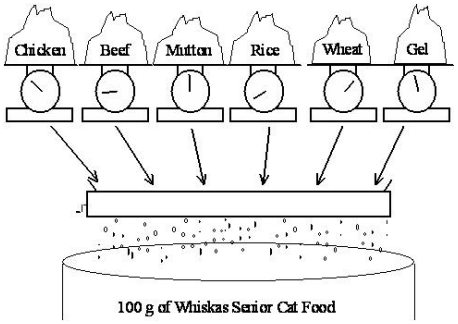
(5a) A Blending Problem

The Whiskas Problem

Problem Description

Whiskas cat food, shown above, is manufactured by Uncle Ben's. Uncle Ben's want to produce their cat food products as cheaply as possible while ensuring they meet the stated nutritional analysis requirements shown on the cans. Thus they want to vary the quantities of each ingredient used (the main ingredients being chicken, beef, mutton, rice, wheat and gel) while still meeting their nutritional standards.

INGREDIENTS: Selected meat derived from chicken, beef and mutton; rice; wheat bran; gel; all essential vitamins and minerals, including thiamine and taurine. No preservatives.	NUTRITIONAL ANALYSIS:	
	Minimum% Crude Protein	8.0
	Minimum% Crude Fat	6.0
	Maximum% Crude Fibre	2.0
	Max % Salt (Naturally Occurring)	0.4



The costs of the chicken, beef, and mutton are \$0.013, \$0.008 and \$0.010 respectively, while the costs of the rice, wheat and gel are \$0.002, \$0.005 and \$0.001 respectively. (All costs are per gram.) For this exercise we will ignore the vitamin and mineral ingredients. (Any costs for these are likely to be very small anyway.)

Each ingredient contributes to the total weight of protein, fat, fibre and salt in the final product. The contributions (in grams) per gram of ingredient are given in the table below.

	Protein	Fat	Fibre	Salt
Chicken	0.100	0.080	0.001	0.002
Beef	0.200	0.100	0.005	0.005
Mutton	0.150	0.110	0.003	0.007
Rice	0.000	0.010	0.100	0.002
Wheat bran	0.040	0.010	0.150	0.008
Gel	-	-	-	-

Simplified Formulation

First we will consider a simplified problem to build a simple Python model.

1. **Identify the Decision Variables**

   Assume Whiskas want to make their cat food out of just two ingredients: Chicken and Beef. We will first define our decision variables:

   x_1 = percentage of chicken meat in a can of cat food
   x_2 = percentage of beef used in a can of cat food

   The limitations on these variables (greater than zero) must be noted but for the Python implementation, they are not entered or listed separately or with the other constraints.

2. **Formulate the Objective Function**

   The objective function becomes:

$$min \$0.013x_1 + \$0.008x_2$$

3. **Formulate the Constraints**

   The constraints on the variables are that they must sum to 100 and that the nutritional requirements are met:

$$\begin{array}{rcl} 1.000x_1 + 1.000x_2 & = & 100.0 \\ 0.100x_1 + 0.200x_2 & \geq & 8.0 \\ 0.080x_1 + 0.100x_2 & \geq & 6.0 \\ 0.001x_1 + 0.005x_2 & \leq & 2.0 \\ 0.002x_1 + 0.005x_2 & \leq & 0.4 \end{array}$$

Solution to Simplified Problem

To obtain the solution to this Linear Program, we can write a short program in Python to call PuLP's modelling functions, which will then call a solver. This will explain step-by-step how to write this Python program. It is suggested that you repeat the exercise yourself.

The start of the your file should then be headed with a short commenting section outlining the purpose of the program. For example:

```
"""
The Simplified Whiskas Model Python Formulation for the PuLP Modeller



Authors: Antony Phillips, Dr Stuart Mitchell    2007
"""
```

Then you will import PuLP's functions for use in your code:

```
# Import PuLP modeler functions
from pulp import *
```

A variable called "prob" (although its name is not important) is created using the `LpProblem` function. It has two parameters, the first being the arbitrary name of this problem (as a string), and the second parameter being either `LpMinimize` or `LpMaximize` depending on the type of LP you are trying to solve:

```
# Create the prob variable to contain the problem data
prob = LpProblem("The Whiskas Problem",LpMinimize)
```

The problem variables "x1" and "x2" are created using the `LpVariable` function. It has four parameters, the first is the arbitrary name of what this variable represents, the second is the lower bound on this variable, the third is the upper bound, and the fourth is essentially the type of data (discrete or continuous). The options for the fourth parameter are `LpContinuous` or `LpInteger`, with the default as `LpContinuous`. If we were modelling the number of cans to produce, we would need to input `LpInteger` since it is discrete data. The bounds can be entered directly as a number, or `None` to represent no bound (i.e. positive or negative infinity), with `None` as the default. If the first few parameters are entered and the rest are ignored (as shown), they take their default values. However, if you wish to specify the third parameter, but you want the second to be the default value, you will need to specifically set the second parameter as it's default value. i.e you cannot leave a parameter entry blank. (e.g LpVariable("example",None,100))

```
# The 2 variables Beef and Chicken are created with a lower limit of zero
x1 = LpVariable("ChickenPercent",0)
x2 = LpVariable("BeefPercent",0)
```

The variable "prob" now begins collecting problem data with the "+=" operator. The objective function is logically entered first, with an important comma `,` at the end of the statement and a short string explaining what this objective function is:

```
# The objective function is added to prob first
prob += 0.013*x1 + 0.008*x2, "Total Cost of Ingredients per can"
```

The constraints are now entered (Note: any "non-negative" constraints were already included when defining the variables). This is done with the "+=" operator again, since we are adding more data to the "prob" variable. The constraint is logically entered after this, with a comma at the end of the constraint equation and a brief description of the cause of that constraint:

```
# The five constraints are entered
prob += x1 + x2 == 100, "PercentagesSum"
prob += 0.100*x1 + 0.200*x2 >= 8.0, "ProteinRequirement"
prob += 0.080*x1 + 0.100*x2 >= 6.0, "FatRequirement"
prob += 0.001*x1 + 0.005*x2 <= 2.0, "FibreRequirement"
prob += 0.002*x1 + 0.005*x2 <= 0.4, "SaltRequirement"
```

Now that all the problem data is entered, the `writeLP` function can be used to copy this information into a .lp file into the directory that your code is running from. Once your code runs successfully, you can open this .lp file with Notepad to see what the above steps were doing. You will notice that there is no assignment operator (such as an equals sign) on this line. This is because the function/method called `writeLP` is being performed to the variable/object "prob" (and the string "WhiskasModel.lp" is an additional parameter). The dot "." between the variable/object and the function/method is important and is seen frequently in Object Oriented software (such as this):

```
# The problem data is written to a .lp file
prob.writeLP("WhiskasModel.lp")
```

The LP is solved using the solver that PuLP chooses. The input brackets after `solve` are left empty in this case, however they can be used to specify which solver to use (e.g prob.solve(CPLEX())):

```
# The problem is solved using PuLP's choice of solver
prob.solve()
```

Now the results of the solver call can be displayed as output to us. Firstly, we request the status of the solution, which can be one of "Not Solved", "Infeasible", "Unbounded", Undefined" or "Optimal". The value of prob.status is returned as an integer, which must be converted to it's significant text meaning using the `LpStatus` dictionary. Since LpStatus is a dictionary, it's input must be in square brackets:

```
# The status of the solution is printed to the screen
print "Status:", LpStatus[prob.status]
```

The variables and their resolved optimum values can now be printed to the screen. The *for* loop makes "v" cycle through all the problem variable names (in this case just "ChickenPercent" and "BeefPercent"). Then it prints each variable name, followed by an equals sign, followed by its optimum value. `.name` and `.varValue` are properties of the object `v`.

```
# Each of the variables is printed with it's resolved optimum value
for v in prob.variables():
    print v.name, "=", v.varValue
```

The optimised objective function value is printed to the screen, using the value function. This ensures that the number is in the right format to be displayed. `.objective` is an attribute of the object 'prob':

```
# The optimised objective function value is printed to the screen
print "Total Cost of Ingredients per can = ", value(prob.objective)
```

Running this file should then produce the output to show that Chicken will make up 33.33%, Beef will make up 66.67% and the Total cost of ingredients per can is 96 cents.

The segmented .py file shown above can be obtained in full here . Opening the file normally will run it, which will execute the commands and exit in less than 1 second. You will need to open this file with IDLE or PyDev to run it and view the output properly.

Full Formulation

Now we will formulate the problem fully with all the variables. Whilst it could be implemented into Python with little addition to our method above, we will look at a better way which does not mix the problem data, and the formulation as much. This will make it easier to change any problem data for other tests. We will start the same way by algebraically defining the problem:

1. **Identify the Decision Variables**

   For the Whiskas Cat Food Problem the decision variables are the percentages of the different ingredients we include in the can. Since the can is 100g, these percentages also represent the amount in g of each ingredient included. In STATS 255, the decisions were the amount in g in each cat food, but it is more convenient to use percentages.

   We must formally define our decision variables, being sure to state the units we are using.

   x_1 = percentage of chicken meat in a can of cat food
   x_2 = percentage of beef used in a can of cat food
   x_3 = percentage of mutton used in a can of cat food
   x_4 = percentage of rice used in a can of cat food
   x_5 = percentage of wheat bran used in a can of cat food
   x_6 = percentage of gel used in a can of cat food

   Note that these percentages must be between 0 and 100.

2. **Formulate the Objective Function**

   For the Whiskas Cat Food Problem the objective is to minimise the total cost of ingredients per can of cat food.

   We know the cost per g of each ingredient. We decide the percentage of each ingredient in the can, so we must divide by 100 and multiply by the weight of the can in g. This will give us the weight in g of each ingredient.

   $$\min \$0.013x_1 + \$0.008x_2 + \$0.010x_3 + \$0.002x_4 + \$0.005x_5 + \$0.001x_6$$

3. **Formulate the Constraints**

   The constraints for the Whiskas Cat Food Problem are that:
   1. The sum of the percentages must make up the whole can (= 100%).
   2. The stated nutritional analysis requirements are met.

   The constraint for the "whole can" is:

   $$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 100$$

   To meet the nutritional analysis requirements, we need to have at least 8g of Protein per 100g, 6g of fat, but no more than 2g of fibre and 0.4g of salt. To formulate these constraints we make use of the previous table of contributions from each ingredient. This allows us to formulate the following constraints on the total contributions of protein, fat, fibre and salt from the ingredients:

$$
\begin{array}{rrrrrrcr}
0.100x_1 & +0.200x_2 & +0.150x_3 & +0.000x_4 & +0.040x_5 & +0.0x_6 & \geq & 8.0 \\
0.080x_1 & +0.100x_2 & +0.110x_3 & +0.010x_4 & +0.010x_5 & +0.0x_6 & \leq & 6.0 \\
0.001x_1 & +0.005x_2 & +0.003x_3 & +0.100x_4 & +0.150x_5 & +0.0x_6 & \leq & 2.0 \\
0.002x_1 & +0.005x_2 & +0.007x_3 & +0.002x_4 & +0.008x_5 & +0.0x_6 & \leq & 0.4
\end{array}
$$

4. **Identify the Data**

   We know the total weight of the can. We also know the cost of the ingredients, the nutritional analysis requirements and the contribution (per gram) of each ingredient in terms of the nutritional analysis. This is enough to formulate the necessary mathematical programme, but we will reconsider our data after solving the mathematical programme and performing some post-optimal analyis.

Solution to Full Problem

To obtain the solution to this Linear Program, we again write a short program in Python to call PuLP's modelling functions, which will then call a solver. This will explain step-by-step how to write this Python program with it's improvement to the above model. It is suggested that you repeat the exercise yourself.

