Building AMPL Models into Your Applications

Robert Fourer

4er@ampl.com

AMPL Optimization Inc.
www.ampl.com — +1 773-336-AMPL

27th European Conference on Operational Research

Glasgow, Scotland, U.K. — 12-15 July 2015
Session MA-18, *Software for Optimization Modeling 1*

Building AMPL Models into Applications

Algebraic modeling languages were developed with the goal of making optimization models much easier to develop, debug, and maintain. However it is not necessary to give up these advantages when embedding a model into a larger system or deploying it to users. Two distinct facilities of modeling languages are commonly used to integrate models into applications:

- Scripting brings the programmer to the modeling language, extending the language so that the same constructs convenient for describing a model can also be used to specify how the model will be used in a broader context.
- APIs bring the modeling language to the programmer, providing access to model objects and methods for applications written in general-purpose programming languages.

The strengths of these two approaches are contrasted by comparing implementations of column-generation schemes for cutting problems.

Outline

Simple roll cutting example

- Solution via command language
- Sensitivity analysis via scripting

Roll cutting by pattern enumeration

- via scripting
- via MATLAB API
- via Java API

Roll cutting by pattern generation

- via scripting
- via MATLAB API

Roll Cutting Problem

Motivation

- 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

Roll cutting

Mathematical Formulation

Given

W set of ordered widths

n number of 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 $i \in W$

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

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

number of rolls of width i cut
must be at least the number ordered

Roll Cutting

AMPL Formulation

Symbolic model

```
set WIDTHS;
param orders {WIDTHS} > 0;

param nPAT integer >= 0;
param nbr {WIDTHS,1..nPAT} integer >= 0;

var Cut {1..nPAT} integer >= 0;

minimize Number:
   sum {j in 1..nPAT} Cut[j];

subj to Fulfill {i in WIDTHS}:
   sum {j in 1..nPAT} nbr[i,j] * Cut[j] >= orders[i];
```

$$\sum_{j=1}^{n} a_{ij} X_j \geq b_i$$

AMPL Formulation *(cont'd)*

Explicit data (independent of model)

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

param nPAT := 9 ;

param nbr:  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 ;
```

Command Language

Model + data = problem instance to be solved

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

ampl: option solver gurobi;

ampl: solve;

Gurobi 6.0.4: optimal solution; objective 20
3 simplex iterations

ampl: display Cut;

4 13   7 4   9 3

ampl: display {j in 1..nPAT, i in WIDTHS: Cut[j] > 0} nbr[i,j];

:       4   7   9  :=
6.77    0   0   4
7.56    1   6   2
17.46   2   1   1
18.76   1   0   0
```

Command Language *(cont'd)*

Solver choice independent of model and data

```
ampl: model cut.mod;
ampl: data cut.dat;
ampl: option solver cplex;
ampl: solve;
CPLEX 12.6.2.0: optimal integer solution; objective 20
3 MIP simplex iterations
ampl: display Cut;
4 13   7 4   9 3
ampl: display {j in 1..nPAT, i in WIDTHS: Cut[j] > 0} nbr[i,j];
:       4   7   9  :=
6.77    0   0   4
7.56    1   6   2
17.46   2   1   1
18.76   1   0   0
```

Scripting

Bring the programmer to the modeling language

Extend modeling language syntax . . .

- Algebraic expressions
- Set indexing expressions
- Interactive commands

. . . with programming concepts

- Loops of various kinds
- If-then and If-then-else conditionals
- Assignments

Parametric Analysis

Increase order levels in 1% steps

- Increase `orders[i]` for all `i in WIDTHS`
- Record results
  - increments at which objective value increases
  - corresponding numbers of rolls cut

. . . display results at the end

Parametric Analysis *(cont'd)*

Script (setup)

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

param prevNumber default
   (sum {i in WIDTHS} i * orders[i]) /
   (max {j in 1..nPAT} sum {i in WIDTHS} i * nbr[i,j]);

param baseOrders {WIDTHS};
let {i in WIDTHS} baseOrders[i] := orders[i];

set INCR default {};   # increments at which the objective changes
param incrObj {INCR};  # corresponding objective values

option solver Gurobi;
option solver_msg 0;
```

Parametric Analysis *(cont'd)*

Script (looping and reporting)

```
for {frac in 1 .. 1.25 by 0.01} {
   let {i in WIDTHS} orders[i] := frac * baseOrders[i];

   solve >Nul;

   if Number > prevNumber then {
      let INCR := INCR union {frac};
      let incrObj[frac] := Number;
      let prevNumber := Number;
   }
}

printf ' Step  Number\n';
printf {frac in INCR}: '%5.2f%5d\n', frac, incrObj[frac];
```

Parametric Analysis *(cont'd)*

Run

