New Programming Interfaces for the AMPL Modeling Language

Robert Fourer, Filipe Brandão

AMPL Optimization Inc.

`{4er,fdabrandao}@ampl.com`

OR 2018: International Conference on Operations Research

Brussels — 12-14 September 2018 — **Session TA-5**

Algebraic Modeling Languages

New Programming Interfaces for the AMPL Modeling Language

Though fundamentally declarative in design, optimization modeling languages are invariably implemented within larger modeling systems that provide a variety of programming options. Although programming is not used to describe models, it facilitates the integration of models into broader algorithmic schemes and business applications. This presentation surveys ways in which a programming interface can be useful, with examples from the AMPL modeling language and system. The focus is on new APIs (application programming interfaces) for controlling AMPL from programs in Python and in R, and on new facilities for invoking Python programs from within AMPL.

3 Ways to Program in an Optimization Modeling System

Work inside the modeling system

- Write scripts using modeling language constructs

Call the modeling system from a programming language

- Make calls to the modeling system's APIs from general-purpose programming languages

Use a programming language inside the modeling system

- Embed programming language statements within modeling language scripts

Features

- Algebraic modeling language
- Built specially for optimization
- Designed to support many solvers

Programming alternatives

- *Scripting* based on modeling language extensions
- *APIs* for C++, C#, Java, MATLAB, Python, R
- *Embedded Python* (coming soon)

Application-building toolkits (not covered here)

- QuanDec / built on Java API
- Opalytics (Accenture) / connected via Python API

Outline

AMPL model

- Optimal roll cutting

AMPL script

- Trading off waste versus overruns

AMPL API programs

- Pattern enumeration in Python / R
- Pattern generation in Python

Embedded Python (a preview)

- Sending Python data to an AMPL model
- Executing Python sttements inside AMPL
- Handling callbcks

AMPL Model

Roll-cutting problem

- Fill orders for rolls of various widths
  - by cutting raw rolls of one (large) fixed width
  - using a variety of cutting patterns

Optimization model

- Decision variables
  - number of raw rolls to cut according to each pattern
- Objective
  - minimize number of raw rolls used
- Constraints
  - meet demands for each ordered width

AMPL Model

Mathematical Formulation

Given

w width of "raw" rolls

W set of (smaller) ordered widths

n number of cutting patterns considered

and

a_{ij} occurrences of width i in pattern j, for each $i \in W$ and $j = 1, \ldots, n$

b_i orders for width i, for each $i \in W$

o limit on overruns

AMPL Model

Mathematical Formulation *(cont'd)*

Determine

X_j number of rolls to cut using pattern j, for each $j = 1, \ldots, n$

to minimize

$\sum_{j=1}^{n} X_j$

total number of rolls cut

subject to

$b_i \leq \sum_{j=1}^{n} a_{ij} X_j \leq b_i + o$, for all $i \in W$

number of rolls of width i cut
must be at least the number ordered,
and must be within the overrun limit

AMPL Formulation

Symbolic model

```
param rawWidth;
set WIDTHS;

param nPatterns integer > 0;
set PATTERNS = 1..nPatterns;

param rolls {WIDTHS,PATTERNS} >= 0, default 0;

param order {WIDTHS} >= 0;
param overrun;

var Cut {PATTERNS} integer >= 0;

minimize TotalCut: sum {p in PATTERNS} Cut[p];

subject to OrderLimits {w in WIDTHS}:
  order[w] <= sum {p in PATTERNS} rolls[w,p] * Cut[p] <= order[w] + overrun;
```

$$b_i \leq \sum_{j=1}^{n} a_{ij} X_j \leq b_i + o$$

AMPL Formulation *(cont'd)*

Explicit data (independent of model)

```
param rawWidth := 64.5 ;

param: WIDTHS: order :=
         6.77    10
         7.56    40
        17.46    33
        18.76    10 ;

param nPatterns := 9 ;

param rolls:  1  2  3  4  5  6  7  8  9 :=
       6.77   0  1  1  0  3  2  0  1  4
       7.56   1  0  2  1  1  4  6  5  2
      17.46   0  1  0  2  1  0  1  1  1
      18.76   3  2  2  1  1  1  0  0  0 ;

param overrun := 6 ;
```

AMPL Command Language

Model + data = problem instance to be solved

```
ampl: model cut.mod;
ampl: data cut.dat;

ampl: option solver cplex;

ampl: solve;

CPLEX 12.8.0.0: optimal integer solution; objective 20
3 MIP simplex iterations
0 branch-and-bound nodes

ampl: option omit_zero_rows 1;
ampl: option display_1col 0;

ampl: display Cut;

4 13   8 5   9 2
```