As with last time, it is advisable to head your file with commenting on it's purpose, and the author name and date. Importing of the PuLP functions is also done in the same way:

```
"""
The Full Whiskas Model Python Formulation for the PuLP Modeller



Authors: Antony Phillips, Dr Stuart Mitchell   2007
"""


# Import PuLP modeler functions
from pulp import *
```

Next, before the "prob" variable or type of problem are defined, the key problem data is entered into dictionaries. This includes the list of Ingredients, followed by the cost of each Ingredient, and it's percentage of each of the four nutrients. These values are clearly laid out and could easily be changed by someone with little knowledge of programming. The ingredients are the reference keys, with the numbers as the data.

```
# Creates a list of the Ingredients
Ingredients = ["CHICKEN", "BEEF", "MUTTON", "RICE", "WHEAT", "GEL"]



# A Dictionary of the costs of each of the Ingredients is created
costs = {"CHICKEN": 0.013,
         "BEEF"   : 0.008,
         "MUTTON" : 0.010,
         "RICE"   : 0.002,
         "WHEAT"  : 0.005,
         "GEL"    : 0.001}


# A dictionary of the protein percent in each of the Ingredients is created
proteinPercent = {"CHICKEN": 0.100,
                  "BEEF"   : 0.200,
                  "MUTTON" : 0.150,
                  "RICE"   : 0.000,
                  "WHEAT"  : 0.040,
                  "GEL"    : 0.000}


# A dictionary of the fat percent in each of the Ingredients is created
fatPercent = {"CHICKEN": 0.080,
              "BEEF"   : 0.100,
              "MUTTON" : 0.110,
              "RICE"   : 0.010,
              "WHEAT"  : 0.010,
              "GEL"    : 0.000}


# A dictionary of the fibre percent in each of the Ingredients is created
fibrePercent = {"CHICKEN": 0.001,
                "BEEF"   : 0.005,
                "MUTTON" : 0.003,
                "RICE"   : 0.100,
                "WHEAT"  : 0.150,
                "GEL"    : 0.000}


# A dictionary of the fibre percent in each of the Ingredients is created
saltPercent = {"CHICKEN": 0.002,
               "BEEF"   : 0.005,
               "MUTTON" : 0.007,
               "RICE"   : 0.002,
               "WHEAT"  : 0.008,
               "GEL"    : 0.000}
```

The "prob" variable is created to contain the formulation, and the usual parameters are passed into `LpProblem`.

```
# Create the prob variable to contain the problem data
prob = LpProblem("The Whiskas Problem", LpMinimize)
```

A dictionary called vars is created which contains the LP variables, with their defined lower bound of zero. The reference keys to the dictionary are the Ingredient names, and the data is Ingr_IngredientName. (e.g. MUTTON: Ingr_MUTTON)

```
# A dictionary called Vars is created to contain the referenced variables
vars = LpVariable.dicts("Ingr",Ingredients,0)
```

Since costs and vars are now dictionaries with the reference keys as the Ingredient names, the data can be simply extracted with a list comprehension as shown. The lpSum function will add the elements of the resulting list. Thus the objective function is simply entered and assigned a name:

```
# The objective function is added to prob first
prob += lpSum([costs[i] * vars[i] for i in Ingredients]), "Total Cost of Ingredients per can"
```

Further list comprehensions are used to define the other 5 constraints, which are also each given names describing them.

```
# The five constraints are added to prob
prob += lpSum([vars[i] for i in Ingredients]) == 100, "PercentagesSum"
prob += lpSum([proteinPercent[i] * vars[i] for i in Ingredients]) >= 8.0 "ProteinRequirement"
prob += lpSum([fatPercent[i] * vars[i] for i in Ingredients]) >= 6.0 "FatRequirement"
prob += lpSum([fibrePercent[i] * vars[i] for i in Ingredients]) <= 2.0, "FibreRequirement"
prob += lpSum([saltPercent[i] * vars[i] for i in Ingredients <= 0.4, "Salt Requirement"
```

Following this, the prob.writeLP line etc follow exactly the same as in the simplified example. To see the entire file, it is available here .

The optimal solution is 60% Beef and 40% Gel leading to a Objective Function value of 52 cents per can.

Post-Optimal Analysis

For The Whiskas Cat Food Problem we will not perform any post-optimal analysis. However, as you will remember from STATS255, a Sensitivity Analysis and a Parametric Analysis can be performed.

Validation

To validate our solution for The Whiskas Cat Food Problem, we can do a quick check that our solution makes sense. First, the percentages should add up to 100%. If not, there is something wrong. Next, we can do a quick check of the constraints to ensure none of them are violated.

Note For large models you won't always be able to check the solution by hand.

The final validation is to write up a *management summary* for your manager and/or client and see if they think our solution is a valid one. If they identify some (or all) of the solution that is not valid, then you should discuss with them the reasons why it is invalid and start a "feedback" loop in your optimisation process.

Presentation of Solution and Analysis

The solution for The Whiskas Cat Food Problem is a simple one to summarise. Here is an example management summary (with the numbers removed):

The Whiskas Cat Food Problem

Stuart Mitchell, 2007

Uncle Ben's want to produce their cat food products as cheaply as possible while ensuring they meet the stated nutritional analysis requirements shown on the cans. Thus they want to vary the quantities of each ingredient used (the main ingredients being chicken, beef, mutton, rice, wheat and gel) while still meeting their nutritional standards.

The stated nutritional analysis requirements are: _____

The cost for each ingredient is: _____

To minimise the cost per 100g can of Whiskas cat food, Whiskas should use
_____ g of Chicken,
_____ g of Beef,

The cost per can is _____

IMPORTANT When presenting your solution you must be careful about the number of decimal places you use. You should not use a greater accuracy than your data allows.

Implementation and Ongoing Monitoring

The first step towards Implementation is a good management summary. You may also want to discuss any issues that may arise from the solution you have found with Uncle Ben's (for example, if you solution can be implemented on their production line).

Ongoing Monitoring may take the form of:

1. Updating your data files and resolving as the data changes (changing costs, nutritional requirements, etc);
2. Resolving our model for new products (e.g., 200g cans);
3. Looking for possible savings (e.g., analysing the cost of the ingredients to see if any price changes will affect the ingredient mix).

 Previous: (4a) Installing PuLP at Home

 Next: (5b) A Transportation Problem

Previous: (5a) A Blending Problem
Next: (5c) A Transshipment Problem

(5b) A Transportation Problem

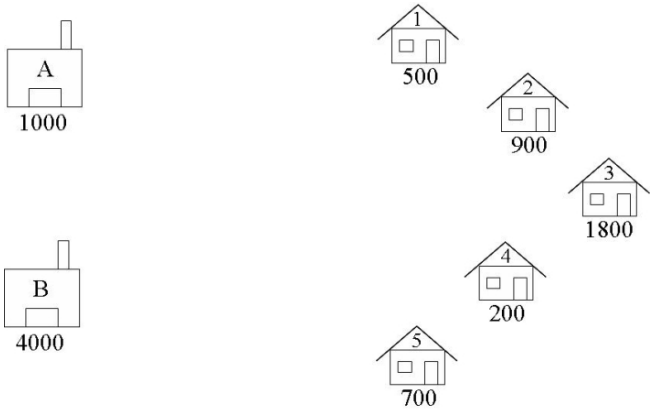
The Beer Distribution Problem

Problem Description

A boutique brewery has two warehouses from which it distributes beer to five carefully chosen bars. At the start of every week, each bar sends an order to the brewery's head office for so many crates of beer, which is then dispatched from the appropriate warehouse to the bar. The brewery would like to have an interactive computer program which they can run week by week to tell them which warehouse should supply which bar so as to minimize the costs of the whole operation. For example, suppose that at the start of a given week the brewery has 1000 cases at warehouse A, and 4000 cases at warehouse B, and that the bars require 500, 900, 1800, 200, and 700 cases respectively. Which warehouse should supply which bar?

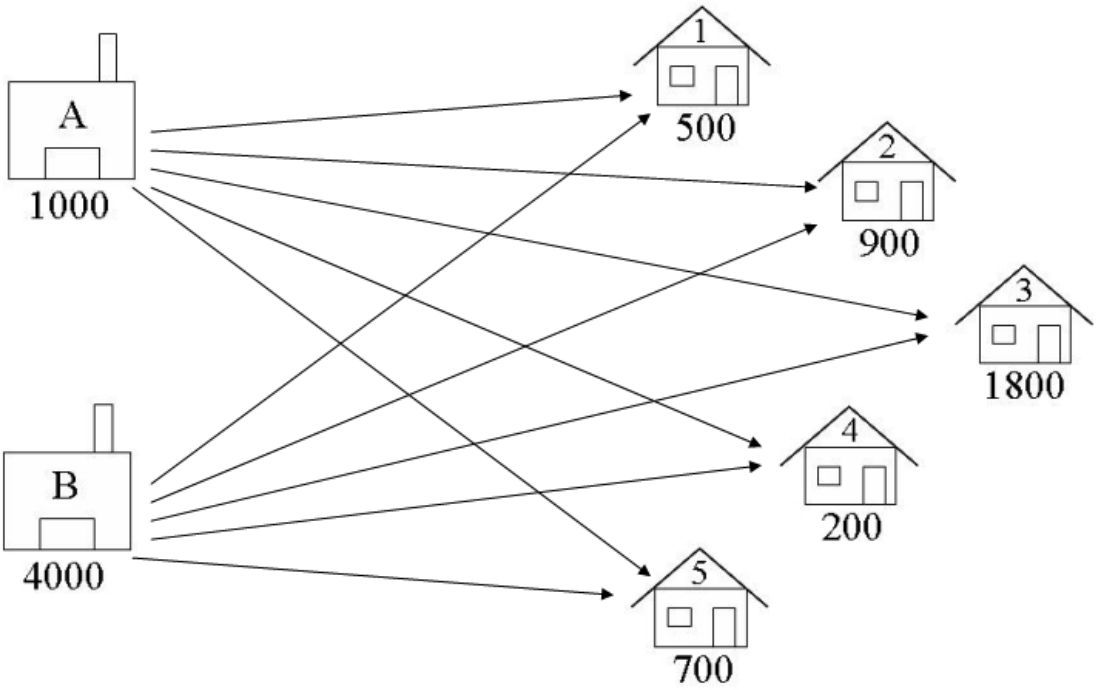
Formulation

For transportation problems, using a graphical representation of the problem is often helpful during formulation. Here is a graphical representation of The Beer Distribution Problem.



1. **Identify the Decision Variables**

   In transportation problems we are deciding how to transport goods from their supply nodes to their demand nodes. The decision variables are the Arcs connecting these nodes, as shown in the diagram below. We are deciding how many crates of beer to transport from each warehouse to each pub.



A1 = number of crates of beer to ship from Warehouse A to Bar 1
.
.
A5 = number of crates of beer to ship from Warehouse A to Bar 5
B1 = number of crates of beer to ship from Warehouse B to Bar 1
.
.
B5 = number of crates of beer to ship from Warehouse B to Bar 5

The lower bound on the variables is Zero, and the values must all be Integers (since the number of crates cannot be negative or fractional). There is no upper bound.

2. **Formulate the Objective Function**

The objective function has been loosely defined as cost. The problem can only be formulated as a linear program if the cost of transportation from warehouse to pub is a linear function of the amounts of crates transported.
Note that this is sometimes not the case. This may be due to factors such as economies of scale or fixed costs. For example, transporting 10 crates may not cost 10 times as much as transporting one crate, since it may be the case that one truck can accommodate 10 crates as easily as one. Usually in this situation there are fixed costs in operating a truck which implies that the costs go up in jumps (when an extra truck is required). In these situations, it is possible to model such a cost by using zero-one integer variables: we will look at this later in the course.

We shall assume then that there is a fixed transportation cost per crate. (If the capacity of a truck is small compared with the number of crates that must be delivered then this is a valid assumption). Lets assume we talk with the financial manager, and she gives us the following transportation costs (dollars per crate):

From **Warehouse** to **Bar**	A	B
1	2	3
2	4	1
3	5	3
4	2	2
5	1	3

Sum of Transporting Costs = Cost per crate for RouteA1 * A1 (number of crates on RouteA1) + ... + Cost per crate for

RouteB5 * B5 (number of crates on RouteB5)

3. **Formulate the Constraints**

The constraints come from considerations of supply and demand. The supply of beer at warehouse A is 1000 cases. The total amount of beer shipped from warehouse A cannot exceed this amount. Similarly, the amount of beer shipped from warehouse B cannot exceed the supply of beer at warehouse B. The sum of the values on all the arcs leading out of a warehouse, must be less than or equal to the supply value at that warehouse:

A1 + A2 + A3 + A4 + A5 <= 1000
B1 + B2 + B3 + B4 + B5 <= 4000

The demand for beer at bar 1 is 500 cases, so the amount of beer delivered there must be at least 500 to avoid lost sales. Similarly, considering the amounts delivered to the other bars must be at least equal to the demand at those bars. Note, we are assuming there are no penalties for oversupplying bars (other than the extra transportation cost we incur). We can *balance* the transportation problem to make sure that demand is met exactly - there will be more on this later. For now, the sum of the values on all the arcs leading into a bar, must be greater than or equal to the demand value at that bar:

A1 + B1 >= 500
A2 + B2 >= 900
A3 + B3 >= 1800
A4 + B4 >= 200
A5 + B5 >= 700

Finally, we have already specified the amount of beer shipped must be non-negative.