```
ampl: include cutSens.run;
 Step  Number
 1.00   20
 1.01   21
 1.07   22
 1.13   23
 1.16   24
 1.22   25
ampl:
```

Cutting *via* Pattern Enumeration

Build the pattern list, then solve

- Read general model
- Read data: demands, raw width
- Compute data: all usable patterns
- Solve problem instance

Scripting

Pattern Enumeration

Model

```
param roll_width > 0;

set WIDTHS ordered by reversed Reals;
param orders {WIDTHS} > 0;

param maxPAT integer >= 0;
param nPAT integer >= 0, <= maxPAT;

param nbr {WIDTHS,1..maxPAT} integer >= 0;

var Cut {1..nPAT} integer >= 0;

minimize Number:
   sum {j in 1..nPAT} Cut[j];

subj to Fulfill {i in WIDTHS}:
   sum {j in 1..nPAT} nbr[i,j] * Cut[j] >= orders[i];
```

Scripting

Pattern Enumeration

Data

```
param roll_width := 64.50 ;

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

Pattern Enumeration

Script (initialize)

```
model cutPAT.mod;
data Sorrentino.dat;

param curr_sum >= 0;
param curr_width > 0;
param pattern {WIDTHS} integer >= 0;

let maxPAT := 1000000;

let nPAT := 0;
let curr_sum := 0;
let curr_width := first(WIDTHS);
let {w in WIDTHS} pattern[w] := 0;
```

Scripting

Pattern Enumeration

Script (loop)

```
repeat {
   if curr_sum + curr_width <= roll_width then {
      let pattern[curr_width] := floor((roll_width-curr_sum)/curr_width);
      let curr_sum := curr_sum + pattern[curr_width] * curr_width;
      }
   if curr_width != last(WIDTHS) then
      let curr_width := next(curr_width,WIDTHS);
   else {
      let nPAT := nPAT + 1;
      let {w in WIDTHS} nbr[w,nPAT] := pattern[w];
      let curr_sum := curr_sum - pattern[last(WIDTHS)] * last(WIDTHS);
      let pattern[last(WIDTHS)] := 0;
      let curr_width := min {w in WIDTHS: pattern[w] > 0} w;
      if curr_width < Infinity then {
         let curr_sum := curr_sum - curr_width;
         let pattern[curr_width] := pattern[curr_width] - 1;
         let curr_width := next(curr_width,WIDTHS);
         }
      else break;
      }
   }
```

Pattern Enumeration

Script (solve, report)

```
option solver gurobi;

solve;

printf "\n%5i patterns, %3i rolls", nPAT, sum {j in 1..nPAT} Cut[j];
printf "\n\n  Cut   ";
printf {j in 1..nPAT: Cut[j] > 0}: "%3i", Cut[j];
printf "\n\n";

for {i in WIDTHS} {
   printf "%7.2f ", i;
   printf {j in 1..nPAT: Cut[j] > 0}: "%3i", nbr[i,j];
   printf "\n";
   }

printf "\nWASTE = %5.2f%%\n\n",
   100 * (1 - (sum {i in WIDTHS} i * orders[i]) / (roll_width * Number));
```

Scripting

Pattern Enumeration

Results

```
ampl: include cutPatEnum.run

Gurobi 5.6.0: optimal solution; objective 18
7 simplex iterations

43 patterns, 18 rolls

  Cut      2  2  3 11

  18.76    3  2  0  0
  17.46    0  1  3  2
   7.56    1  1  1  3
   6.77    0  0  0  1

WASTE =  2.34%
```

Scripting

Pattern Enumeration

Data 2

```
param roll_width := 349 ;

param: WIDTHS: orders :=
       28.75     7
       33.75    23
       34.75    23
       37.75    31
       38.75    10
       39.75    39
       40.75    58
       41.75    47
       42.25    19
       44.75    13
       45.75    26 ;
```