Command Language *(cont'd)*

Solver choice independent of model and data

```
ampl: model cut.mod;
ampl: data cut.dat;

ampl: option solver gurobi;

ampl: solve;

Gurobi 8.0.0: optimal solution; objective 20
7 simplex iterations
1 branch-and-cut nodes

ampl: option omit_zero_rows 1;
ampl: option display_1col 0;

ampl: display Cut;

2 1    4 13    8 5    9 1
```

AMPL Model

Command Language *(cont'd)*

Results available for browsing

```
ampl: display {p in PATTERNS} sum {w in WIDTHS} w * rolls[w,p];

1 63.84   3 59.41   5 64.09   7 62.82   9 59.66   # material used
2 61.75   4 61.24   6 62.54   8 62.0              # in each pattern

ampl: display sum {p in PATTERNS}
ampl?   Cut[p] * (rawWidth - sum {w in WIDTHS} w * rolls[w,p]);

62.32                                             # total waste
                                                  # in solution

ampl: display OrderLimits.lslack;

 6.77  0                                          # overruns
 7.56  0                                          # of each pattern
17.46  0
18.76  5
```

AMPL Script

Trade off two objectives

- Minimize rolls cut
  - Fewer overruns, possibly more waste
- Minimize waste
  - Less waste, possibly more overruns

```
minimize TotalCut:
   sum {p in PATTERNS} Cut[p];

minimize TotalWaste:
   sum {p in PATTERNS}
      Cut[p] * (rawWidth - sum {w in WIDTHS} w * rolls[w,p]);
```

AMPL Script

Parametric Analysis of Tradeoff

Minimize rolls cut

- Set large overrun limit in data

Minimize waste

- Reduce overrun limit 1 roll at a time
- If there is a change in number of rolls cut
  - record total waste (increasing)
  - record total rolls cut (decreasing)
- Stop when no further progress possible
  - problem becomes infeasible *or*
  - total rolls cut falls to the minimum
- Report table of results

AMPL Script

Parametric Analysis *(cont'd)*

Script (setup and initial solve)

```
model cutTradeoff.mod;
data cutTradeoff.dat;

set OVER default {} ordered by reversed Integers;

param minCut;
param minCutWaste;
param minWaste {OVER};
param minWasteCut {OVER};

param prev_cut default Infinity;

option solver gurobi;
option solver_msg 0;

objective TotalCut;
solve >Nul;

let minCut := TotalCut;
let minCutWaste := TotalWaste;

objective TotalWaste;
```

AMPL Script

Parametric Analysis *(cont'd)*

Script (looping and reporting)

```
for {k in overrun .. 0 by -1} {
   let overrun := k;

   solve >Nul;

   if solve_result = 'infeasible' then break;

   if TotalCut < prev_cut then {
      let OVER := OVER union {k};
      let minWaste[k] := TotalWaste;
      let minWasteCut[k] := TotalCut;
      let prev_cut := TotalCut;
   }

   if TotalCut = minCut then break;
}

printf 'Min%3d rolls with waste%6.2f\n\n', minCut, minCutWaste;
printf ' Over  Waste  Cut\n';
printf {k in OVER}: '%4d%8.2f%5d\n', k, minWaste[k], minWasteCut[k];
```

AMPL Script

Parametric Analysis *(cont'd)*

Script run

```
ampl: include cutTradeoff.run

Min 20 rolls with waste 62.04

 Over  Waste  Number
  10   46.72    22
   7   47.89    21
   5   54.76    20

ampl:
```

AMPL API Program

Solve by pattern enumeration

- Set up a cutting-stock model
- Read data
  - demands, raw width
  - orders, overrun limit
- Compute data: all "good" patterns
  - extract widths from demand list
  - enumerate all non-dominated patterns
- Solve for the cutting plan

Hybrid approach

- Control & pattern enumeration in a programming language
- Model & modeling expressions in AMPL

Pattern Enumeration

Principles of APIs

- APIs for "all" popular languages
  - C++, C#, Java, MATLAB, Python, R
- Common overall design
- Common implementation core in C++
- Customizations for each language and its data structures

Key to examples: Python and R

- AMPL entities
- AMPL API Python/R objects
- AMPL API Python/R methods
- Python/R functions etc.

