New Programming Tools and Interfaces for Deploying AMPL Models

Robert Fourer, Filipe Brandão

AMPL Optimization Inc.

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

INFORMS Annual Meeting

Phoenix — 4-7 November 2018 — **Session MC70**

New Advanced Deployment Features of Modeling Languages

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

- Use modeling system APIs
  created for various general-purpose programming languages

Mix modeling and programming constructs

- Embed programming language statements
  within model definitions and scripts
- Write programs in general-purpose languages
  that modify or extend model definitions

AMPL

Features

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

Programming options

- *Scripting* based on modeling language extensions
- *APIs* for C++, C#, Java, MATLAB, Python, R
- *Embedded Python* processed by the Python API
  - (available 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)

- Specifying Python data in an AMPL model
- Executing Python sttements inside AMPL
- Handling callbacks

AMPL Command Environment

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
  - don't exceed demands too much

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

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
```

AMPL Model

Command Language *(cont'd)*

Solver choice independent of model and data

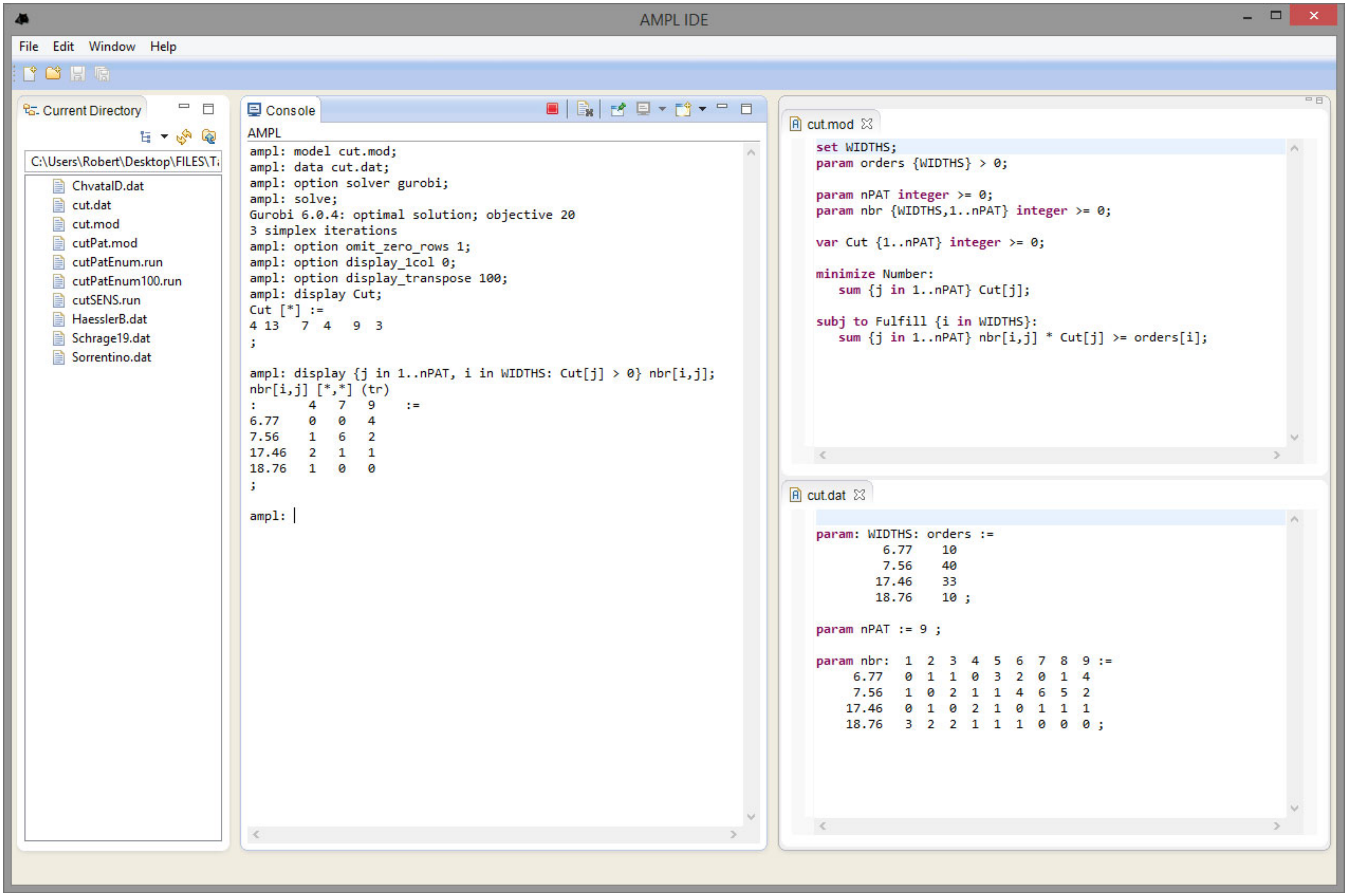
```
ampl: model cut.mod;
ampl: data cut.dat;