Scripting

Pattern Enumeration

Results 2

```
ampl: include cutPatEnum.run

Gurobi 4.6.1: optimal solution; objective 34
291 simplex iterations

54508 patterns,  34 rolls

  Cut     8  1  1  1  3  1  1  1  1  2  7  2  3  1  1

  45.75   3  2  0  0  0  0  0  0  0  0  0  0  0  0  0
  44.75   1  2  2  1  0  0  0  0  0  0  0  0  0  0  0
  42.25   0  2  0  0  4  2  2  1  0  0  0  0  0  0  0
  41.75   4  2  0  2  0  0  0  0  2  1  1  0  0  0  0
  40.75   0  0  4  4  1  4  3  0  2  3  1  6  3  2  2
  39.75   0  0  0  0  0  0  0  2  0  0  5  0  0  2  0
  38.75   0  0  1  0  0  0  0  0  4  0  0  0  0  2  3
  37.75   0  0  0  0  0  0  1  0  0  4  0  0  6  2  4
  34.75   0  0  0  0  4  0  3  1  0  0  0  3  0  1  0
  33.75   0  0  0  0  0  3  0  4  0  1  2  0  0  0  0
  28.75   0  0  2  2  0  0  0  2  1  0  0  0  0  0  0

WASTE =  0.69%
```

Scripting

Pattern Enumeration

Data 3

```
param roll_width := 172 ;

param: WIDTHS: orders :=
       25.000     5
       24.750    73
       18.000    14
       17.500     4
       15.500    23
       15.375     5
       13.875    29
       12.500    87
       12.250     9
       12.000    31
       10.250     6
       10.125    14
       10.000    43
        8.750    15
        8.500    21
        7.750     5 ;
```

Pattern Enumeration

Results 3 (using a subset of patterns)

```
ampl: include cutPatEnum.run

Gurobi 4.6.1: optimal solution; objective 33
722 simplex iterations
40 branch-and-cut nodes

273380 patterns,  33 rolls

  Cut     1  1  1  1  4  4  4  1  1  2  5  2  1  1  1  3

  25.00   2  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0
  24.75   1  2  1  0  5  4  3  2  2  2  2  1  1  0  0  0
  18.00   0  0  0  0  1  0  0  1  0  0  0  1  1  5  1  0
  17.50   0  3  0  0  0  0  0  0  0  0  0  0  0  0  1  0
  ........
  10.12   0  2  0  0  0  1  2  0  0  0  0  0  0  0  0  0
  10.00   0  0  0  0  0  2  0  1  3  0  6  0  0  2  0  0
   8.75   0  0  1  0  0  0  0  0  0  2  0  2  0  0  0  2
   8.50   0  0  2  0  0  2  0  0  0  0  0  4  3  0  0  0
   7.75   0  0  0  0  1  0  0  1  0  0  0  0  0  0  0  0

WASTE =  0.62%
```

Cutting via Pattern Generation

Generate the pattern list by a series of solves

- Solve LP relaxation using subset of patterns
- Add "most promising" pattern to the subset
  - Minimize reduced cost given dual values
  - Equivalent to a knapsack problem
- Iterate as long as there are promising patterns
  - Stop when minimum reduced cost is zero
- Solve IP using all patterns found

Scripting

Pattern Generation

Cutting model

```
set WIDTHS ordered by reversed Reals;
param orders {WIDTHS} > 0;

param nPAT integer >= 0, <= maxPAT;
param nbr {WIDTHS,1..nPAT} integer >= 0;

var Cut {1..nPAT} integer >= 0;

minimize Number:
   sum {j in 1..nPAT} Cut[j];

subj to Fulfill {i in WIDTHS}:
   sum {j in 1..nPAT} nbr[i,j] * Cut[j] >= orders[i];
```

Scripting

Pattern Generation

Knapsack model

```
param roll_width > 0;
param price {WIDTHS} default 0.0;

var Use {WIDTHS} integer >= 0;

minimize Reduced_Cost:
   1 - sum {i in WIDTHS} price[i] * Use[i];

subj to Width_Limit:
   sum {i in WIDTHS} i * Use[i] <= roll_width;
```

Pattern Generation

Script (problems, initial patterns)

```
model cutPatGen.mod;
data Sorrentino.dat;

problem Cutting_Opt: Cut, Number, Fill;
   option relax_integrality 1;
   option presolve 0;
problem Pattern_Gen: Use, Reduced_Cost, Width_Limit;
   option relax_integrality 0;
   option presolve 1;

let nPAT := 0;
for {i in WIDTHS} {
   let nPAT := nPAT + 1;
   let nbr[i,nPAT] := floor (roll_width/i);
   let {i2 in WIDTHS: i2 <> i} nbr[i2,nPAT] := 0;
   };
```