AMPL Model File

Same pattern-cutting model

```
param nPatterns integer > 0;

set PATTERNS = 1..nPatterns;  # patterns
set WIDTHS;                   # finished widths

param order {WIDTHS} >= 0;    # rolls of width j ordered
param overrun;                # permitted overrun on any width

param rawWidth;                    # width of raw rolls to be cut
param rolls {WIDTHS,PATTERNS} >= 0, default 0;
                                   # rolls of width i in pattern j

var Cut {PATTERNS} integer >= 0;   # raw rolls to cut in each pattern

minimize TotalRawRolls: sum {p in PATTERNS} Cut[p];

subject to FinishedRollLimits {w in WIDTHS}:
   order[w] <= sum {p in PATTERNS} rolls[w,p] * Cut[p] <= order[w] + overrun;
```

Some Python Data

A float, an integer, and a dictionary

```
roll_width = 64.5

overrun = 6

Orders = {
     6.77: 10,
     7.56: 40,
    17.46: 33,
    18.76: 10
}
```

. . . can also work with lists and Pandas dataframes

Some R Data

A float, an integer, and a dataframe

```
roll_width <- 64.5

overrun <- 6

orders <- data.frame(

  width  = c( 6.77, 7.56, 17.46, 18.76 ),
  demand = c(   10,   40,    33,    10 )

)
```

Pattern Enumeration in Python

Load & generate data, set up AMPL model

```
def cuttingEnum(dataset):
  from amplpy import AMPL

  # Read orders, roll_width, overrun

  exec(open(dataset+'.py').read(), globals())

  # Enumerate patterns

  widths = list(sorted(orders.keys(), reverse=True))
  patmat = patternEnum(roll_width, widths)

  # Set up model

  ampl = AMPL()
  ampl.option['ampl_include'] = 'models'
  ampl.read('cut.mod')
```

Pattern Enumeration in R

Load & generate data, set up AMPL model

```
cuttingEnum <- function(dataset) {
  library(rAMPL)

  # Read orders, roll_width, overrun

  source(paste(dataset, ".R", sep=""))

  # Enumerate patterns

  patmat <- patternEnum(roll_width, orders$width)
  cat(sprintf("\n%d patterns enumerated\n\n", ncol(patmat)))

  # Set up model

  ampl <- new(AMPL)
  ampl$setOption("ampl_include", "models")
  ampl$read("cut.mod")
```

Pattern Enumeration in Python

Send data to AMPL

```
# Send scalar values

ampl.param['nPatterns'] = len(patmat)
ampl.param['overrun'] = overrun
ampl.param['rawWidth'] = roll_width

# Send order vector

ampl.set['WIDTHS'] = widths
ampl.param['order'] = orders

# Send pattern matrix

ampl.param['rolls'] = {
   (widths[i], 1+p): patmat[p][i]
   for i in range(len(widths))
   for p in range(len(patmat))
}
```

Pattern Enumeration in R

Send data to AMPL

```
# Send scalar values

ampl$getParameter("nPatterns")$set(ncol(patmat))
ampl$getParameter("overrun")$set(overrun)
ampl$getParameter("rawWidth")$set(roll_width)

# Send order vector

ampl$getSet("WIDTHS")$setValues(orders$width)
ampl$getParameter("order")$setValues(orders$demand)

# Send pattern matrix

df <- as.data.frame(as.table(patmat))
df[,1] <- orders$width[df[,1]]
df[,2] <- as.numeric(df[,2])

ampl$getParameter("rolls")$setValues(df)
```

AMPL API

Pattern Enumeration in Python

Solve and get results

```
# Solve

ampl.option['solver'] = 'gurobi'
ampl.solve()

# Retrieve solution

CuttingPlan = ampl.var['Cut'].getValues()
cutvec = list(CuttingPlan.getColumn('Cut.val'))
```

Pattern Enumeration in R

Solve and get results

```
# Solve

ampl$setOption("solver", "gurobi")
ampl$solve()

# Retrieve solution

CuttingPlan <- ampl$getVariable("Cut")$getValues()
solution <- CuttingPlan[CuttingPlan[,-1] != 0,]
```

Pattern Enumeration in Python

Display solution

```
# Prepare solution data

summary = {
    'Data': dataset,
    'Obj': int(ampl.obj['TotalRawRolls'].value()),
    'Waste': ampl.getValue(
                 'sum {p in PATTERNS} Cut[p] * \
                    (rawWidth - sum {w in WIDTHS} w*rolls[w,p])'
             )
}

solution = [
    (patmat[p], cutvec[p])
    for p in range(len(patmat))
    if cutvec[p] > 0
]


# Create plot of solution

cuttingPlot(roll_width, widths, summary, solution)
```