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
```

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
```

IDE for Command Language



AMPL Script

Trade off two objectives

- Minimize rolls cut
  - Fewer rolls, fewer overruns but less efficient patterns
- Minimize waste
  - More efficient patterns but more rolls, 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

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 patterns having waste < smallest width
- Solve for the cutting plan

Hybrid approach

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

Preface

AMPL 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;
```

AMPL API

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)
```

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)
}
```

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
```

AMPL API

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)
  }
}
```

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()
```

Pattern Enumeration in R

Plotting routine (cont'd)

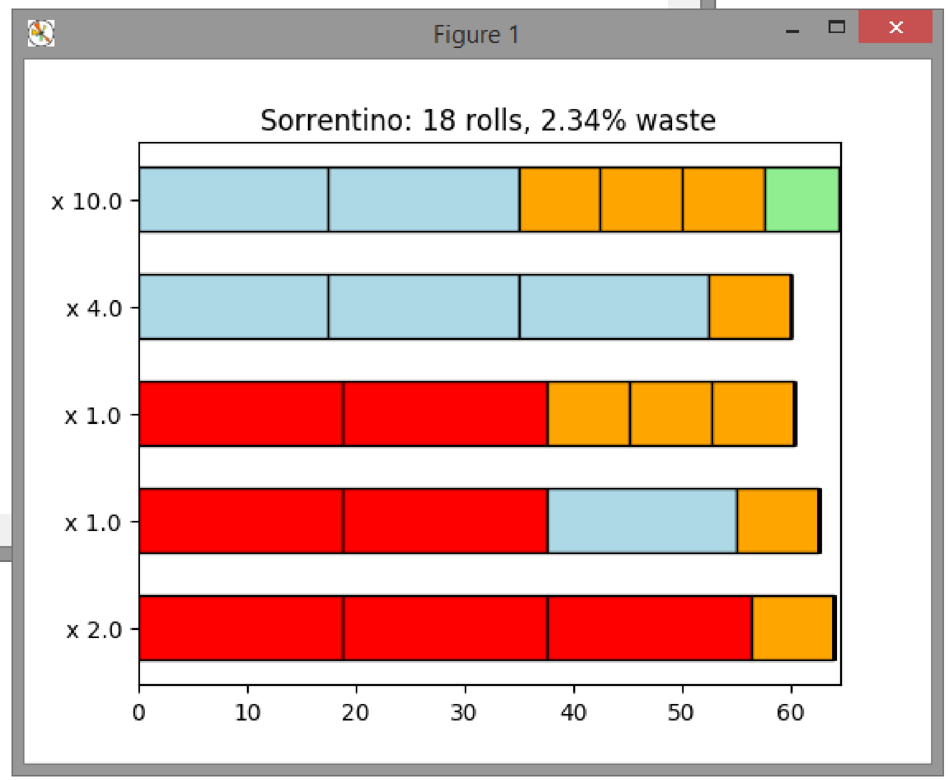
```
  for(j in 1:length(pattern)) {
      if(pattern[j] >= 1) {
        for(k in 1:pattern[j]) {
          data <- rbind(data, widths[j])
          color <- c(color, pal[j])
        }
      }
    }

    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)

}
```