Solution

Whilst the LP as defined above could be formulated into Python code in the same way as the Blending Problem (Whiskas), for Transportation Problems, there is a more efficient way which we will use in this course.

First, start your Python file with a heading and the import PuLP statement:

```
"""
The Beer Distribution Problem for the PuLP Modeller



Authors: Antony Phillips, Dr Stuart Mitchell    2007
"""


# Import PuLP modeller functions
from pulp import *
```

The start of the formulation is a simple definition of the nodes and their limits/capacities. The node names are put into lists, and their associated capacities are put into dictionaries with the node names as the reference keys:

```
# Creates a list of all the supply nodes
Warehouses = ["A","B"]



# Creates a dictionary for the number of units of supply for each supply node
supply = {"A": 1000,
          "B": 4000}


# Creates a list of all demand nodes
Bars = ["1", "2", "3", "4", "5"]


# Creates a dictionary for the number of units of demand for each demand node
demand = {"1": 500,
          "2": 900,
          "3": 1800,
          "4": 200,
          "5": 700}
```

The cost data is then inputted into a list, with two sub lists: the first containing the costs of shipping from Warehouse A, and the second containing the costs of shipping from Warehouse B. Note here that the commenting and structure of the code makes the data appear as a table for easy editing. The `Warehouses` and `Bars` lists (Supply and Demand nodes) are added to make a large list (of all nodes) and inputted into PuLPs `makeDict` function. The second parameter is the costs list as was previously created, and the last parameter sets the default value for an arc cost. Once the cost dictionary is created, if 'costs["A"]["1"]' is called, it will return the cost of transporting from warehouse A to bar 1, 2. If 'costs["C"]["2"]' is called, it will return 0, since this is the defined default.

```
# Creates a list of costs of each transportation path
costs = [   #Bars
         #1 2 3 4 5
         [2,4,5,2,1],#A  Warehouses
         [3,1,3,2,3] #B
         ]
```

The `prob` variable is created using the `LpProblem` function, with the usual input parameters.

```
# Creates the prob variable to contain the problem data
prob = LpProblem("Beer Distribution Problem",LpMinimize)
```

A list of tuples is created containing all the arcs.

```
# Creates a list of tuples containing all the possible routes for transport
Routes = [(w,b) for w in Warehouses for b in Bars]
```

A dictionary called Vars is created which contains the LP variables. The reference keys to the dictionary are the warehouse name, then the bar name(["A"]["2"]) , and the data is Route_Tuple. (e.g. ["A"]["2"]: Route_A_2). The lower limit of zero is set, the upper limit of None is set, and the variables are defined to be Integers.

```
# A dictionary called Vars is created to contain the referenced variables (the routes)
vars = LpVariable.dicts("Route",(Warehouses,Bars),0,None,LpInteger)
```

The objective function is added to the variable `prob` using a list comprehension. Since vars and costs are now dictionaries (with further internal dictionaries), they can be used as if they were tables, as `for (w,b) in Routes` will cycle through all the combinations/arcs. Note that `i` and `j` could have been used, but `w` and `b` are more meaningful.

```
# The objective function is added to prob first
prob += lpSum([vars[w][b]*costs[w][b] for (w,b) in Routes]), "Sum of Transporting Costs"
```

The supply and demand constraints are added using a normal *for* loop and a list comprehension. Supply Constraints: For each warehouse in turn, the values of the decision variables (number of beer cases on arc) to each of the bars is summed, and then constrained to being less than or equal to the supply max for that warehouse. Demand Constraints: For each bar in turn, the values of the decision variables (number on arc) from each of the warehouses is summed, and then constrained to being greater than or equal to the demand minimum.

```
# The supply maximum constraints are added to prob for each supply node (warehouse)
for w in Warehouses:
    prob += lpSum([vars[w][b] for b in Bars]) <= supply[w], "Sum of Products out of Warehouse %s"%w



# The demand minimum constraints are added to prob for each demand node (bar)
for b in Bars:
   prob += lpSum([vars[w][b] for w in Warehouses]) >= demand[b], "Sum of Products into Bars %s"%b
```

Following this is the `prob.writeLP` line, and the rest as explained in previous examples. The entire file shown segmented above can be downloaded here.

You will notice that the linear programme solution was also an integer solution. As long as Supply and Demand are integers, the linear programming solution will always be an integer. Read about naturally integer solutions for more details.

Post-Optimal Analysis

Validation

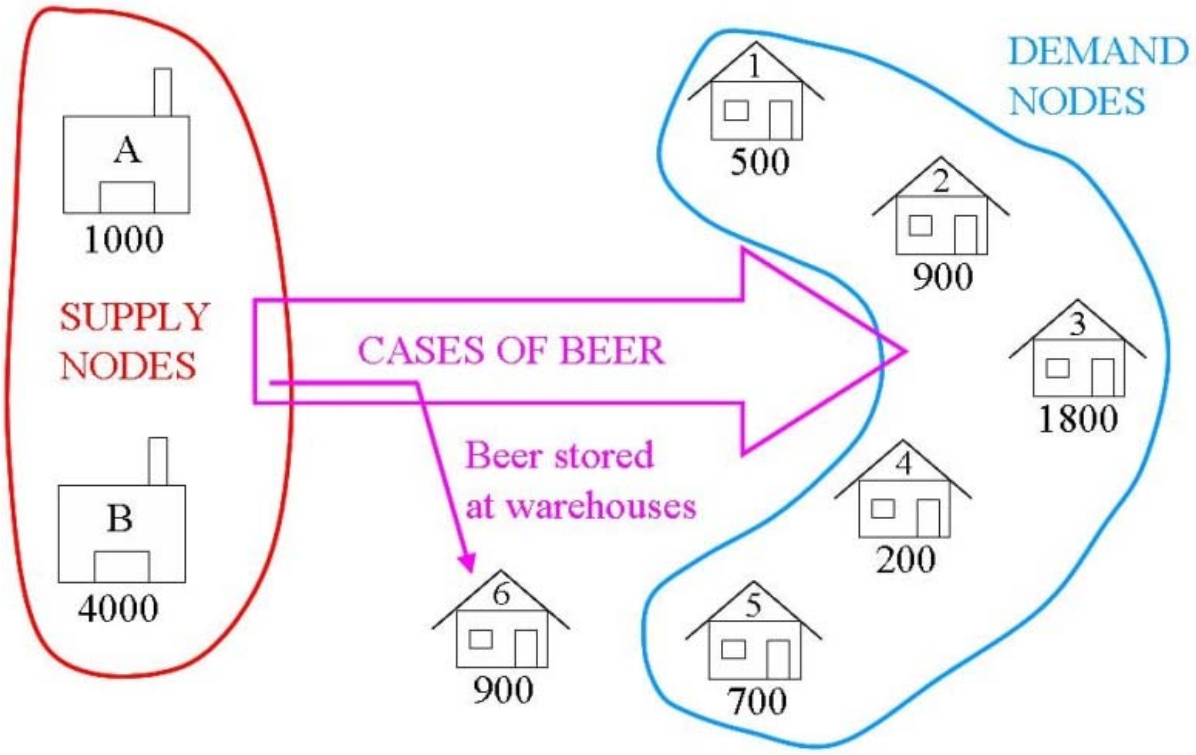
Since we have guaranteed the Supply and Demand are integer, we know that the solution to the linear programme will be integer, so we don't need to check the integrality of the solution.

Storage and "Buying In"

Transportation models are usually *balanced*, i.e., the total supply = the total demand. This is because extra supply usually must be stored somewhere (with an associated storage cost) and extra demand is usually satisfied by purchasing extra goods from alternative sources (this is know as "buying in" extra goods) or by substituting another product (incurring a penalty cost).

In The Beer Distribution Problem, the total supply is 5000 cases of beer, but the total demand is only for 4100 cases. The

extra supply can be sent to an *dummy* demand node. The cost of flow going to the dummy demand node is then the storage cost at each of the supply nodes.



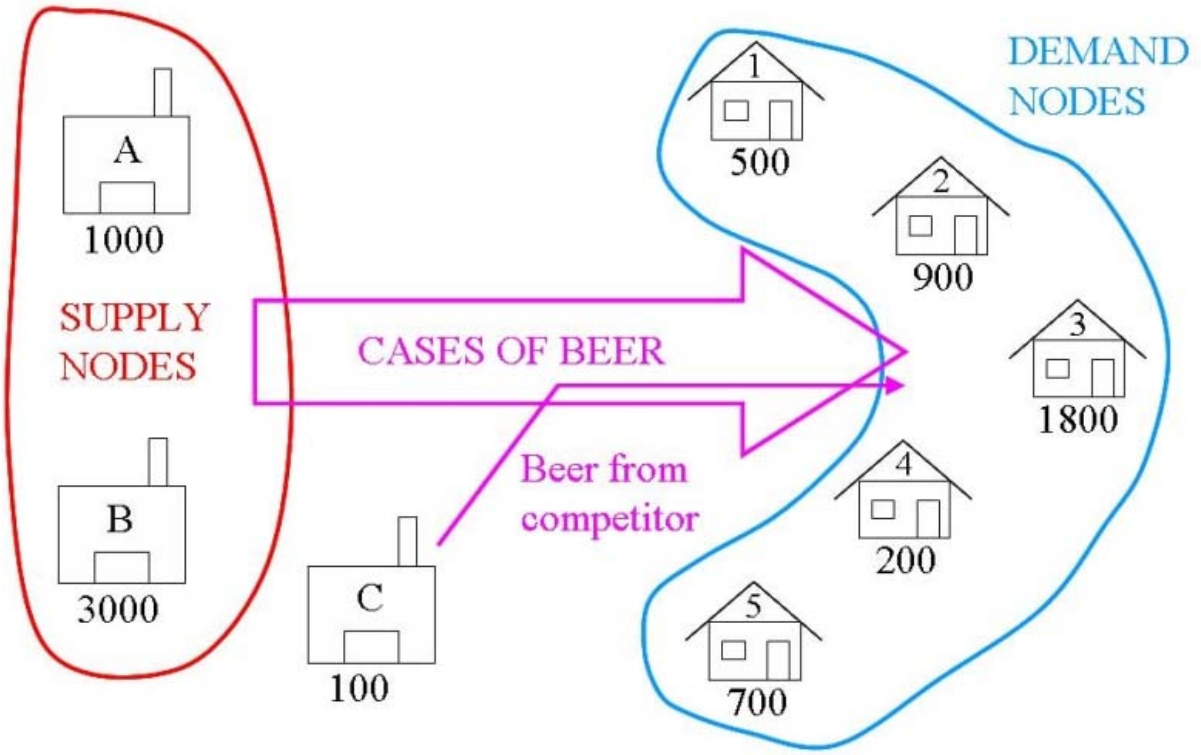
This is added into the above model very simply. Since the objective function and constraints all operated on the original supply, demand and cost lists/dictionaries, the only changes that must be made to include another demand node are:

```
# Creates a list of all demand nodes
Bars = ["1","2","3","4","5","D"]



# Creates a dictionary for the number of units of demand for each demand node
demand = {"1": 500,
          "2": 900,
          "3": 1800,
          "4": 200,
          "5": 700,
          "D": 900}


# Creates a list of costs of each transportation path
costs = [   #Bars
         #1 2 3 4 5 D
         [2,4,5,2,1,0],#A  Warehouses
         [3,1,3,2,3,0] #B
         ]
```

The `Bars` list is expanded and the `Demand` dictionary is expanded to make the Dummy Demand require 900 crates, to balance the problem. The cost list is also expanded, to show the cost of "sending to the Dummy Node" which is realistically just leaving the stock at the warehouses. This may have an associated cost which could be entered here instead of the zeros. Note that the solution could still be solved when there was an unbalanced excess supply.

If a transportation problem has more demand than supply, we can balance the problem using a dummy supply node. Note that with excess demand, the problem is `Infeasible` when unbalanced.

Assume there has been a production problem and only 4000 cases of beer could be produced. Since the total demand is 4100, we need to get extra cases of beer from the dummy supply node.