Pattern Enumeration in R

Display solution

```
  # Prepare solution data
  data <- dataset
  obj <- ampl$getObjective("TotalRawRolls")$value()
  waste <- ampl$getValue(
    "sum {p in PATTERNS} Cut[p] * (rawWidth - sum {w in WIDTHS} w*rolls[w,p])"
  )
  summary <- list(data=dataset, obj=obj, waste=waste)

  # Create plot of solution
  cuttingPlot(roll_width, orders$width, patmat, summary, solution)
}
```

AMPL API

Pattern Enumeration in Python

Enumeration routine

```
def patternEnum(roll_width, widths, prefix=[]):
  from math import floor
  max_rep = int(floor(roll_width/widths[0]))
  if len(widths) == 1:
      patmat = [prefix+[max_rep]]
  else:
     patmat = []
     for n in reversed(range(max_rep+1)):
        patmat += patternEnum(roll_width-n*widths[0], widths[1:], prefix+[n])
  return patmat
```

Pattern Enumeration in R

Enumeration routine

```
patternEnum <- function(roll_width, widths, prefix=c()) {
  cur_width <- widths[length(prefix)+1]
  max_rep <- floor(roll_width/cur_width)
  if (length(prefix)+1 == length(widths)) {
      return (c(prefix, max_rep))
  } else  {
      patterns <- matrix(nrow=length(widths), ncol=0)
      for (n in 0:max_rep) {
          patterns <- cbind(
              patterns,
              patternEnum(roll_width-n*cur_width, widths, c(prefix, n))
          )
      }
      return (patterns)
  }
}
```

AMPL API

Pattern Enumeration in Python

Plotting routine

```
def cuttingPlot(roll_width, widths, summ, solution):
    import numpy as np
    import matplotlib.pyplot as plt

    ind = np.arange(len(solution))
    acc = [0]*len(solution)

    colorlist = ['red','lightblue','orange','lightgreen',
                 'brown','fuchsia','silver','goldenrod']
```

Pattern Enumeration in R

Plotting routine

```
cuttingPlot <- function(roll_width, widths, patmat, summary, solution) {

  pal <- rainbow(length(widths))
  par(mar=c(1,1,1,1))
  par(mfrow=c(1,nrow(solution)))

  for(i in 1:nrow(solution)) {
    pattern <- patmat[, solution[i, 1]]
    data <- c()
    color <- c()}
```

Pattern Enumeration in Python

Plotting routine (cont'd)

```
for p, (patt, rep) in enumerate(solution):
   for i in range(len(widths)):
      for j in range(patt[i]):
         vec = [0]*len(solution)
         vec[p] = widths[i]
         plt.barh(ind, vec, 0.6, acc,
                    color=colorlist[i%len(colorlist)], edgecolor='black')
         acc[p] += widths[i]

plt.title(summ['Data'] + ": " +
   str(summ['Obj']) + " rolls" + ", " +
   str(round(100*summ['Waste']/(roll_width*summ['Obj']),2)) + "% waste"
   )

plt.xlim(0, roll_width)
plt.xticks(np.arange(0, roll_width, 10))
plt.yticks(ind, tuple("x {:}".format(rep) for patt, rep in solution))

plt.show()
```

AMPL API

Pattern Enumeration in R

Plotting routine (cont'd)

```
  for(j in 1:length(pattern)) {
      if(pattern[j] >= 1) {
        for(k in 1:pattern[j]) {
          data <- rbind(data, widths[j])
          color <- c(color, pal[j])
        }
      }
    }

    label <- sprintf("x %d", solution[i, -1])

    barplot(data, main=label, col=color,
            border="white", space=0.04, axes=FALSE, ylim=c(0, roll_width))

  }

  print(summary)

}
```