Pattern Generation

Script (generation loop)

```
repeat {
   solve Cutting_Opt;
   let {i in WIDTHS} price[i] := Fill[i].dual;
   solve Pattern_Gen;
   printf "\n%7.2f%11.2e  ", Number, Reduced_Cost;
   if Reduced_Cost < -0.00001 then {
      let nPAT := nPAT + 1;
      let {i in WIDTHS} nbr[i,nPAT] := Use[i];
   }
   else break;
   for {i in WIDTHS} printf "%3i", Use[i];
};
```

Pattern Generation

Script (final integer solution)

```
option Cutting_Opt.relax_integrality 0;
option Cutting_Opt.presolve 10;
solve Cutting_Opt;

if Cutting_Opt.result = "infeasible" then
   printf "\n*** No feasible integer solution ***\n\n";
else {
   printf "Best integer: %3i rolls\n\n", sum {j in 1..nPAT} Cut[j];
   for {j in 1..nPAT: Cut[j] > 0} {
      printf "%3i of:", Cut[j];
      printf {i in WIDTHS: nbr[i,j] > 0}: "%3i x %6.3f", nbr[i,j], i;
      printf "\n";
      }
   printf "\nWASTE = %5.2f%%\n\n",
      100 * (1 - (sum {i in WIDTHS} i * orders[i]) / (roll_width * Number));
   }
```

Pattern Generation

Results (relaxation)

```
ampl: include cutpatgen.run
  20.44  -1.53e-01    1  3  2  0
  18.78  -1.11e-01    0  1  3  0
  18.37  -1.25e-01    0  1  0  3
  17.96  -4.17e-02    0  6  0  1
  17.94  -1.00e-06

Optimal relaxation: 17.9412 rolls
 10.0000 of:  1 x 6.770  3 x  7.560  2 x 17.460
  4.3333 of:  1 x 7.560  3 x 17.460
  3.1961 of:  1 x 7.560  3 x 18.760
  0.4118 of:  6 x 7.560  1 x 18.760
WASTE =  2.02%
```

Pattern Generation

Results (integer)

```
Rounded up to integer: 20 rolls

  Cut    10  5  4  1

   6.77   1  0  0  0
   7.56   3  1  1  6
  17.46   2  3  0  0
  18.76   0  0  3  1

WASTE = 12.10%

Best integer: 19 rolls

  Cut    10  5  3  1

   6.77   1  0  0  0
   7.56   3  1  1  6
  17.46   2  3  0  0
  18.76   0  0  3  1

WASTE =  7.48%
```

General Observations

Scripts in practice

- Large and complicated
  - Multiple files
  - Hundreds of statements
  - Millions of statements executed
- Run within broader applications

Prospective improvements

- Faster loops
- True script functions
  - Arguments and return values
  - Local sets & parameters
  - Callback functions

But . . .

Scripting

Limitations

Performance

- Interpreted language
- Complex set & data structures

Expressiveness

- Based on a declarative language
- Not object-oriented

So . . .

APIs (application programming interfaces)

Bring the modeling language to the programmer

- Data and result management in a general-purpose programming language
- Modeling and solving through calls to AMPL

Development details

- Partnership with OptiRisk Systems
  - Christian Valente, principal developer
- Long-term development & maintenance by AMPL
  - Victor Zverovich, project coordinator

Cutting Revisited

Hybrid approach

- Control & pattern creation from a programming language
  - Pattern enumeration: finding all patterns
  - Pattern generation: solving knapsack problems
- Model & modeling commands in AMPL

Two programming languages

- Java
- MATLAB

Key to examples

- AMPL entities
- Java/MATLAB objects
- Java/MATLAB methods for working with AMPL
- Java/MATLAB functions

AMPL Model File

Basic 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 rolls {WIDTHS,PATTERNS} >= 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;
```

Pattern Enumeration in MATLAB

Load & generate data, set up AMPL model

```
function cuttingEnum(dataFile)

% Get data from .mat file: roll_width, overrun, widths, orders
load(dataFile);

% Generate pattern matrix
[widthsDec,ind] = sort(widths,'descend');
patmat = patternEnum(roll_width,widthsDec);
patmat(:,ind) = patmat;

% Initialize and load cutting-stock model from file
ampl = AMPL();
ampl.read('cut.mod');
```

Pattern Enumeration in MATLAB

Send data to AMPL

```
% Send scalar values
ampl.getParameter('overrun').setValues(overrun);
ampl.getParameter('nPatterns').setValues(length(patmat));

% Send order vector
WidthOrder = DataFrame(1, 'WIDTHS', 'order');
WidthOrder.setColumn('WIDTHS', num2cell(widths));
WidthOrder.setColumn('order', orders);
ampl.setData(WidthOrder, 'WIDTHS');