This dummy supply node is added in simply and logically into the `Warehouse` list, `Supply` dictionary, and `costs` list. The Supply value is chosen to balance the problem, and cost of transport is zero to all demand nodes.

```
# Creates a list of all the supply nodes
Warehouses = ["A","B","D"]



# Creates a dictionary for the number of units of supply for each supply node
supply = {"A": 1000,
          "B": 3000,
          "D": 100}


# Creates a list of all demand nodes
Bars = ["1","2","3","4","5"]


# Creates a dictionary for the number of units of demand for each demand node
demand = {"1": 500,
          "2": 900,
          "3": 1800,
          "4": 200,
          "5": 700}


# Creates a list of costs of each transportation path
costs = [   #Bars
         #1 2 3 4 5 D
         [2,4,5,2,1], #A
         [3,1,3,2,3], #B  Warehouses
         [0,0,0,0,0]  #D
         ]
```

Presentation of Solution and Analysis

There are many ways to present the solution to The Beer Distribution Problem: as a list of shipments, in a table, etc.

```
TRANSPORTATION SOLUTION -- Non-zero shipments
TotalCost = ____

Ship ___ crates of beer from warehouse A to pub 1
Ship ___ crates of beer from warehouse A to pub 5
Ship ___ crates of beer from warehouse B to pub 1
Ship ___ crates of beer from warehouse B to pub 2
Ship ___ crates of beer from warehouse B to pub 3
Ship ___ crates of beer from warehouse B to pub 4
```

This information gives rise to the following management summary:

The Beer Distribution Problem

Mike O'Sullivan, 1234567

We are minimising the transportation cost for a brewery operation. The brewery transports cases of beer from its warehouses to several bars.

The brewery has two warehouses (A and B respectively) and 5 bars (1, 2, 3, 4 and 5).

The supply of crates of beer at the warehouses is:

_____

The forecasted demand (in crates of beer) at the bars is:

_____

The cost of transporting 1 crate of beer from a warehouse to a bar is given in the following table:

_____

To minimise the transportation cost the brewery should make the following shipments:

Ship ___ crates of beer from warehouse A to pub 1 Ship ___ crates of beer from warehouse A to pub 5 Ship ___ crates of beer from warehouse B to pub 1 Ship ___ crates of beer from warehouse B to pub 2 Ship ___ crates of beer from warehouse B to pub 3 Ship ___ crates of beer from warehouse B to pub 4

The total transportation cost of this shipping schedule is $_____.

Ongoing Monitoring

Ongoing Monitoring may take the form of:

1. Updating your data files and resolving as the data changes (changing costs, supplies, demands);
2. Resolving our model for new nodes (e.g., new warehouses or bars);
3. Looking to see if cheaper transportation options are available along routes where the transportation cost affects the optimal solution (e.g., how much total savings can we get by reducing the transportation cost from Warehouse B to Bar 1).


Previous: (5a) A Blending Problem


Next: (5c) A Transshipment Problem

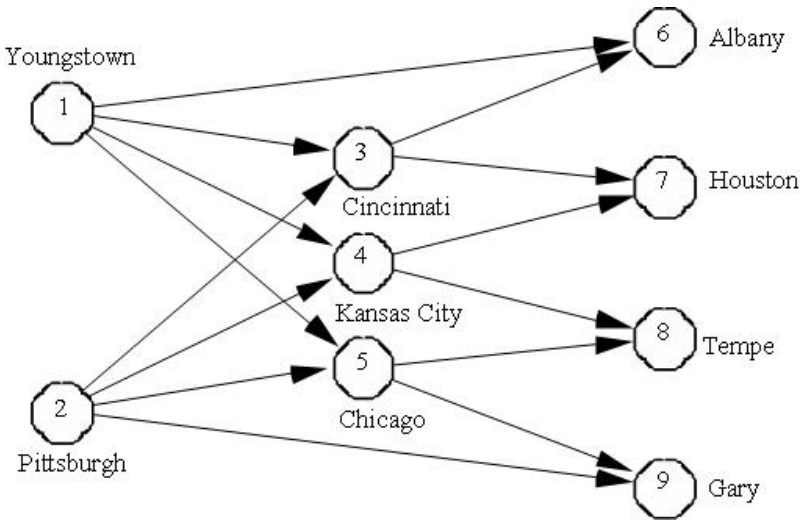
Previous: (5b) A Transportation Problem
Next: (5d) A Facility Location Problem

(5c) A Transshipment Problem

The American Steel Problem

Problem Description

American Steel, an Ohio-based steel manufacturing company, produces steel at its two steel mills located at Youngstown and Pittsburgh. The company distributes finished steel to its retail customers through the distribution network of regional and field warehouses shown below:



The network represents shipment of finished steel from American Steel's two steel mills located at Youngstown (node 1) and Pittsburgh (node 2) to their field warehouses at Albany, Houston, Tempe, and Gary (nodes 6, 7, 8 and 9) through three regional warehouses located at Cincinnati, Kansas City, and Chicago (nodes 3, 4 and 5). Also, some field warehouses can be directly supplied from the steel mills.

The table below presents the minimum and maximum flow amounts of steel that may be shipped between different cities along with the cost per 1000 ton/month of shipping the steel. For example, the shipment from Youngstown to Kansas City is contracted out to a railroad company with a minimal shipping clause of 1000 tons/month. However, the railroad cannot ship more then 5000 tons/month due the shortage of rail cars.

From node	*To node*	*Cost*	*Minimum*	*Maximum*
Youngstown	Albany	500	–	1000
Youngstown	Cincinnati	350	–	3000
Youngstown	Kansas City	450	1000	5000
Youngstown	Chicago	375	–	5000
Pittsburgh	Cincinnati	350	–	2000
Pittsburgh	Kansas City	450	2000	3000
Pittsburgh	Chicago	400	–	4000
Pittsburgh	Gary	450	–	2000
Cincinnati	Albany	350	1000	5000
Cincinnati	Houston	550	–	6000
Kansas City	Houston	375	–	4000
Kansas City	Tempe	650	–	4000
Chicago	Tempe	600	–	2000
Chicago	Gary	120	–	4000

The current monthly demand at American Steel's four field warehouses is as follows:

Field Warehouses	*Monthly Demand*

Albany, N.Y.	3000
Houston	7000
Tempe	4000
Gary	6000

The Youngstown and Pittsburgh mills can produce up to 10,000 tons and 15,000 tons of steel per month, respectively. The management wants to know the least cost monthly shipment plan.

Formulation

1. **Identify the Decision Variables**

   The decision variables for this problem are the same as for the transportation problem, the `Flow` of goods (cases of beer in The Beer Distribution Problem, tons of steels here) through the network. In A Transportation Problem all *arcs* from the supply nodes to the demand nodes existed (although in Forestry Management we used upper bounds to removes some arcs). In The American Steel Problem, the network has *transshipment nodes* and arcs don't exist between all nodes. To cater for this, we explicitly name all our arcs - these are our decision variables.

2. **Formulate the Objective Function**

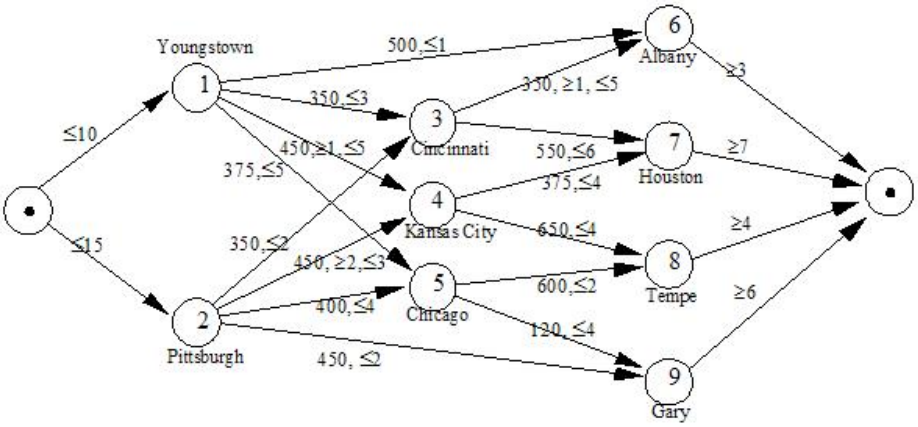
   The objective of transshipment problems in general and The American Steel Problem in particular is to minimise the cost of shipping goods through the network:

3. **Formulate the Constraints**

   All the nodes have supply and demand, demand = 0 for supply nodes, supply = 0 for demand nodes and supply = demand = 0 for transshipment nodes.

   The only constraints in the transshipment problem are *flow conservation* constraints. These constraints simply state that the flow of goods into a node must be greater than or equal to the flow of goods out of a node.

   Transshipment problems are often presented as a network formulation:



Solution

As always, your file will start with an introduction and the import statement.

```
"""
The American Steel Problem for the PuLP Modeller


Authors: Antony Phillips, Dr Stuart Mitchell   2007
"""


# Import PuLP modeller functions
from pulp import *
```

A list of all the node names is created. Whilst the nodes are not the problem variables, they are important in formulating the constraints. A dictionary with each node as the reference key is created, and a list containing the supply and demand of that node is the stored data.

```
# List of all the nodes
Nodes = ["Youngstown",
         "Pittsburgh",
         "Cincinatti",
         "Kansas City",
         "Chicago",
         "Albany",
         "Houston",
         "Tempe",
         "Gary"]



nodeData = {# NODE        Supply Demand
         "Youngstown":    [10000,0],
         "Pittsburgh":    [15000,0],
         "Cincinatti":    [0,0],
         "Kansas City":   [0,0],
         "Chicago":       [0,0],
         "Albany":        [0,3000],
         "Houston":       [0,7000],
         "Tempe":         [0,4000],
         "Gary":          [0,6000]
         }
```

A list of all the arc names is created, and a dictionary containing; the cost per ton of steel sent on that arc, and the bounds, is created. The cost values are in $ per ton, since the other quantities are in tonnes, not thousands of tonnes.

```
# List of all the arcs
Arcs = [("Youngstown","Albany"),
        ("Youngstown","Cincinatti"),
        ("Youngstown","Kansas City"),
        ("Youngstown","Chicago"),
        ("Pittsburgh","Cincinatti"),
        ("Pittsburgh","Kansas City"),
        ("Pittsburgh","Chicago"),
        ("Pittsburgh","Gary"),
        ("Cincinatti","Albany"),
        ("Cincinatti","Houston"),
        ("Kansas City","Houston"),
        ("Kansas City","Tempe"),
        ("Chicago","Tempe"),
        ("Chicago","Gary")]



arcData = { #      ARC                Cost Min Max
        ("Youngstown","Albany"):      [0.5,0,1000],
        ("Youngstown","Cincinatti"):  [0.35,0,3000],
        ("Youngstown","Kansas City"): [0.45,1000,5000],
        ("Youngstown","Chicago"):     [0.375,0,5000],
        ("Pittsburgh","Cincinatti"):  [0.35,0,2000],
        ("Pittsburgh","Kansas City"): [0.45,2000,3000],
        ("Pittsburgh","Chicago"):     [0.4,0,4000],
        ("Pittsburgh","Gary"):        [0.45,0,2000],
        ("Cincinatti","Albany"):      [0.35,1000,5000],
        ("Cincinatti","Houston"):     [0.55,0,6000],
        ("Kansas City","Houston"):    [0.375,0,4000],
        ("Kansas City","Tempe"):      [0.65,0,4000],
        ("Chicago","Tempe"):          [0.6,0,2000],
        ("Chicago","Gary"):           [0.12,0,4000]
        }
```

The `splitDict` function is used to create separate dictionaries from the dictionary inputs which had lists as the stored data. This splits up the lists so that (for example) 'Supply["Youngstown"]' will return `10000`, instead of having to use nodeData["Youngstown"][1] to get the value.

```
# Splits the dictionaries to be more understandable
(supply, demand) = splitDict(nodeData)
(costs, mins, maxs) = splitDict(arcData)
```

The problem variables are created with the usual parameters. "Route" is the arbitrary name for the category the problem variables fall into. The names in the list `Arcs` will form the main part of the problem variable names. It is important to note that both the lower and upper bounds have been set to `None`. This is because the bounds on each of the variables is different and so we cannot express what they individually are, in this line of code. Lastly, we specify that the solution must be integer, since it is fair to assume that each ton is inseparable. This is a certain degree of rounding which is appropriate.

```
# Creates the boundless Variables as Integers
vars = LpVariable.dicts("Route",Arcs,None,None,LpInteger)
```

The bounds on the variables are created by modifying the property `bounds` of the objects in `Vars`. There is no equals sign required for this. It could also have been done using 'Vars[a].lower([Mins[a])' and 'Vars[a].lower([Maxs[a])'

```
# Creates the upper and lower bounds on the variables
for a in Arcs:
    vars[a].bounds(mins[a], maxs[a])
```

The variable `prob` is created to store the problem data.

```
# Creates the prob variable to contain the problem data
prob = LpProblem("American Steel Problem",LpMinimize)
```

The objective function is added to prob. This is simply the cost of sending a tonne down each arc multiplied by the number of tonnes sent down.

```
# Creates the objective function
prob += lpSum([vars[a] * costs[a] for a in Arcs]), "Total Cost of Transport"
```

The constraints are all added in this statement. This ensures that the amount of steel flowing into a node is greater than or equal to the amount exiting. This caters for the fact that this problem is unbalanced - it has excess supply. There is no merit, and added cost, in sending steel to a node and not sending is out, so the excess supply will stay at the mills. The code should read as: The amount of steel the node is creating + the amount of steel on each of the arcs leading into the node must be greater than or equal to the amount of steel demanded by the node plus the amount of steel on each of the arcs exiting the node. Since the Arcs are stored as tuples with the "from" and "to" node names stored, when `for (i,j) in Arcs if j==n` is written, it says that we only look at the Arcs which have "n" (the node we are currently constraining) as the "to" node.

```
# Creates all problem constraints - this ensures the amount going into each node is at least equal to the amount leaving
for n in Nodes:
    prob += (supply[n]+ lpSum([vars[(i,j)] for (i,j) in Arcs if j == n]) >=
             demand[n]+ lpSum([vars[(i,j)] for (i,j) in Arcs if i == n])), "Steel Flow Conservation in Node:%s"%n
```

Now the formulation is complete, the standard end to the code can be written, starting at the prob.writeLP line. The full code is available here.