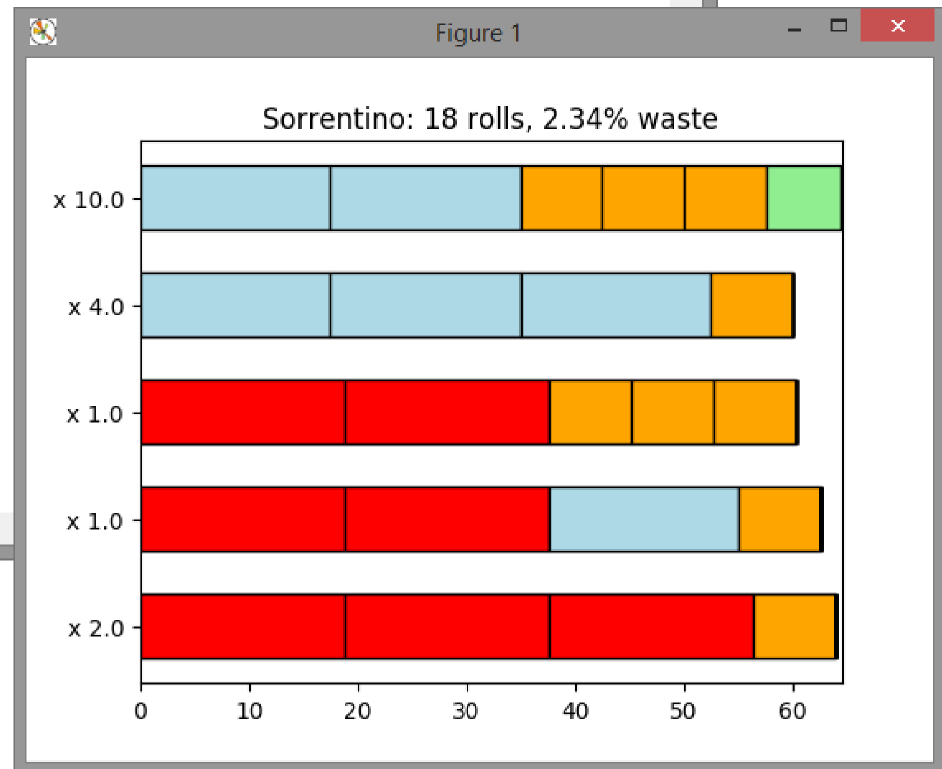
AMPL API

Pattern Enumeration in Python

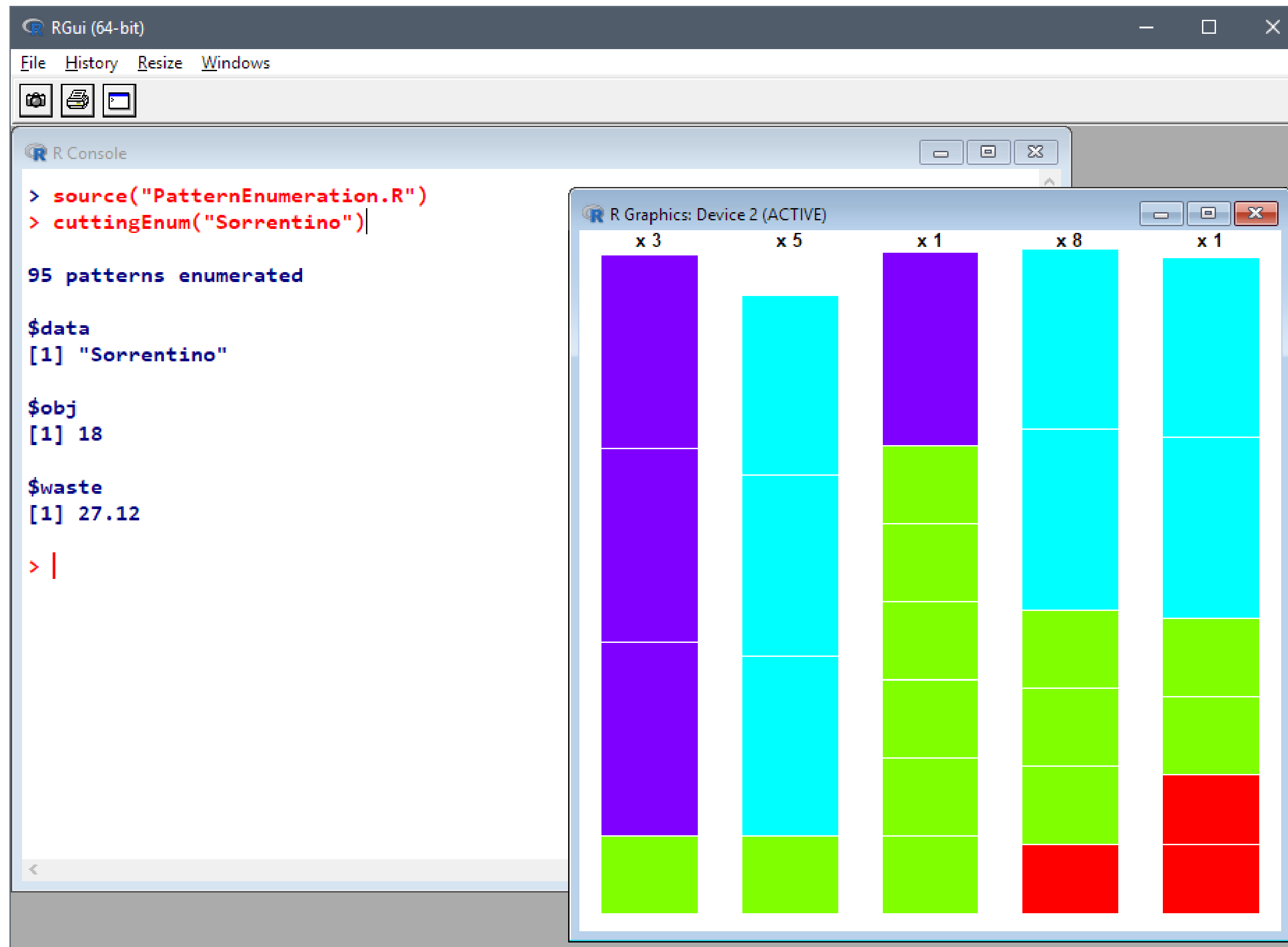
sw: running ipython

File Edit Help

```
sw: ipython

Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  6 2014, 22:16:31) [MSC v.1600 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from pattern_enumeration import *

In [2]: cuttingEnum('Sorrentino')

Gurobi 7.5.0: optimal solution; objective 18
9 simplex iterations
1 branch-and-cut nodes
```



Pattern Enumeration in R



AMPL API Program

Solve by pattern generation

- Solve continuous relaxation using subset of “easy” patterns
- Add “most promising” pattern to the subset
  - Minimize reduced cost given dual values
  - Equivalent to a one-constraint (“knapsack”) problem
- Iterate as long as there are promising patterns
  - Stop when minimum reduced cost is zero
- Solve IP using all patterns found

Two AMPL objects

- `Master` is the cutting model with current pattern subset
- `Sub` is the one-constraint knapsack problem

AMPL API

Pattern Generation in Python

Get data, set up master problem

```
function cuttingGen(dataset)
  from amplpy import AMPL

  # Read orders, roll_width, overrun; extract widths

  exec(open(dataset+'.py').read(), globals())
  widths = list(sorted(orders.keys(), reverse=True))

  # Set up cutting (master problem) model

  Master = AMPL()
  Master.option['ampl_include'] = 'models'
  Master.read('cut.mod')

  # Define a param for sending new patterns

  Master.eval('param newPat {WIDTHS} integer >= 0;')