% Send pattern matrix
AllPatterns = DataFrame(2, 'WIDTHS', 'PATTERNS', 'rolls');
AllPatterns.setMatrix(patmat', num2cell(widths), num2cell(1:length(patmat)));
ampl.setData(AllPatterns)
```

AMPL API

Pattern Enumeration in MATLAB

Solve and report

```
% Solve
ampl.setOption('solver' ,'gurobi');
ampl.solve

% Retrieve solution
CuttingPlan = ampl.getVariable('Cut').getValues();
cutvec = CuttingPlan.getColumnAsDoubles('val');

% Display solution
cuttingPlot (roll_width, widths, patmat(cutvec>0,:), cutvec(cutvec>0))
```

AMPL API

Pattern Enumeration in MATLAB

Enumeration routine

```
function patmat = patternEnum(rollwidth,widths)
if length(widths) == 1
    patmat = floor(rollwidth/widths(1));
else
   patmat = [];
   for n = floor(rollwidth/widths(1)):-1:0
       patnew = patternEnum (rollwidth-n*widths(1), widths(2:end));
       patmat = [patmat; n*ones(size(patnew,1),1) patnew];
   end
end
```

AMPL API

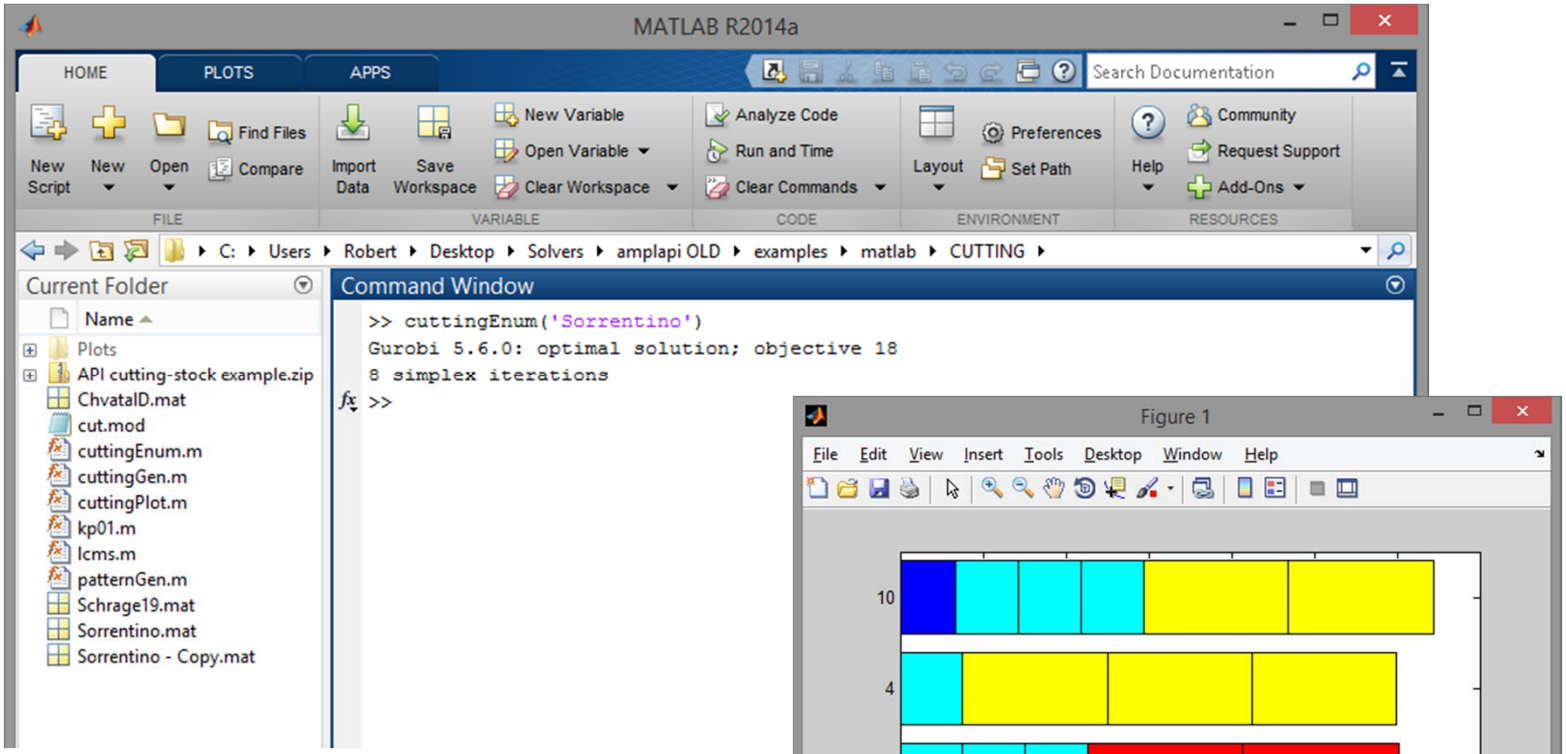
Pattern Enumeration in MATLAB

Plotting routine