Post-Optimal Analysis

Validation

For our solution to be valid we need it to be integer. Observing the flow values shows that it is in fact integer. In fact, any network flow problem with integer supplies, demands and arc capacities has naturally integer solutions.

Presentation of Solution and Analysis

There is quite a bit of information to summarise and many ways to present it. Some suggestions include:

1. Summarise the problem as usual and list the shipments that American Steel should make (similar to the transportation problem);
2. Summarise the problem as usual and present a table of shipments that American Steel should make;
3. Draw the network formulation for the problem (being sure to specify what the labels mean). Then draw the actual solution on top of the network formulation. You could colour code flows long arcs to show if they are at their bounds.

Implementation and Ongoing Monitoring

Ongoing monitoring of the supply, demand and bounds will help American Steel to keep making good decisions.

Previous: (5b) A Transportation Problem

Next: (5d) A Facility Location Problem

Previous: (5c) A Transshipment Problem

Next: (5e) A Cutting Stock Problem

(5d) A Facility Location Problem

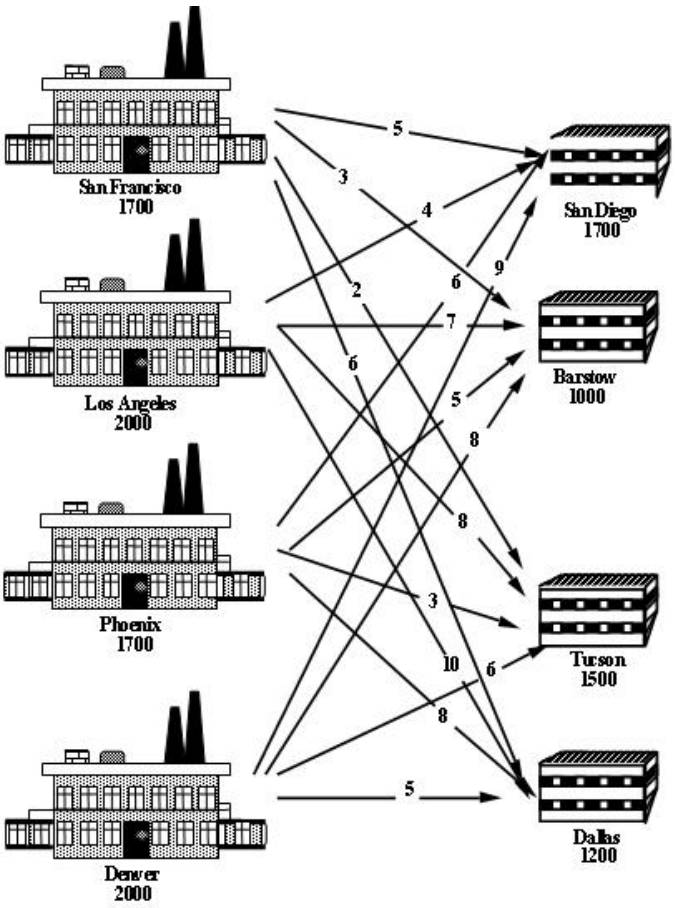
The Computer Plant Problem

Problem Description

Consider the Cosmic Computer company who wish to set up production in America. They are contemplating building production plants in up to 4 locations. Each location has planning restrictions that effectively determine the monthly production possible in each location. The monthly fixed costs of the different locations have been calculated by the accounting department, and are listed below:

Location	Monthly Fixed Cost
San Francisco	$70,000
Los Angeles	$70,000
Phoenix	$65,000
Denver	$70,000

These plants will supply stores at 4 locations. Predicted demand and per-unit transport costs for supplying the 4 production plants to the 4 stores are shown below, along with the quantity each factory can supply:



Where should the company set up their plants to minimise their total costs (fixed plus transportation)?

Formulation

This problem is essentially a transportation problem, but with master-slave constraints at the supply nodes. The lines where this code framework is different/additional to the Transportation Problem code framework will be pointed out.

1. **Identify the Decision Variables**

   The decisions are two-fold:
   1. Where to build the plants?
   2. Where to ship the computers? This is specified by Flow in the Transportation Problem.

2. **Formulate the Objective Function**

   We have already incorporated the transportation cost in the Transportation Problem, now we have to add the fixed (construction + maintenance) cost.

3. **Formulate the Constraints**

   The constraints are the same as in the Transportation Problem, except that the supply is determined by the construction (or not) of a plant:

   Demand at each store must be met. Supply at each factory must be within that plants limitations if it is build, or zero if it is not.

Solution

The introductory commenting and import statement is entered.

```
"""
The Computer Plant Problem for the PuLP Modeller



Authors: Antony Phillips, Dr Stuart Mitchell  2007
"""


#Import PuLP modeller functions
from pulp import *
```

A list of the supply nodes (Plants) is created. The Supply values and Fixed costs of each plant are also entered into a list in a dictionary with the Plant names as the reference keys. (In a Transportation Model, only the Supply values would be entered)

```
# Creates a list of all the supply nodes
Plants = ["San Francisco",
          "Los Angeles",
          "Phoenix",
          "Denver"]



# Creates a dictionary of lists for the number of units of supply at
# each plant and the fixed cost of running each plant
supplyData = {#Plant     Supply  Fixed Cost
          "San Francisco":[1700, 70000],
          "Los Angeles"  :[2000, 70000],
          "Phoenix"      :[1700, 65000],
          "Denver"       :[2000, 70000]
          }
```

A list of the demand nodes (Stores) is created. The Demand values are also entered into a dictionary with the Store names as the reference keys.

```
# Creates a list of all demand nodes
Stores = ["San Diego",
          "Barstow",
          "Tucson",
          "Dallas"]



# Creates a dictionary for the number of units of demand at each store
demand = { #Store    Demand
          "San Diego" :1700,
          "Barstow"   :1000,
          "Tucson"    :1500,
          "Dallas"    :1200}
```

The costs of each of the transportation routes are entered as 4 lists (one for each plant), in a large list. Each of the sub lists contains the transport costs to each of the stores from that plant.

```
# Creates a list of costs for each transportation path
costs = [  #Stores
         #SD BA TU DA
         [5, 3, 2, 6], #SF
         [4, 7, 8, 10],#LA    Plants
         [6, 5, 3, 8], #PH
         [9, 8, 6, 5]  #DE
         ]
```

A list of tuples containing all the arcs in the problem is created.

```
# Creates a list of tuples containing all the possible routes for transport
Routes = [(p,s) for p in Plants for s in Stores]
```

The `supplyData` dictionary is split up into two dictionaries. (additional to the Transportation Problem)

```
# Splits the dictionaries to be more understandable
(supply,fixedCost) = splitDict(supplyData)
```

The cost data is then made into a Dictionary so that 'costs["Phoenix"]["Barstow"]' will return the cost: 5.

```
# The cost data is made into a dictionary
costs = makeDict([Plants,Stores],costs,0)
```

The problem variables are created. This differs from the Transportation Model since two sets of variables must be made, one as usual for the flow and one for whether or not we are to build each plant.

```
# Creates the problem variables of the Flow on the Arcs
flow = LpVariable.dicts("Route",(Plants,Stores),0,None,LpInteger)



# Creates the master problem variables of whether to build the Plants or not
build = LpVariable.dicts("BuildaPlant",Plants,0,1,LpInteger)
```

The variable `prob` is created.

```
# Creates the prob variable to contain the problem data
prob = LpProblem("Computer Plant Problem",LpMinimize)
```

The Objective Function is added to `prob`. This is simply the addition of the transport costs and the fixed costs (the difference from the Tranportation Problem).

```
# The objective function is added to prob - The sum of the transportation costs and the building fixed costs
prob += lpSum([flow[p][s]*costs[p][s] for (p,s) in Routes])+lpSum([fixedCost[p]*build[p] for p in Plants]),"Total Costs"
```

The constraints are added to `prob`. The supply constraint has the 'build[p]' variable in it, since if the plant is not built,

the flow will have to be zero. The demand constraint is standard, ensuring that all demands are met.

```
# The Supply maximum constraints are added for each supply node (plant)
for p in Plants:
    prob += lpSum([flow[p][s] for s in Stores]) <= supply[p]*build[p], "Sum of Products out of Plant %s"%p



# The Demand minimum constraints are added for each demand node (store)
for s in Stores:
    prob += lpSum([flow[p][s] for p in Plants]) >= demand[s], "Sum of Products into Stores %s"%s
```

Following this is the prob.writeLP statement and the standard end to the formulation. The full file is available here. The Optimal Solution for the objective function is $228,100.

Presentation of Solution and Analysis

Your management summary should contain:

1. A brief summary of the problem description;
2. A construction plan for the plants;
3. A transportation plan that describes the shipments from plants to stores;
4. A description of your post-optimal analysis:
    - A brief description of the analysis undertaken (and why);
    - A summary of the results of your analysis;

Implementation and Ongoing Monitoring

Before implementation, sensitivity analysis of both the fixed and transportation costs could identify any data that needs to be confirmed before proceeding. This could also identify any problems should construction costs run over budget.

Ongoing monitoring of the transportation costs, production levels (supply) and demands will produce optimal shiiping plans. If the fixed cost at any plant rises significantly, relocation may be considered (this can also take the form of a facility location problem).


Previous: (5c) A Transshipment Problem


Next: (5e) A Cutting Stock Problem

Previous: (5d) A Facility Location Problem

Next: (5f) Sudoku As An LP

(5e) A Cutting Stock Problem

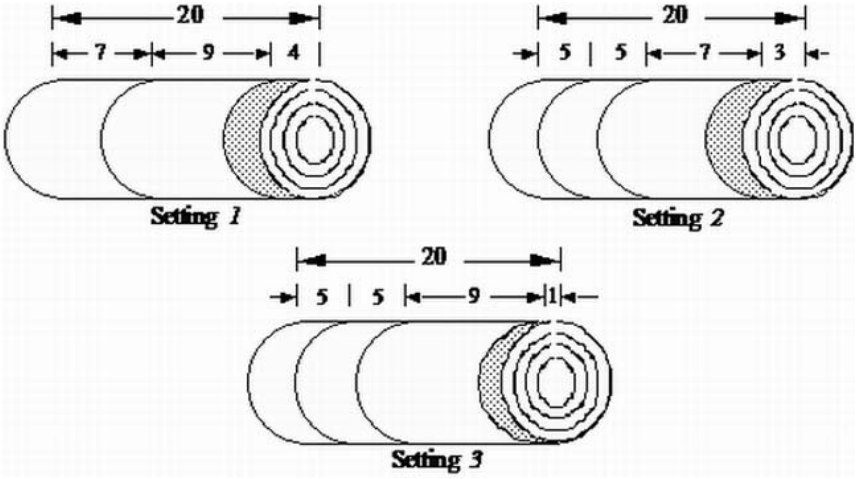
The Sponge Roll Production Problem

Problem Description

The Better Food Company produces cream-filled sponge rolls with a standard width of 20 cm each. Each 20cm roll costs the company $1.00 to produce. Special customer orders with different widths are produced by cutting (slitting) the standard rolls of sponge into shorter lengths. The fixed daily orders are summarized in the following table. These orders need to be met at least cost.