  # Set solve options

  Master.option['solver'] = 'gurobi'
  Master.option['relax_integrality'] =  1
```

AMPL API

Pattern Generation in Python

Send data to master problem

```
# Send scalar values

Master.param['nPatterns'] = len(widths)
Master.param['overrun'] = overrun
Master.param['rawWidth'] = roll_width

# Send order vector

Master.set['WIDTHS'] = widths
Master.param['order'] = orders

# Generate and send initial pattern matrix

Master.param['rolls'] = {
   (widths[i], 1+i): int(floor(roll_width/widths[i]))
      for i in range(len(widths))
  }
```

Pattern Generation in Python

Set up subproblem

```
# Define knapsack subproblem

Sub = AMPL()
Sub.option['solver'] = 'gurobi'

Sub.eval('''
   set SIZES;
   param cap >= 0;
   param val {SIZES};
   var Qty {SIZES} integer >= 0;
   maximize TotVal: sum {s in SIZES} val[s] * Qty[s];
   subject to Cap: sum {s in SIZES} s * Qty[s] <= cap;
''')

# Send subproblem data

Sub.set['SIZES'] = widths
Sub.param['cap'] = roll_width
```

Pattern Generation in Python

Generate patterns and re-solve cutting problems

```
# Alternate between master and sub solves

while True:
    Master.solve()

    Sub.param['val'].setValues(Master.con['OrderLimits'].getValues())
    Sub.solve()
    if Sub.obj['TotVal'].value() <= 1.00001:
       break

    Master.param['newPat'].setValues(Sub.var['Qty'].getValues())
    Master.eval('let nPatterns := nPatterns + 1;')
    Master.eval('let {w in WIDTHS} rolls[w, nPatterns] := newPat[w];')

# Compute integer solution

Master.option['relax_integrality'] = 0
Master.solve()
```