```
function cuttingPlot (roll_width,widths,patmat,cutvec)

plotmat = zeros(length(cutvec),sum(max(patmat)));
colors = jet(length(widths));

plotpos = 0;

for j = 1:length(widths)
   for i = 1:length(cutvec)
      plotmat(i,plotpos+1:plotpos+patmat(i,j)) = widths(j);
   end
   for i = 1:max(patmat(:,j))
      colormat(plotpos+i,:) = colors(j,:);
   end
   plotpos = plotpos + max(patmat(:,j));
end

colormap(colormat); shading faceted
h = barh(plotmat,'stacked');

set (h, 'edgecolor','black')
set(gca,'YTickLabel',num2cell(cutvec))
```

AMPL API

Pattern Enumeration in MATLAB



Pattern Enumeration in Java

Generate patterns, set up AMPL model

```
public static void main(String[] args) throws IOException {
  import static com.ampl.examples.CuttingStock.Sorrentino;
  int[] sortedWidths = widths.clone();
  sortDescending(sortedWidths);
  ArrayList<Integer> patterns = new ArrayList<>();
  patternEnum (roll_width, sortedWidths, 0, patterns);

  // Initialize and load cutting-stock model from file
  AMPL ampl = new AMPL();
  try {
    ampl.read("cut.mod");
```

Pattern Enumeration in Java

Send data to AMPL

```
ampl.getParameter("overrun").setValues(overrun);
int numPatterns = patterns.size() / widths.length;
ampl.getParameter("nPatterns").setValues(numPatterns);

DataFrame widthOrder = new DataFrame(1, "WIDTHS", "order");
widthOrder.setColumn("WIDTHS", widths);
widthOrder.setColumn("order", orders);
ampl.setData(widthOrder, true);

DataFrame allPatterns = new DataFrame(2, "WIDTHS", "PATTERNS", "rolls");
for (int i = 0; i < widths.length; i++) {
  for (int j = 0; j < numPatterns; j++) {
    allPatterns.addRow(
      sortedWidths[i], j + 1, patterns.get(j * widths.length + i));
  }
}
ampl.setData(allPatterns, false);
```

AMPL API

Pattern Enumeration in Java

Solve and report solution

```
    ampl.setOption("solver", "gurobi");
    ampl.solve();
    printSolution (ampl.getVariable("Cut"), ampl.getParameter("rolls"));
  } finally {
    ampl.close();
  }
}
```

Pattern Generation in MATLAB

Set up AMPL, get data

```
function cuttingGen(dataFile)

% Initialize
ampl = AMPL();

% Load cutting-stock model from file
ampl.read('cut.mod');
Cut = ampl.getVariable('Cut');
Limits = ampl.getConstraint('FinishedRollLimits');

% Get data from .mat file: roll_width, overrun, widths, orders
load(dataFile);
```

Pattern Generation in MATLAB

Send data to AMPL

```
% Send scalar values
ampl.getParameter('overrun').setValues(overrun);
ampl.getParameter('nPatterns').setValues(length(widths));

% Send order vector
WidthOrder = DataFrame(1, 'WIDTHS', 'order');
WidthOrder.setColumn('WIDTHS', num2cell(widths));
WidthOrder.setColumn('order', orders);
ampl.setData(WidthOrder, 'WIDTHS');

% Generate and send initial pattern matrix
maxpat = floor(roll_width./widths);
patmat = diag(maxpat);
InitPatterns = DataFrame(2, 'WIDTHS', 'PATTERNS', 'rolls');
InitPatterns.setMatrix(patmat, num2cell(widths), num2cell(1:length(widths)));
ampl.setData(InitPatterns);
```

Pattern Generation in MATLAB

Set up for generation loop

```
% Set solve options
ampl.setOption('solver','gurobi');
ampl.setOption('relax_integrality','1');

% Set up DataFrame for sending AMPL new patterns
ampl.eval('param newpat {WIDTHS} integer >= 0;');
NewPattern = DataFrame(1, 'WIDTHS', 'newpat');
NewPattern.setColumn('WIDTHS', num2cell(widths));