Order	Desired Width (cm)	Desired Number of Rolls
A	5	150
B	7	200
C	9	300

An order is filled by setting the cutting knives to the desired widths. Usually, there are a number of ways in which standard rolls can be slit to fill a given order. The figure below shows three possible knife settings for the 20-cm roll. Although there are other feasible settings, we limit the discussion for the moment to considering settings 1, 2 and 3 in the figure. Note that the shaded area in each diagram represents lengths of sponge that are too short to be used in meeting orders, and so these pieces must be thrown away. Such wastage is called *trim loss*.



The Better Food Company wants to know how to cut the 20cm rolls to minimise the cost of meeting their customer orders.

Formulation

The Sponge Roll Production Problem is an example of a *Cutting Stock* model. Cutting Stock models are a specialisation of *Set Covering* models. The generalised set covering model is:

$$\begin{aligned} &\min \quad c^T x \\ &\text{subject to } Ax \geq b \\ &\qquad x \geq 0, x \text{ integer} \end{aligned}$$

For cutting stock models, each column of the model represents a possible cutting pattern and each constraint represents a requirement for the patterns (in The Sponge Roll Production Problem the requirements are the customer orders). Much of the complexity in cutting stock models (and indeed set covering models in general) is removed during the generation of the columns. Each column represents a feasible pattern, but the complexity of producing feasible patterns is removed from the mathematical programme.

When formulating we will consider two alternatives:

1. The usual formulation (by row);
2. The columnwise formulation.

1. **Identify the Decision Variables**

   The decision variables are the number of times to use each sensible cutting pattern.

2. **Formulate the Objective Function**

Each cutting pattern has an associated cost, in this case the cost of the 20cm roll used (i.e., $1.00). The objective function is the cost of the total number of 20cm rolls required to be cut.

3. **Formulate the Constraints**

The constraints ensure that the desired number of smaller rolls are produced by cutting the 20cm rolls.

Solution

The introductory commenting and import statement is entered.

```
"""
The Sponge Roll Problem for the PuLP Modeller



Authors: Antony Phillips, Dr Stuart Mitchell   2007
"""


# Import PuLP modeller functions
from pulp import *
```

A list of the roll lengths is entered and a dictionary linking the roll lengths to their demand.

```
# A list of all the roll lengths is created
LenOpts = ["5","7","9"]



# A dictionary of the demand for each roll length is created
rollDemand = {"5": 150,
              "7": 200,
              "9": 300}
```

A list of the pattern names we are analysing is entered, and then a list (set out to be viewed as a table) of the number of rolls each pattern can make, for each roll length.

```
# A list of all the patterns is created
PatternNames = ["A","B","C"]



# Creates a list of the number of rolls in each pattern for each different roll length
patterns = [#A B C
            [0,2,2],# 5
            [1,1,0],# 7
            [1,0,1] # 9
            ]
```

The cost of each 20cm roll to be cut up at $1 is entered.

```
# The cost of each 20cm long sponge roll used
cost = 1
```

The pattern table is made into a dictionary (as has been done in previous examples), so that 'Patterns["7"]["B"]' will return the number of rolls of length 7 when cutting according to pattern B.

```
# The pattern data is made into a dictionary
patterns = makeDict([LenOpts,PatternNames],patterns,0)
```

The problem variables are created and will be of the form `Patt_A, Patt_B, Patt_C,` due to the first two parameters. A negative number of rolls cut into the patterns cannot happen, so the lower bound is zero, and there is no upper bound. The number of 20cm rolls cut up into patterns must also be an integer.

```
# The problem variables of the number of each pattern to make are created
vars = LpVariable.dicts("Patt",PatternNames,0,None,LpInteger)
```

The variable `prob` is created.

```
# The variable prob is created
prob = LpProblem("Cutting Stock Problem",LpMinimize)
```

The objective function is added to `prob`. This is the number of each pattern used multiplied by the the cost of each 20cm roll cut up.

```
# The objective function is entered - the total number of large rolls used * the fixed cost of each
prob += lpSum([vars[i]*cost for i in PatternNames]),"Production Cost"
```

The constraints are added to `prob`. In this case, the only constraints are ensuring that the demand for each shorter roll length is met.

```
# The demand minimum constraint is entered
for i in rollLengths:
    prob += lpSum([vars[j]*patterns[i][j] for j in PatternNames])>=rollDemand[i],"Ensuring enough %s cm rolls"%i
```

After the constraints is the usual end to the Python formulation starting with the `prob.writeLP` line. The full working file is available here.

Post-Optimal Analysis

One common extension for cutting stock problems is the sale of any extra products and/or the trim loss at discounted prices. For example, The Better Food Company can sell any excess sponge rolls for: 15c for 5cm rolls, 25c for 7cm rolls and 35c for 9cm rolls. They can also get rid of any trim loss for 4c per cm.

To incorporate the selling of excess sponge rolls, the following is added/changed:

Since each roll length now has both an associated demand and an associated surplus cost, we put these into a list in a dictionary and then use splitDict. These two sections of code replace the `rollDemand` dictionary from the simple formulation. The `surplusPrice` dictionary can now be used.

```
 # A dictionary of the demand and surplus sale price of each roll length is created
rollData = {#Length Demand SalePrice
             "5":   [150,   0.25],
             "7":   [200,   0.33],
             "9":   [300,   0.40]}



# The rollData is made into separate dictionaries
(rollDemand,surplusPrice) = splitDict(rollData)
```

A new set of problem variables need to be created, and so the `vars` dictionary in the problem above needs to be more specifically renamed `pattVars`. The new additional variable dictionary is `surplusVars`, linking the the keys of the roll lengths ("5","7" and "9") to the variables `Surp_5`, `Surp_7` and `Surp_9` which are the number of rolls of surplus of each length.
The objective function has another lpSum term which is subtracted from the original. This term represents the sum of the money made from selling the surplus of each roll length. Lastly, an additional term is put into the demand constraint equation, because each roll that is counted as surplus cannot be used to fill the roll demand. [Note: the slash \ is just a line continuation operator and has only been included so the line image could fit onto the screen. Using PyDev or IDLE, this will not be needed since there is a horizontal scroll]

```
# The problem variables of the number of surplus rolls for each length are created
surplusVars = LpVariable.dicts("Surp",rollLengths,0,None,LpInteger)
```

```
# The objective function is entered - the total number of large rolls used * the fixed cost of each minus the surplus sales
prob += lpSum([pattVars[i]*cost for i in PatternNames]) - \
        lpSum([surplusVars[i]*surplusPrice[i] for i in LenOpts]),"Net Production Cost"



# The demand minimum constraint is entered
for i in LenOpts:
    prob += lpSum([pattVars[j]*patterns[i][j] for j in PatternNames]) - surplusVars[i]\
    >= rollDemand[i],"Ensuring enough %s cm rolls"%i
```

Running this successfully (before adding trim loss) will result in a objective function value of 287.5

The trim loss is incorporated into the model differently to the surplus since the trim sold in each pattern is exactly known before the model runs, and the value of trim sold is simply based on how rolls were cut into each pattern.
Firstly, the sale value for each centremetre of trim is entered and the number of centremetres of trim in each pattern is added to a dictionary. This could actually be calculated using loops, but we have explicitly entered it instead, for this simple model with only 3 pattern choices.

```
# A dictionary of the number of cms of trim in each pattern is created
trim = {"A": 4,
        "B": 2,
        "C": 1}
```

```
# The sale value of each cm of trim
trimValue = 0.04
```

Lastly, a trim term is put into the objective function after the surplus term. It represents the value of the trim sold and has the meaning: the number of each pattern created multiplied by the number of centremetres of trim caused by using that pattern multiplied by the value of each centremetre of trim.

```
# The objective function is entered - the total number of large rolls used * the fixed cost of each minus the surplus sales, and the trim sales
prob += lpSum([pattVars[i]*cost for i in PatternNames]) - lpSum([surplusVars[i]*surplusPrice[i] for i in LenOpts]) \
- lpSum([pattVars[i]*trim[i]*trimValue for i in PatternNames]),"Net Production Cost"
```

This should give you an objective function value of $251.5

The full working file with surplus and trim included is available here.

Validation

We need to make sure the number of each pattern cut is integer. We should check to make sure we have met our demand correctly, i.e., ensure the patterns are correctly defined. We should make sure that the number of rolls cut in a pattern will fit into a 20cm roll, i.e., the pattern is

valid.

Presentation of Solution and Analysis

Your management summary for The Sponge Roll Production should contain a brief problem description, some discussion on your method for generating cutting patterns and a description of the patterns used. It should then present a summary of the optimal use of the cutting patterns.

Implementation and Ongoing Monitoring

To implement your solution you should ensure that the cutting patterns you are using can be set up of the cutting machines. You could also check for hidden set-up costs, i.e., costs incurred by switching from one cutting pattern to the next. Ongoing monitoring should take the form of consistent checking of the product line. Introducing new products means that you will need to regenerate your cutting patterns. As usual, monitoring the costs and demands for any beneficial opportunities should take place.

Extra for Experts

In the above problem we only analysed 3 different patterns for cutting the sponge rolls, whereas there are in fact many different cutting patterns that are worth considering, as you will remember from STATS 255. All the patterns should be considered and the rest for this case can be calculated simply by hand. However, supposing the sponge roll was longer, or many different lengths were required instead of just three, it would take too long to solve by hand. The calculation of these other patterns can be done in Python at the start of your code so you do not need to explicitly enter them.

This is done by using 2 logical functions:

The first is the function that calculates the patterns themselves using a series of for loops. This function will only work for length options lists of size 3. The code works quite simply:
`j` represents the number of length 5 rolls to be cut, `k` represents the number of length 7 rolls to be cut and `l` represents the number of length 9 rolls to be cut. The range of each of these loops is the logical range that `j`,`k` and `l` can take. That is, a minimum value of zero, and a maximum value of how many of that length could be cut if the full length of the roll was cut into pieces that size. e.g. for `j`, the maximum possible value is 4 which is computed with 'int(totalRollLength/lenOpts[1])'. The `int` function rounds the result of the inner calculation down to the nearest integer (i.e. it truncates off the decimal part). The `+1` ensures that the top value is reached when iterating through the for loop.

Inside the for loops, the sum of the lengths for that combination is calculated. If this sum is less than or equal to the total roll length then the cutting pattern is possible. In STATS255, the patterns with trim of 5 or more would not have been used, however there is no harm in having such patterns at [0,0,0] whereby there is 20cm trim and no rolls. Therefore they are calculated along with the other patterns. Supposing the trimValue was greater than 0.05, and the cost of each roll was still 1.00, this would actually make the problem have a negative infinity cost, since the [0,0,0] pattern would make money.

```
def calculatePatterns(totalRollLength,lenOpts):
    """
     The patterns are created using for loops

     The inputs are:
     totalRollLength - the length of the roll
     lenOpts - a list of the sizes of cutting options


     It returns the list of patterns


     Authors: Antony Phillips, Dr Stuart Mitchell    2007
    """


    patterns =[]


    # All possible combinations of cutting patterns are trialed, and the feasible/sensible patterns are added to the list
    for j in range (0,int(totalRollLength/lenOpts[0])+1):
        for k in range (0, int(totalRollLength/lenOpts[1])+1):
            for l in range (0, int(totalRollLength/lenOpts[2])+1):
                lengthSum = j*lenOpts[0]+k*lenOpts[1]+l*lenOpts[2]
                if lengthSum <= totalRollLength:
                    patterns += [[j,k,l]]


    return patterns
```

Since the patterns are now calculated, they need to be assigned names and a trim value automatically too. This is done using another function which calls the calculatePatterns function. Whilst both functions could have been combined into one (or we could have even used no functions), it is much simpler for the reader to interpret it this way.

Apart from calling calculatePatterns, the makePatterns function does 3 key things:
- Creates a name list to correspond to each of the patterns (P0,P1...)
- Calculates the trim for each pattern and puts it in a dictionary, with the pattern name as the reference key
- Prints the name of the patterns next to the list of the number of rolls of each length in that pattern

```
def makePatterns(totalRollLength,lenOpts):
    """
     Makes the different cutting patterns for a cutting stock problem.

     The inputs are:
     totalRollLength : the length of the roll
     lenOpts: a list of the sizes of cutting options as strings

     Authors: Antony Phillips, Dr Stuart Mitchell    2007
    """

    # calculatePatterns is called to create a list of the feasible cutting options in tlist
    patterns = calculatePatterns(totalRollLength,lenOpts)

    # The list PatternNames is created
    PatternNames = []
    for i in range(len(patterns)):
        PatternNames += ["P"+str(i)]

    # The amount of trim (unused material) for each pattern is calculated and added to the dictionary
    # trim, with the reference key of the pattern name.
    trim = {}
    for name,pattern in zip(PatternNames,patterns):
        ssum = 0
        for rep,l in zip(pattern,lenOpts):
            ssum += rep*l
        trim[name] = totalRollLength - ssum
    # The different cutting lengths are printed, and the number of each roll of that length in each
    # pattern is printed below. This is so the user can see what each pattern contains.
    print "Lens: %s" %lenOpts
    for name,pattern in zip(PatternNames,patterns):
        print name + "  = %s"%pattern

    return (PatternNames,patterns,trim)
```

The final changes must be made to the main code to use these functions. PatternNames, patterns and trim no longer need to be created explicitly in the main function and instead this line is put in:

```
(PatternNames,patterns,trim) = makePatterns(totalRollLength,[int(l) for l in LenOpts])
```

The second argument is the length options list, but it is altered so that all the values are integers instead of strings.