Pattern Generation in Python

Display solution

```
# Retrieve solution

cutting_plan = Master.var['Cut'].getValues()
cutvec = list(cutting_plan.getColumn('Cut.val'))

# Prepare summary data

summary = {
      'Data': dataset,
      'Obj': int(Master.obj['TotalRawRolls'].value()),
      'Waste': Master.getValue(
               'sum {p in PATTERNS} Cut[p] * \
                  (rawWidth - sum {w in WIDTHS} w*rolls[w,p])'
               )
}

# Retrieve patterns and solution

npatterns = int(Master.param['nPatterns'].value())
rolls = Master.param['rolls'].getValues().toDict()
cutvec = Master.var['Cut'].getValues().toDict()
```

Pattern Generation in Python

Display solution

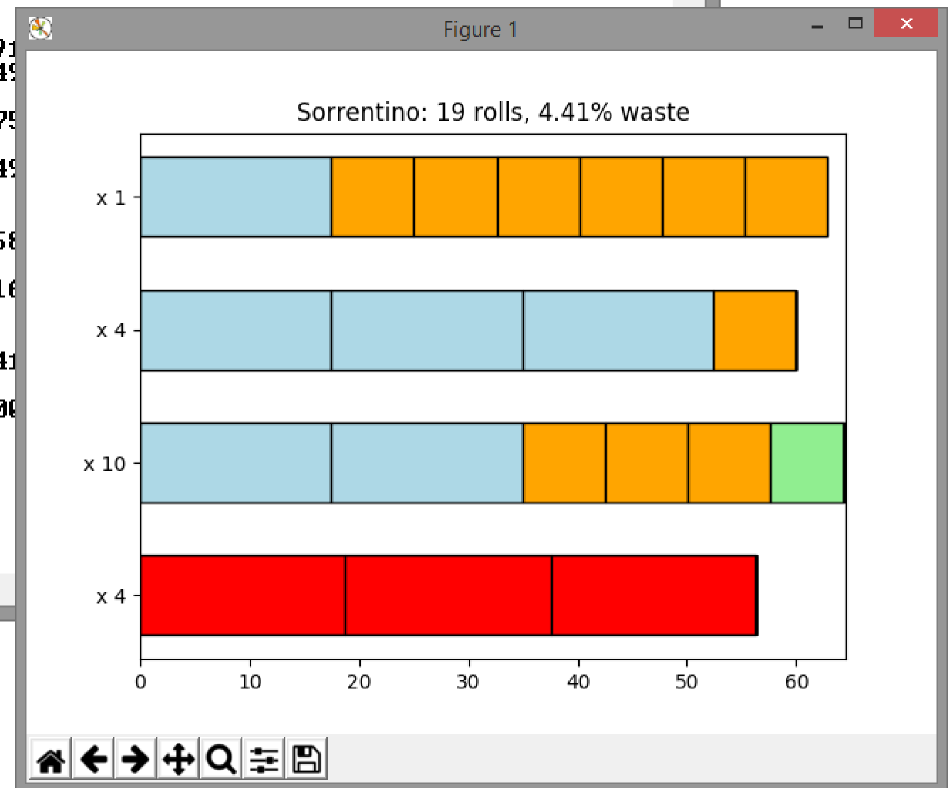
```
# Prepare solution data

solution = [
   ([int(rolls[widths[i], p+1][0])
      for i in range(len(widths))], int(cutvec[p+1][0]))
   for p in range(npatterns)
   if cutvec[p+1][0] > 0
]

# Create plot of solution

cuttingPlot(roll_width, widths, summary, solution)
```

Pattern Generation in Python

sw: running ipython

File Edit Help

```
sw: ipython --no-banner

In [1]: from pattern_generation import *

In [2]: cuttingGen('Sorrentino')
Gurobi 7.5.0: optimal solution; objective 20.44444444
Gurobi 7.5.0: optimal solution; objective 1.152777
2 simplex iterations
1 branch-and-cut nodes
Gurobi 7.5.0: optimal solution; objective 18.791
Gurobi 7.5.0: optimal solution; objective 1.1249
1 simplex iterations
Gurobi 7.5.0: optimal solution; objective 18.375
3 simplex iterations
Gurobi 7.5.0: optimal solution; objective 1.1249
1 simplex iterations
1 branch-and-cut nodes
Gurobi 7.5.0: optimal solution; objective 17.958
5 simplex iterations
Gurobi 7.5.0: optimal solution; objective 1.0416
5 simplex iterations
1 branch-and-cut nodes
Gurobi 7.5.0: optimal solution; objective 17.941
5 simplex iterations
Gurobi 7.5.0: optimal solution; objective 1.0000
1 simplex iterations
1 branch-and-cut nodes
Gurobi 7.5.0: optimal solution; objective 19
3 simplex iterations
1 branch-and-cut nodes
```