Pattern Enumeration in Python

sw: running ipython

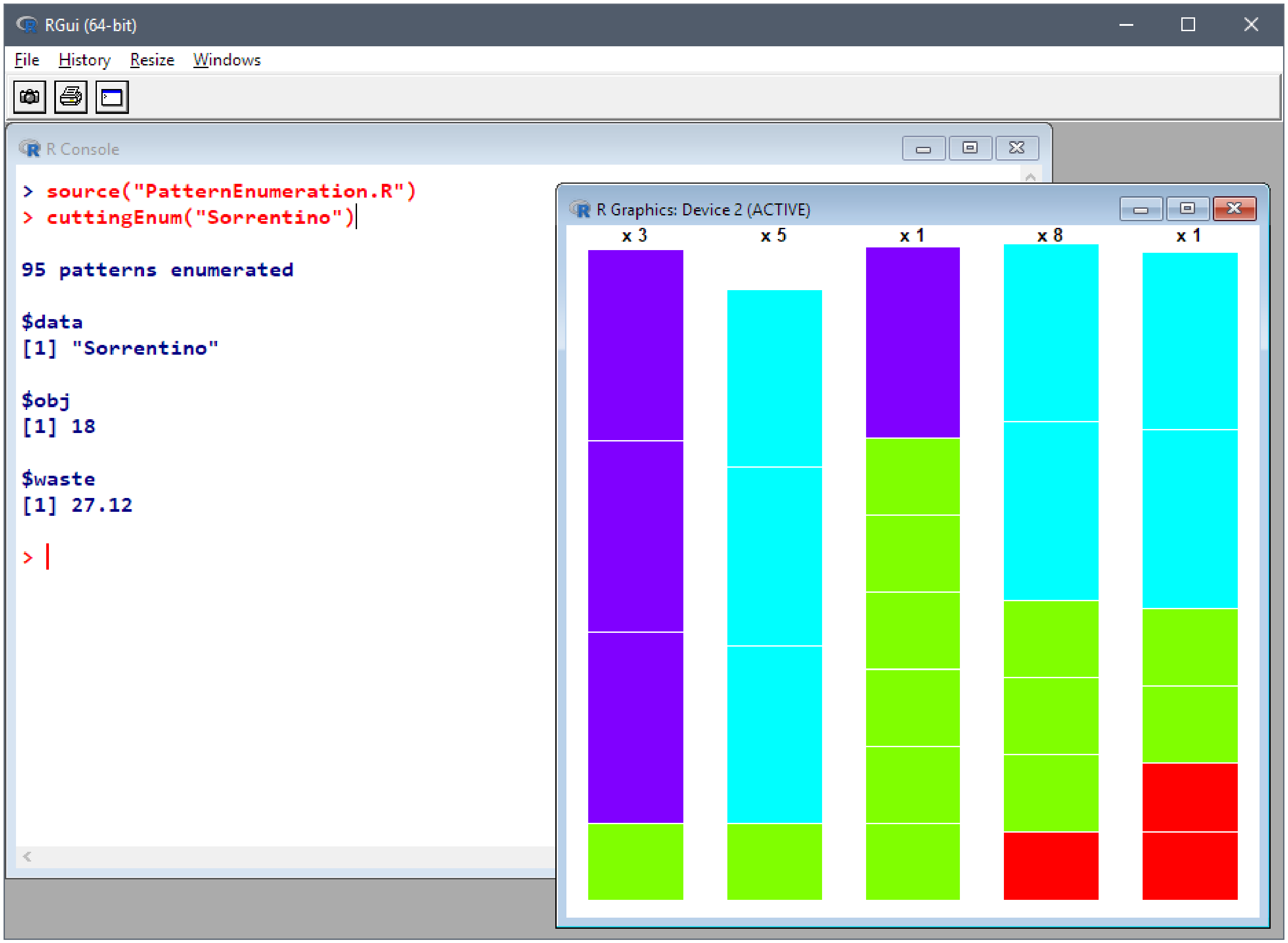
File Edit Help

```
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
```



AMPL API

Pattern Enumeration in R



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 scripts

- Generate specialized constraints for lot sizing

Handling callbacks

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

Embedded Python

AMPL Model

Symbolic sets, parameters, variables, objective, constraints

diet.mod

```
# 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];
```

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()
```

Embedded Python

Sending Data to AMPL (Embedded)

Move data correspondences into the model

dietpy.mod

```
# 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)

Process with PyMPL language extension

```
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

lotsize.mod

```
$PARAM[NT]{8};

var x {1..NT}, >= 0;   # production lot size
var y {1..NT}, binary; # production set-up
var s {0..NT}, >= 0;   # inventory level

var r {1..NT}, ${">= 0" if BACKLOG else ">= 0, <= 0"}$;

                                        # use these variables iff BACKLOG > 0
```

Embedded Python

Executing Python inside AMPL

Invoke Python generators for special lot-sizing constraints

lotsize.mod

```
$EXEC{
def mrange(a, b):
    return range(a, b+1)
s = ['s[{}]'.format(t) for t in mrange(0, NT)]
y = ['y[{}]'.format(t) for t in mrange(1, NT)]
d = [demand[t] for t in mrange(1, NT)]
if BACKLOG is False:
    WW_U_AMPL(s, y, d, NT, prefix='w')
else:
    r = ['r[{}]'.format(t) for t in mrange(1, NT)]
    WW_U_B_AMPL(s, r, y, d, NT, prefix='w')
};
```