The last alteration that must be made is the argument order in the makeDict function. This is because when our patterns list is created using a function, it is actually transposed from the way we entered it explicitly in the first example. The makeDict line should be:

```
patterns = makeDict([PatternNames,LenOpts],patterns,0)
```

Since the code above has been written using functions, any function can be altered to be more efficient or versatile. In the above formulation, the only factor limiting LenOpts lists to length 3 is the calculatePatterns function. This function can be made recursive so it can calculate the pattern list for an input LenOpts of any size. An example of how this can be done is shown below: (when this is called, pass an empty list '[]' as the starting head variable)

The full working code using this recursive function is available here. The objective function value is now $246.5

```
def calculatePatterns(totalRollLength,lenOpts, head):
    """
     Recursively calculates the list of options lists for a cutting stock problem. The input
     tlist is a pointer, and will be the output of the function call.

     The inputs are:
     totalRollLength - the length of the roll
     lenOpts - a list of the sizes of remaining cutting options
     head - the current list that has been passed down though the recusion

     Returns the list of patterns

     Authors: Bojan Blazevic, Dr Stuart Mitchell    2007
    """
    if lenOpts:
        patterns =[]
        #take the first option off lenOpts
        opt = lenOpts[0]
        for rep in range(int(totalRollLength/opt)+1):
            #reduce the length
            l = totalRollLength - rep*opt
            h = head[:]
            h.append(rep)

            patterns.extend(calculatePatterns(l, lenOpts[1:], h))
    else:
        #end of the recursion
        patterns = [head]
    return patterns
```

Lastly, for people who prefer using an Object-Oriented approach, it is possible to define a Pattern class. This object-oriented method is used later in the Column Generation sections, and it is a good idea to understand how it works. This code can be found here.

 Previous: (5d) A Facility Location Problem


Next: (5f) Sudoku As An LP

Previous: (5e) A Cutting Stock Problem

Next: (5g) Column Generation

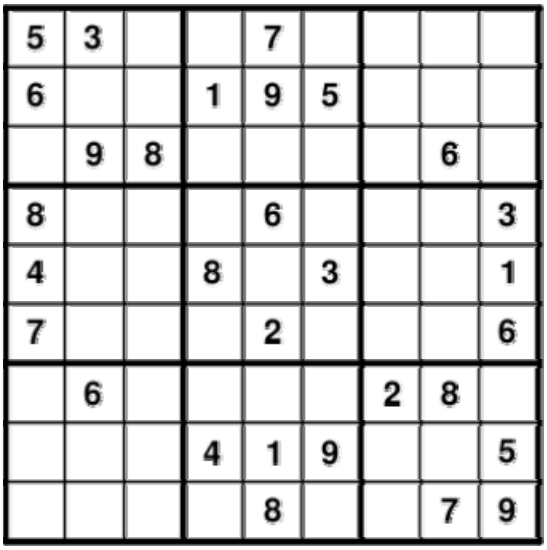
(5f)A Sudoku Problem formulated as an LP

Problem Description

A sudoku problem is a problem where there are is an incomplete 9x9 table of numbers which must be filled according to several rules:

Within any of the 9 individual 3x3 boxes, each of the numbers 1 to 9 must be found
Within any column of the 9x9 grid, each of the numbers 1 to 9 must be found
Within any row of the 9x9 grid, each of the numbers 1 to 9 must be found

On this page we will formulate the below problem from wikipedia to model using PuLP. Once created, our code will need little modification to solve any sudoku problem at all.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Formulation

1. **Identify the Decision Variables**

   In order to formulate this problem as a linear program, we cannot simply create a variable for each of the 81 squares between 1 and 9 representing the value in that square. This is because in linear programming there is no "not equal to" operator and so we cannot use the necessary constraints of no squares within a box/row/column being equal in value to each other. Whilst we can ensure the sum of all the values in a box/row/column equal 45, this will still result in many solutions satisfying the 45 constraint but still with 2 of the same number in the same box/row/column.

   Instead, we must create 729 individual binary (0-1) problem variables. These represent 9 problem variables per square for each of 81 squares, where the 9 variables each correspond to the number that might be in that square. The binary nature of the variable says whether the existence of that number in that square is true or false. Therefore, there can clearly be only 1 of the 9 variables for each square as true (1) and the other 8 must be false (0) since only one number can be placed into any square. This will become more clear.

2. **Formulate the Objective Function**

   Interestingly, with sudoku there is no solution that is *better* than another solution, since a solution by definition, must satisfy all the constraints. Therefore, we are not really trying to minimise or maximise anything, we are just trying to find the values on our variables that satisfy the constraints. Therefore, whilst either LpMinimize or LpMaximize must be entered, it is not important which. Similarly, the objective function can be anything, so in this example it is simply zero.

i.e we are trying to minimize zero, subject to our constraints (meeting the constraints being the important part)

3. **Formulate the Constraints**

   These are simply the known constraints of a sudoku problem plus the constraints on our own created variables we have used to express the features of the problem:
   - The values in the squares in any row must be each of 1 to 9
   - The values in the squares in any column must be each of 1 to 9
   - The values in the squares in any box must be each of 1 to 9 (a box is one of the 9 non-overlapping 3x3 grids within the overall 9x9 grid)
   - There must be only one number within any square (seems logically obvious, but it is important to our formulation to ensure because of our variable choices)
   - The starting sudoku numbers must be in those same places in the final solution (this is a constraint since these numbers are not changeable in the actual problem, whereas we can control any other numbers. If none or very few starting numbers were present, the sudoku problem would have a very large number of feasible solutions, instead of just one)

Solution

The introductory commenting and import statement are entered

```
"""
The Sudoku Problem Formulation for the PuLP Modeller



Authors: Antony Phillips, Dr Stuart Mitchell  2007
"""


# Import PuLP modeler functions
from pulp import *
```

In the unique case of the sudoku problem, the row names, column names and variable option values are all the exact same list of numbers (as strings) from "1" to "9".

```
# A list of strings from "1" to "9" is created
Sequence = ["1","2","3","4","5","6","7","8","9"]



# The Vals, Rows and Cols sequences all follow this form
Vals = Sequence


Rows = Sequence


Cols = Sequence
```

A list called `Boxes` is created with 9 elements, each being another list. These 9 lists correspond to each of the 9 boxes, and each of the lists contains tuples as the elements with the row and column indices for each square in that box. Fully explicitly entering the values in a similar way to the following would have had the same effect (but would have been a waste of time): Boxes = [[("1","1"),("1","2"),("1","3"),("2","1")...("3","3")], [("1","4")...("3","6")], [("1","7")...("3","9")], [("4","1")...("6","3")], ...]
Therefore, Boxes[0] will return a list of tuples containing the locations of each of the 9 squares in the first box.

```
# The boxes list is created, with the row and column index of each square in each box
Boxes =[]
for i in range(3):
    for j in range(3):
        Boxes += [[(Rows[3*i+k],Cols[3*j+l]) for k in range(3) for l in range(3)]]
```

The prob variable is created to contain the problem data. LpMinimize has the same effect as LpMaximise in this case.

```
# The prob variable is created to contain the problem data
prob = LpProblem("Sudoku Problem",LpMinimize)
```

The 729 problem variables are created since the `(Vals,Rows,Cols)` creates a variable for each combination of value, row and column. An example variable would be: Choice_4_2_9, and it is defined to be a binary variable (able to take only

the integers 1 or 0. If Choice_4_2_9 was 1, it would mean the number 4 was present in the square situated in row 2, column 9. (If it was 0, it would mean there was not a 4 there)

```
# The problem variables are created
vars = LpVariable.dicts("Choice",(Vals,Rows,Cols),0,1,LpInteger)
```

As explained above, the objective function (what we try to change using the problem variables) is simply zero (constant) since we are only concerned with any variable combination that can satisfy the constraints.

```
# The arbitrary objective function is added
prob += 0, "Arbitrary Objective Function"
```

Since there are 9 variables for each square, it is important to specify that only exactly one of them can take the value of "1" (and the rest are "0"). Therefore, the below code reads: for each of the 81 squares, the sum of all the 9 variables (each representing a value that could be there) relating to that particular square must equal 1.

```
# A constraint ensuring that only one value can be in each square is created
for r in Rows:
    for c in Cols:
        prob += lpSum([vars[v][r][c] for v in Vals]) == 1, ""
```

These constraints ensure that each number (value) can only occur once in each row, column and box.

```
# The row, column and box constraints are added for each value
for v in Vals:
    for r in Rows:
        prob += lpSum([vars[v][r][c] for c in Cols]) == 1,""

    for c in Cols:
        prob += lpSum([vars[v][r][c] for r in Rows]) == 1,""


    for b in Boxes:
        prob += lpSum([vars[v][r][c] for (r,c) in b]) == 1,""
```

The starting numbers are entered as constraints i.e a "5" in row "1" column "1" is true.

```
# The starting numbers are entered as constraints
prob += vars["5"]["1"]["1"] == 1,""
prob += vars["6"]["2"]["1"] == 1,""
prob += vars["8"]["4"]["1"] == 1,""
prob += vars["4"]["5"]["1"] == 1,""
prob += vars["7"]["6"]["1"] == 1,""
prob += vars["3"]["1"]["2"] == 1,""
prob += vars["9"]["3"]["2"] == 1,""
prob += vars["6"]["7"]["2"] == 1,""
prob += vars["8"]["3"]["3"] == 1,""
prob += vars["1"]["2"]["4"] == 1,""
prob += vars["8"]["5"]["4"] == 1,""
prob += vars["4"]["8"]["4"] == 1,""
prob += vars["7"]["1"]["5"] == 1,""
prob += vars["9"]["2"]["5"] == 1,""
prob += vars["6"]["4"]["5"] == 1,""
prob += vars["2"]["6"]["5"] == 1,""
prob += vars["1"]["8"]["5"] == 1,""
prob += vars["8"]["9"]["5"] == 1,""
prob += vars["5"]["2"]["6"] == 1,""
prob += vars["3"]["5"]["6"] == 1,""
prob += vars["9"]["8"]["6"] == 1,""
prob += vars["2"]["7"]["7"] == 1,""
prob += vars["6"]["3"]["8"] == 1,""
prob += vars["8"]["7"]["8"] == 1,""
prob += vars["7"]["9"]["8"] == 1,""
prob += vars["3"]["4"]["9"] == 1,""
prob += vars["1"]["5"]["9"] == 1,""
prob += vars["6"]["6"]["9"] == 1,""
prob += vars["5"]["8"]["9"] == 1,""
prob += vars["9"]["9"]["9"] == 1,""
```

The problem is written to an LP file, solved using CPLEX (due to CPLEX's simple output) and the solution status is printed to the screen

```
# The problem data is written to an .lp file
prob.writeLP("Sudoku.lp")



# The problem is solved using CPLEX
prob.solve(CPLEX())


# The status of the solution is printed to the screen
print "Status:", LpStatus[prob.status]
```

Instead of printing out all 729 of the binary problem variables and their respective values, it is most meaningful to draw the solution in a text file. The code also puts lines inbetween every third row and column to make the solution easier to read. The sudokuout.txt file is created in the same folder as the .py file.

```
# A file called sudokuout.txt is created/overwritten for writing to
sudokuout = open(sudokuout.txt,w)



# The solution is written to the sudokuout.txt file
for r in Rows:
    if r == "1" or r == "4" or r == "7":
                    sudokuout.write("+-------+-------+-------+\n")
    for c in Cols:
        for v in Vals:
            if value(vars[v][r][c])==1:

                if c == "1" or c == "4" or c =="7":
                    sudokuout.write("| ")


                sudokuout.write(v + " ")


                if c == "9":
                    sudokuout.write("|\n")




sudokuout.write("+-------+-------+-------+")


sudokuout.close()
```

A note of the location of the solution is printed to the solution

```
# The location of the solution is give to the user
print "Solution Written to sudokuout.txt",
```

The full file above is given provided here.