% Compute multiplier for integer weights
[n,d] = rat(widths);
intmult = lcms(d);
```

Pattern Generation in MATLAB

Loop 1: Retrieve duals & look for new pattern

```
while 1

   ampl.solve

   DualPrices = Limits.getValues;
   dualvec = DualPrices.getColumnAsDoubles('dual');

   wgt = []; val = [];
   for w = 1:length(widths)
      if dualvec(w) > 0
         wgt = [wgt widths(w)*ones(1,maxpat(w))];
         val = [val dualvec(w)*ones(1,maxpat(w))];
      end
   end

   % Solve knapsack problem for potential new pattern

   [kmax,z] = kp01 (round(intmult*wgt), val, intmult*roll_width);

   if kmax < 1.000001
     break;
   end
```

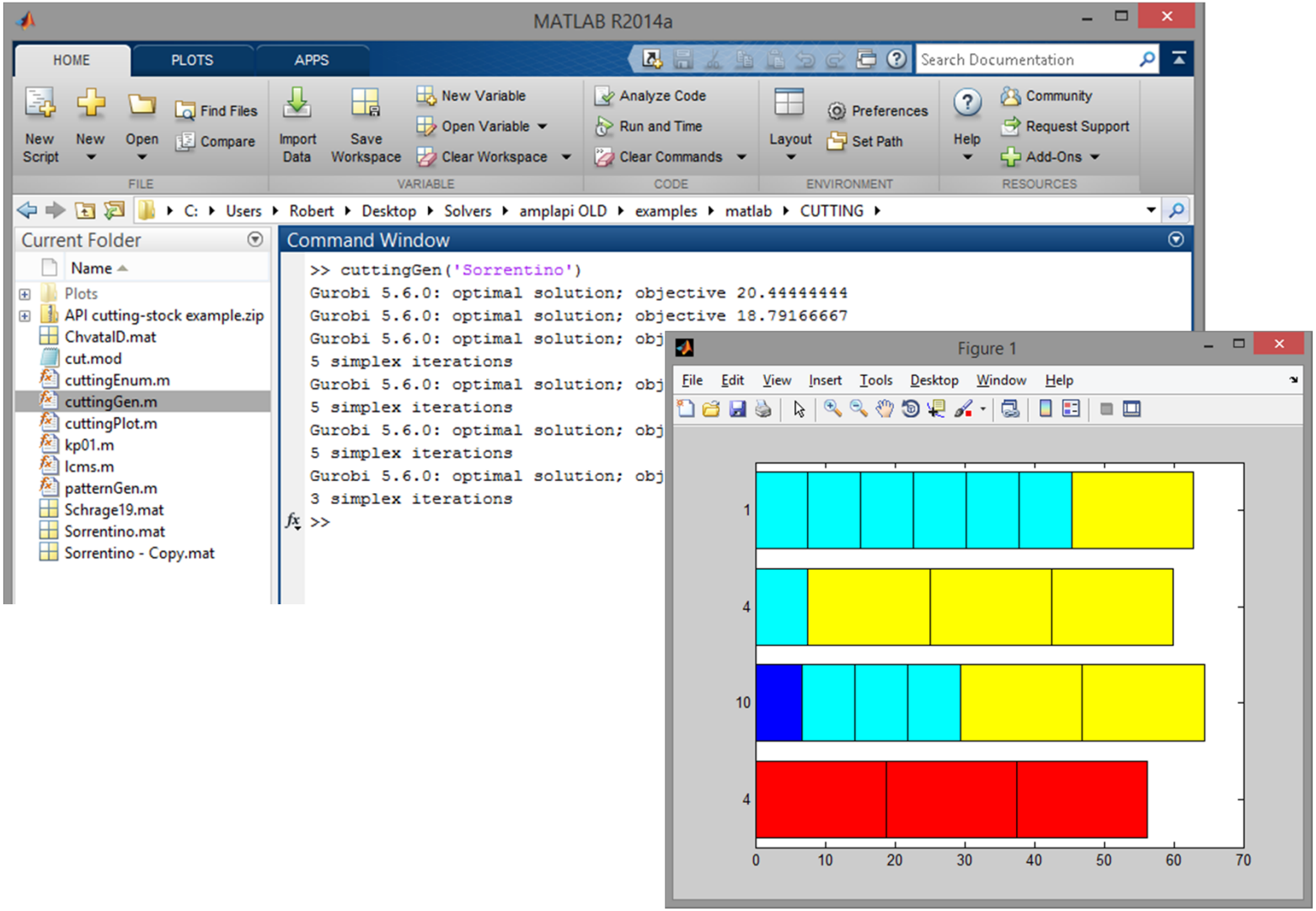
Pattern Generation in MATLAB

Loop 2: Send new pattern to AMPL