```
ampl = AMPL(langext=PyMPL())
ampl.read('lotsize.mod')
ampl.solve()
```

Executing Python inside AMPL

Optional listing of generated constraints

```
var ws {wi in 0..8} = s[wi];
var wr {wi in 1..8} = r[wi];
var wy {wi in 1..8} = y[wi];

param wD {1..8, 1..8};

data;

param wD :=
[1,1]400 [1,2]800 [1,3]1600 [1,4]2400 [1,5]3600 [1,6]4800 [1,7]6000 [1,8]7200
[2,1]0   [2,2]400 [2,3]1200 [2,4]2000 [2,5]3200 [2,6]4400 [2,7]5600 [2,8]6800
[3,1]0   [3,2]0   [3,3]800  [3,4]1600 [3,5]2800 [3,6]4000 [3,7]5200 [3,8]6400
[4,1]0   [4,2]0   [4,3]0    [4,4]800  [4,5]2000 [4,6]3200 [4,7]4400 [4,8]5600
[5,1]0   [5,2]0   [5,3]0    [5,4]0    [5,5]1200 [5,6]2400 [5,7]3600 [5,8]4800
[6,1]0   [6,2]0   [6,3]0    [6,4]0    [6,5]0    [6,6]1200 [6,7]2400 [6,8]3600
[7,1]0   [7,2]0   [7,3]0    [7,4]0    [7,5]0    [7,6]0    [7,7]1200 [7,8]2400
[8,1]0   [8,2]0   [8,3]0    [8,4]0    [8,5]0    [8,6]0    [8,7]0    [8,8]1200
;

model;
```

Executing Python inside AMPL

Optional listing of generated constraints (cont'd)

```
var wa {1..8};
var wb {1..8};

subject to wXY {wt in 1..8}: wa[wt] + wb[wt] + wy[wt] >= 1;

subject to wXA {wk in 1..8, wt in wk..min(8, wk+8-1): wD[wt,wt]>0}:

    ws[wk-1] >=
        sum {wi in wk..wt} wD[wi,wi] * wa[wi]
        - sum {wi in wk..wt-1} wD[wi+1,wt] * wy[wi];

subject to wXB {wk in 1..8, wt in max(1, wk-8+1)..wk: wD[wt,wt]>0}:

    wr[wk] >=
        sum {wi in wt..wk} wD[wi,wi] * wb[wi]
        - sum {wi in wt+1..wk} wD[wt,wi-1] * wy[wi];
```

Embedded Python

Callbacks

AMPL model with embedded Python

```
$SET[OBJECTS]{list(range(n))};
$SET[RESOURCES]{list(range(m))};

$PARAM[value]{value, i0=0};

$PARAM[weight]{{
    (i, j): weight[i][j]
    for i in range(n)
    for j in range(m)
}};

$PARAM[capacity]{capacity, i0=0};

var x {OBJECTS} >= 0 <= 1 integer;

subject to Limits {r in RESOURCES}:
  sum {i in OBJECTS} weight[i, r] * x[i] <= capacity[r];

maximize Profit:
  sum {i in OBJECTS} value[i] * x[i];
```

Embedded Python

Callbacks

Callback function

```
def callback(model, where):

    global solinfo

    if where == gpy.GRB.Callback.MIPSOL:  # new MIP solution found

        nodecnt = model.cbGet(gpy.GRB.Callback.MIPSOL_NODCNT)
        obj = model.cbGet(gpy.GRB.Callback.MIPSOL_OBJ)
        solinfo.append((nodecnt, obj))    # append to solution list

        solcnt = model.cbGet(gpy.GRB.Callback.MIPSOL_SOLCNT)

        print(
            '** New solution at node {:.0f}, obj {:g}, sol {:d} **'.format(
                nodecnt, obj, solcnt
            ),
            file=log  # write to log.txt
        )

        if time()-t0 >= 10 and solcnt >= 2:
            model.terminate()             # stop solution process and return
```

Callbacks

AMPL Python API: Export problem, solve, import solution

```
from pympl import PyMPL
from amplpy import AMPL
import gurobipy as gpy

ampl = AMPL(langext=PyMPL())
ampl.read('multiknapsack.mod')

grb_model = ampl.exportGurobiModel()
grb_model.params.threads = 1
grb_model.params.timelimit = 10
t0 = time()
solinfo = []  # list to store objective values and node counts
log = open('log.txt', 'w')
grb_model.optimize(callback)

ampl.importGurobiSolution(grb_model)
ampl.display('{i in OBJECTS: x[i] != 0} x[i]')
print(solinfo)  # print stored objective values and node counts
```