The final solution should be the following:

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Extra for Experts

In the above formulation we did not consider the fact that there may be multiple solutions if the sudoku problem is not well defined.

We can make our code return all the solutions by editing our code as shown after the prob.writeLP line. Essentially we are just looping over the solve statement, and each time after a successful solve, adding a constraint that the same solution cannot be used again. When there are no more solutions, our program ends.

```
# A variable containing the status on whether to continue looking for solutions
continuesolving = 1



while continuesolving == 1:
    prob.solve(CPLEX())

    # The status of the solution is printed to the screen
    print "Status:", LpStatus[prob.status]


    # The solution is printed if it was deemed "optimal" i.e met the constraints
    if LpStatus[prob.status] == "Optimal":

        # The solution is written to the sudokuout.txt file
        for r in Rows:
            if r == "1" or r == "4" or r == "7":
                        sudokuout.write("+-------+-------+-------+\n")
            for c in Cols:
                for v in Vals:
                    if value(vars[v][r][c])==1:

                        if c == "1" or c == "4" or c =="7":
                            sudokuout.write("| ")


                        sudokuout.write(v + " ")


                        if c == "9":
                            sudokuout.write("|\n")




        sudokuout.write("+-------+-------+-------+\n\n")


        # The constraint is added that the same solution cannot be returned again
        prob += lpSum([vars[v][r][c] for v in Vals for r in Rows for c in Cols if value(vars[v][r][c])==1]) <= 80,""




    # If a new optimal solution cannot be found, the loop is made to break and to end the program
    else: continuesolving = 0




sudokuout.close()


# The location of the solutions is give to the user
print "Solutions Written to sudokuout.txt"
```

The full file using this is available here. When using this code for sudoku problems with a large number of solutions, it could take a *very* long time to solve them all. To create sudoku problems with multiple solutions from unique solution sudoku problem, you can simply delete a starting number constraint. You may find that deleting several constraints will still lead to a single optimal solution but the removal of one particular constraint leads to a sudden dramatic increase in the number of solutions.

 Previous: (5e) A Cutting Stock Problem

 Next: (5g) Column Generation

Previous: (5f) Sudoku As An LP

Next: (5h) Column Generation 2

(5g) Column Generation

Column Generation is an efficient way of solving large LPs. In the Cutting Stock Problem we created every possible pattern for lengths that could be cut from the 20cm rolls, and made each pattern a problem variable. However, if the roll length had been 100m long (for example), this would have required over 700 patterns to be created (with the same cutting options of 5cm, 7cm and 9cm). To overcome this problem we can initially solve the problem by explicitly defining just a few patterns to be considered. Then, given the dual variable values from this first solution, we can formulate a sub-problem to find if any other patterns have a negative reduced cost. This means that the new pattern would further reduce the objective function value. The main problem is re-solved with this new pattern and the dual values from this solve are used to find another pattern. When the reduced cost is greater than or equal to zero, the optimal solution is found since the objective function cannot be reduced any further.

A good explanation of the above process is under Delayed Column Generation at Wikipedia.

To implement the Column Generation process, 3 blocks of code are required to be written: A main file, a masterSolve function and a subSolve function. It is a good idea to create a separate file for the functions and the Pattern class definition.

Main File

A file header explains the purpose of the program and identifies the Author/s. The main file only needs to import the CG.py function library, and not PuLP. This is because the main file itself does not use any PuLP functions.

```
"""
The Sponge Roll Problem with Column Generation for the PuLP Modeller



Authors: Antony Phillips,  Dr Stuart Mitchell  2008
"""


# Import Column Generation functions
from CG import *
```

Since a loop is required later on to ensure more patterns are searched for until there are none that would reduce the objective function, the morePatterns variable must be initially set to True. The list of starting patterns must also be entered. Any starting patterns are adequate as long as the problem is feasible/solvable.

```
# The roll data is created
rollData = {#Length Demand SalePrice
            "5":   [150,   0.25],
            "7":   [200,   0.33],
            "9":   [300,   0.40]}



# The boolean variable morePatterns is set to True to test for more patterns
morePatterns = True


# A list of starting patterns is created
patternslist = [[4,0,0],[0,2,0],[0,0,2] ]
```

Each of the starting patterns are used to create an object of the class Pattern, in the list Patterns. This class is defined in the CG.py function file.

```
# The starting patterns are instantiated with the Pattern class
Patterns = []
for i in patternslist:
    Patterns += [Pattern("P" + str(len(Patterns)), i)]
```

The dual variable values from the master problem are passed as an input into the sub problem which passes out the amended Patterns list of Pattern class objects.

```
# This loop will be repeated until morePatterns is set to False
while morePatterns == True:

    # Solve the problem as a Relaxed LP
    duals = masterSolve(Patterns, rollData)


    # Find another pattern
    Patterns, morePatterns = subSolve(Patterns, duals)
```

The problem is solved for a final time with all added patterns, and the solution and variable values are passed as output. The problem is solved as a non-relaxed Integer Problem since having any particular pattern cut a non integer number of times is impossible.

```
# Re-solve as an Integer Problem
solution, varsdict = masterSolve(Patterns, rollData, relax = False)
```

Displays the solution variables and objective value.

```
# Display Solution
for i,j in varsdict.items():
    print i, "=", j




print "objective = ", solution
```

The main file is available here.

Function File CG.py

The second file contains the Pattern class definition, and the functions of masterSolve & subSolve. Before these it is important to import the PuLP functions:

```
"""
Column Generation Functions



Authors: Antony Phillips,  Dr Stuart Mitchell  2008
"""


# Import PuLP modeler functions
from pulp import *
```

Class Definition

The Pattern class has 4 pattern constants (cost, trimValue, totalRollLength & lenOpts) defined inside it as class variables. There are 3 functions inside the Pattern class:
__init__: creates a Pattern object and assigns the **name** and **lengthsdict** attributes out of the inputs of the name and a list of the number of rolls of each length in that pattern.
__str__: returns the name of a pattern.
trim: returns the trim of a pattern, calculated using the class variables and attributes of that pattern.

```
class Pattern:
    """
    Information on a specific pattern in the SpongeRoll Problem
    """
    cost = 1
    trimValue = 0.04
    totalRollLength = 20
    lenOpts = ["5", "7", "9"]

    def __init__(self, name, lengths = None):
        self.name = name
        self.lengthsdict = dict(zip(self.lenOpts,lengths))


    def __str__(self):
        return self.name


    def trim(self):
        return Pattern.totalRollLength - sum([int(i)*int(self.lengthsdict[i]) for i in self.lengthsdict])
```

masterSolve Function

This function simply solves the Sponge Roll Problem LP and returns the relevant result depending on if this is the final call to this function (when the LP is not relaxed).

The only required inputs are the list of Pattern objects (Patterns), the rollData and whether or not the LP is relaxed. The rollData is split normally and prob is created.

```
def masterSolve(Patterns, rollData, relax = True):




    # The rollData is made into separate dictionaries
    (rollDemand,surplusPrice) = splitDict(rollData)


    # The variable prob is created
    prob = LpProblem("Cutting Stock Problem",LpMinimize)
```

Depending on the value of the input variable **relax**, the problem variables are made to be either LpInteger or LpContinuous.

```
    # vartype represents whether or not the variables are relaxed
    if relax:
        vartype = LpContinuous
    else:
        vartype = LpInteger


    # The problem variables are created
    pattVars = LpVariable.dicts("Pattern", Patterns, 0, None, vartype)
    surplusVars = LpVariable.dicts("Surplus", Pattern.lenOpts, 0, None, vartype)
```

The objective function and constraints are logically added to the prob variable. The class variables are used and the Pattern object list **Patterns** is used in several list comprehensions.

```
    # The objective function is entered: (the total number of large rolls used * the cost of each) -
    # (the value of the surplus stock) - (the value of the trim)
    prob += lpSum([pattVars[i]*Pattern.cost for i in Patterns]) - lpSum([surplusVars[i]*surplusPrice[i]\
            for i in Pattern.lenOpts]) - lpSum([pattVars[i]*i.trim()*Pattern.trimValue for i in Patterns])


    # The demand minimum constraint is entered
    for j in Pattern.lenOpts:
        prob += lpSum([pattVars[i]*i.lengthsdict[j] for i in Patterns]) - surplusVars[j]>=rollDemand[j],"Min%s"%j
```

The problem is solved using CPLEX and with no output messages. The solution variables are then rounded, otherwise values that were meant to be exact values may be slightly off due to floating point representation error.

```
    # The problem is solved using CPLEX with no output
    prob.solve(CPLEX(msg=0))



    # The variable values are rounded
    prob.roundSolution()
```

Since the state of relaxation of the LP determines what output is necessary, an *if* statement is used. When the LP is relaxed, it is not the final run through masterSolve and we are still looking for more patterns. Therefore the required output is the dictionary of dual variables values

```
    if relax:
            # Creates a dual variables list
            duals = {}
            for name,i in zip([Min5,Min7,'Min9'],Pattern.lenOpts):
                duals[i] = prob.constraints[name].pi

            return duals
```

Alternatively, if the problem is not relaxed then the function needs to return the value of the objective function, and the optimum variable values. The function also prints the number of rolls of each length in each pattern name, so the solution can be fully interpreted.

```
    else:
            # Creates a dictionary of the variables and their values
            varsdict = {}
            for v in prob.variables():
                varsdict[v.name] = v.varValue

            # The number of rolls of each length in each pattern is printed
            for i in Patterns:
                print i, " = %s"%[i.lengthsdict[j] for j in Pattern.lenOpts]


            return value(prob.objective), varsdict
```

subSolve Function

This function searches for another pattern to add to the master LP which would reduce the objective function. This is done by minimising the reduced cost of any potential new pattern. If the minimum value is less than zero, it is worth adding. The reduced cost of a pattern is calculated by the cost of that pattern minus the sum of the dual variables on each of the length constraints multiplied by the number of rolls of that length in the new pattern.

The inputs to the subSolve function are the list of Pattern objects, and the duals dictionary. The prob variable is created first.

```
def subSolve(Patterns, duals):

    # The variable prob is created
    prob = LpProblem("SubProb",LpMinimize)
```

A variable for the number of each length roll in the pattern is created, along with a variable for the trim length.

```
    # The problem variables are created
    vars = LpVariable.dicts("Roll Length", Pattern.lenOpts, 0, None, LpInteger)
    trim = LpVariable("Trim", 0 ,None,LpInteger)
```

The reduced cost of a new pattern is entered. Note that the cost of adding the pattern is it's cost ($1 in this case) minus the money gained back from sale of trim. This is because the amount of trim is a fixed attribute for any given pattern.

```
    # The objective function is entered: the reduced cost of a new pattern
    prob += (Pattern.cost - Pattern.trimValue*trim) - lpSum([vars[i]*duals[i] for i in Pattern.lenOpts]), "Objective"
```

The only constraint to this problem is that the total length of the trim and the smaller cut rolls must be equal to the length of the initial roll.

```
# The conservation of length constraint is entered
prob += lpSum([vars[i]*int(i) for i in Pattern.lenOpts]) + trim == Pattern.totalRollLength, "lengthEquate"
```

The problem is solved and the results are rounded.

```
# The problem is solved using CPLEX
prob.solve(CPLEX_DLL(msg=0))


# The variable values are rounded
prob.roundSolution()
```

The new pattern is written to a dictionary so that there is no problem with the values being out of order.

```
# The new pattern is written to a dictionary
varsdict = {}
newPattern = {}
for v in prob.variables():
    varsdict[v.name] = v.varValue
for i,j in zip(Pattern.lenOpts,["Roll_Length_5","Roll_Length_7","Roll_Length_9"]):
    newPattern[i] = int(varsdict[j])
```

If the value of the objective function is negative then this new pattern will reduce the value of master LP objective function and so it is added to the Patterns list, as a new instance of the Pattern class. Otherwise, morePatterns is set to False, so that the loop in the main function will end. Some values of prob.objective would be *very* small negatives that were meant to be zero, but took a non-zero value due to floating point representation error. To stop these values being interpreted as negatives (when they should be zero), value(prob.objective) is tested to be less than -10^-5.

```
# Check if there are more patterns which would reduce the master LP objective function further
if value(prob.objective) < -10**-5:
    morePatterns = True # continue adding patterns
    Patterns += [Pattern("P" + str(len(Patterns)), [newPattern[i] for i in ["5","7","9"]])]
else:
    morePatterns = False # all patterns have been added


return Patterns, morePatterns
```

The function file is available here.

 Previous: (5f) Sudoku As An LP

 Next: (5h) Column Generation 2