```
    widthlist = wgt(z);
    for w = 1:length(widths)
       newpat(w) = length(find(widthlist==widths(w)));
    end
    patmat = [patmat; newpat];

    NewPattern.setColumn('newpat', newpat);
    ampl.setData(NewPattern);

    ampl.eval('let nPatterns := nPatterns + 1;');
    ampl.eval('let {w in WIDTHS} rolls[w,nPatterns] := newpat[w];');
end

% Compute and display integer solution

ampl.setOption('relax_integrality','0');
ampl.solve;

CuttingPlan = Cut.getValues();
cutvec = CuttingPlan.getColumnAsDoubles('val');

cuttingPlot (roll_width, widths, patmat(cutvec>0,:), cutvec(cutvec>0))
```

AMPL API

Pattern Generation in MATLAB



Data Transfer: Alternatives

Process

- Define symbolic sets & parameters in AMPL model
- Create corresponding objects in program
- Transfer data using API methods
  - Program to AMPL
  - AMPL to program

Methods for transfer between . . .

- Scalar values
- Collections of values
  - AMPL indexed expressions
  - Java arrays, MATLAB matrices
- Relational tables
  - AMPL "table" structures
  - API DataFrame objects in Java, MATLAB

Deployment: Alternatives

Scripting: Give (temporary) control to AMPL

- Write needed files
- Invoke AMPL to run some scripts
- Read the files that AMPL leaves on exit

API: Interact with AMPL

- Execute AMPL statements individually
- Read model, data, script files when convenient
- Exchange data tables directly with AMPL
  - populate sets & parameters
  - invoke any available solver
  - extract values of variables & result expressions

. . . all embedded within your program's logic

Availability

Java API version 1.0 released
MATLAB API version 1.0 released

- Add-ons to all AMPL distributions
- Download from www.ampl.com/products/api/

C++ API in final development

- Release planned for this summer

More languages to follow

- R
- Python
- .NET: C#, Visual Basic

www.ampl.com


Firefox
AMPL license generation
AMPL download
AMPL | STREAMLINED MODELING FOR REAL OPTIMIZATION
70.38.71.71
HOME
PRODUCTS
RESOURCES
ABOUT US
TRY AMPL
AMPL
STREAMLINED MODELING FOR REAL OPTIMIZATION
Build optimization into your large-scale applications — quickly and reliably — using AMPL's powerful yet intuitive algebraic modeling system.
DATA
MODEL
SOLVE
DEPLOY
ANALYZE
AMPL FOR BUSINESS
Streamlined optimization development in business applications of all kinds.
Read More
AMPL FOR TEACHING
Free AMPL and solvers. Full-featured, time-limited. Easy to install & distribute.
Read More
AMPL FOR RESEARCH
Optimization modeling for engineering, science, economics, management.
Read More
SOLVERS
FREE SOLVER TRIALS!
Buy from us >>
Open-source optimizers >>
Full solver list >>
CPLEX
Gurobi
Xpress
CONOPT
KNITRO
MINOS
SNOPT
AMPL
The AMPL system is a sophisticated modeling tool that supports the entire optimization modeling lifecycle: development, testing, deployment, and maintenance.
By using a high-level representation that represents optimization models in the same ways that people think about them, AMPL promotes rapid development and reliable results.
AMPL integrates a modeling language for describing optimization data, variables, objectives, and constraints; a command language for browsing models and analyzing results; and a scripting language
WHAT'S NEW?
March 6-8
6th INFORMS Optimization Society Conference, Houston, Texas.
AMPL sponsorship of this event & table in the exhibit area

AMPL Readings

- R. Fourer, "Modeling Languages versus Matrix Generators for Linear Programming." *ACM Transactions on Mathematical Software* **9** (1983) 143–183.
- R. Fourer, D.M. Gay, B.W. Kernighan, "A Modeling Language for Mathematical Programming." *Management Science* **36** (1990) 519–554.
- Robert Fourer, "Database Structures for Mathematical Programming Models." *Decision Support Systems* **20** (1997) 317–344.
- R. Fourer, D.M. Gay, B.W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, Belmont, CA (first edition 1993, second edition 2003).
- Robert Fourer, On the Evolution of Optimization Modeling Systems. M. Groetschel (ed.), *Optimization Stories*. Documenta Mathematica (2012) 377-388.