Embedded Python *(a preview)*

Sending Python data to an AMPL model

- via AMPL API for Python
- via Python references in the AMPL model

Executing Python statements inside AMPL

Handling callbacks

- Write callback function in Python
- Export problem + callback, solve, import results

Embedded Python

AMPL Model

Symbolic sets, parameters, variables, objective, constraints

```
# DATA

set FOOD ;
set NUTR ;

param cost { FOOD } > 0;

param f_min { FOOD } >= 0;
param f_max {j in FOOD } >= f_min [j];

param n_min { NUTR } >= 0;
param n_max {i in NUTR } >= n_min [i];

param amt {NUTR , FOOD } >= 0;

# MODEL

var Buy {j in FOOD } >= f_min [j], <= f_max [j];

minimize total_cost:
    sum {j in FOOD } cost [j] * Buy[j];

subject to diet {i in NUTR }:
    n_min [i] <= sum {j in FOOD } amt[i,j] * Buy[j] <= n_max [i];
```

Embedded Python

Python Data

Lists, dictionaries

```
food = ['BEEF', 'CHK', 'FISH', 'HAM', 'MCH', 'MTL', 'SPG', 'TUR']

cost = {
    'HAM': 2.89, 'BEEF': 3.59, 'MCH': 1.89, 'FISH': 2.29,
    'CHK': 2.59, 'MTL': 1.99, 'TUR': 2.49, 'SPG': 1.99
}

.......

amt = [
    [ 60,    8,   8,  40,   15,  70,   25,   60],
    [ 20,    0,  10,  40,   35,  30,   50,   20],
    [ 10,   20,  15,  35,   15,  15,   25,   15],
    [ 15,   20,  10,  10,   15,  15,   15,   10],
    [928, 2180, 945, 278, 1182, 896, 1329, 1397],
    [295,  770, 440, 430,  315, 400,  379,  450]
]
```

Embedded Python

Sending Data to AMPL (API)

Call `ampl` methods to read model, send data

```
from amplpy import AMPL
ampl = AMPL()
ampl.read('diet.mod')

ampl.set['FOOD'] = food

ampl.param['cost'] = cost
ampl.param['f_min'] = f_min
ampl.param['f_max'] = f_max

ampl.set['NUTR'] = nutr

ampl.param['n_min'] = n_min
ampl.param['n_max'] = n_max

ampl.param['amt'] = {
    (n, f): amt[i][j]
    for i, n in enumerate(nutr)
    for j, f in enumerate(food)
}

ampl.solve()
```

Sending Data to AMPL (Embedded)

Move data correspondences into the model

```
# SYMBOLIC DATA WITH PYTHON LINKS

$SET[FOOD]{ food };

$PARAM[cost{^FOOD}]{ cost };
$PARAM[f_min{^FOOD}]{ f_min };
$PARAM[f_max{^FOOD}]{ f_max };

$SET[NUTR]{ nutr };

$PARAM[n_min{^NUTR}]{ n_min };
$PARAM[n_max{^NUTR}]{ n_max };

$PARAM[amt]{{
      (n, f): amt[i][j]
      for i, n in enumerate(nutr)
      for j, f in enumerate(food)
    }};

# MODEL

var Buy {j in FOOD } >= f_min [j], <= f_max [j];

.......
```

Sending Data to AMPL (Embedded)

Move data correspondences into the model

```
from amplpy import AMPL
from pympl import PyMPL

ampl = AMPL(langext=PyMPL())
ampl.read('dietpy.mod’)

ampl.solve()
```

Executing Python inside AMPL

Fix AMPL variables according to Python variable

```
var r{1..NT}, ${">= 0" if BACKLOG else ">= 0, <= 0"}$;
                                      # use these variables iff BACKLOG > 0
```

Callback Functions

Export problem, solve, import solution

```
def callback(model, where):
    """Gurobi callback function."""

    if where == gpy.GRB.Callback.MIPSOL:

        nodecnt = model.cbGet(gpy.GRB.Callback.MIPSOL_NODCNT)
        obj = model.cbGet(gpy.GRB.Callback.MIPSOL_OBJ)
        solcnt = model.cbGet(gpy.GRB.Callback.MIPSOL_SOLCNT)
        print(
            '**** New soln at node {:.0f}, obj {:g}, sol {:d} ****'.format(
                nodecnt, obj, solcnt
            )
        )

grb_model = ampl.exportGurobiModel()
grb_model.optimize(callback)

ampl.importGurobiSolution(grb_model)
ampl.display('{i in OBJECTS: x[i] != 0} x[i]')
```