Adding Optimization to Your Applications

Efficient and Effective Strategies
Using Algebraic Modeling with AMPL

Robert Fourer
4er@ampl.com
AMPL Optimization Inc.
www.ampl.com — +1 773-336-AMPL

27th European Conference on Operational Research
Session TA-09, Vendor Session 1
Adding Optimization to Your Applications: Efficient and Effective Strategies Using Algebraic Modeling with AMPL

Algebraic modeling languages have been developed with the goal of making optimization models much easier to develop, debug, and maintain. Moreover, it is not necessary to give up these advantages when embedding a model into a larger system or deploying it to users. This talk describes features of the AMPL modeling language that work together to help you build optimization models quickly and use completed models effectively. Following an introduction to development of algebraic optimization models, the presentation introduces two distinct modeling language facilities commonly used to integrate models into projects and 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 approaches are contrasted using a common example.
The Optimization Modeling Cycle

Steps

- Communicate with problem owner
- Build model
- Prepare data
- Generate optimization problem
- Submit problem to solver
  * CPLEX, Gurobi, KNITRO, CONOPT, MINOS, . . .
- Report & analyze results
- Repeat!

Goals for optimization software

- Do this quickly and reliably
- Get results before client loses interest
- Deploy for application
Optimization Modeling Languages

Two forms of an optimization problem

- Modeler’s form
  - Mathematical description, easy for people to work with
- Algorithm’s form
  - Explicit data structure, easy for solvers to compute with

Idea of a modeling language

- A computer-readable modeler’s form
  - You write optimization problems in a modeling language
  - Computers translate to algorithm’s form for solution

Advantages of a modeling language

- Faster modeling cycles
- More reliable modeling
- More maintainable applications
Algebraic Modeling Languages

Formulation concept

- Define data in terms of sets & parameters
  - Analogous to database keys & records
- Define decision variables
- Minimize or maximize a function of decision variables
- Subject to equations or inequalities that constrain the values of the variables

Advantages

- Familiar
- Powerful
- Implemented
The AMPL Modeling Language

Features
- Algebraic modeling language
- Variety of data sources
- Connections to all solver features

Advantages
- Powerful, general expressions
- Natural, easy-to-learn design
- Efficient processing scales well with problem size

3 ways to use
- Command language
- Scripting language
- Programming interface (API)
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 $i \in W$
Roll cutting

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**

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

  number of rolls of width \( i \) cut
  must be at least the number ordered
Roll Cutting

AMPL Formulation

Symbolic model

```AMPL
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];
```

\[ \sum_{j=1}^{n} a_{ij} X_j \geq b_i \]
Roll Cutting

AMPL Formulation \textit{(cont’d)}

Explicit data (independent of model)

<table>
<thead>
<tr>
<th>param: WIDTHS: orders :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.77 10</td>
</tr>
<tr>
<td>7.56 40</td>
</tr>
<tr>
<td>17.46 33</td>
</tr>
<tr>
<td>18.76 10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>param nPAT := 9 ;</th>
</tr>
</thead>
<tbody>
<tr>
<td>param nbr: 1 2 3 4 5 6 7 8 9 :=</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>6.77 0 1 1 0 3 2 0 1 4</td>
</tr>
<tr>
<td>7.56 1 0 2 1 1 4 6 5 2</td>
</tr>
<tr>
<td>17.46 0 1 0 2 1 0 1 1 1</td>
</tr>
<tr>
<td>18.76 3 2 2 1 1 0 0 0 0</td>
</tr>
</tbody>
</table>

Roll Cutting
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
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
```
IDE for Command Language
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
Scripting

Parametric Analysis (cont’d)

Script (setup)

```plaintext
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;
```
Scripting

Parametric Analysis (cont’d)

Script (looping and reporting)

```plaintext
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
';
printf {frac in INCR}: '%5.2f%5d
', frac, incrObj[frac];
```
Scripting

Parametric Analysis (cont’d)

Script run

```ampl
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:
```
Scripting

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
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];
Pattern Enumeration

Data

param roll_width := 64.50 ;
param: WIDTHS: orders :=
    6.77  10
    7.56  40
    17.46 33
    18.76 10 ;
Scripting

Pattern Enumeration

Script (initialize)

```plaintext
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;
```
Pattern Enumeration

Script (loop)

```plaintext
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;
  }
}
```
Scripting

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%
Pattern Enumeration

Data 2

```plaintext
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 ;
```
Pattern Enumeration

Results 2

ampl: include cutPatEnum.run

Gurobi 4.6.1: optimal solution; objective 34
291 simplex iterations

54508 patterns, 34 rolls

<table>
<thead>
<tr>
<th>Cut</th>
<th>8</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>7</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>45.75</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>44.75</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>42.25</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>41.75</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>40.75</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>39.75</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>38.75</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>37.75</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>34.75</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>33.75</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28.75</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

WASTE = 0.69%
## Scripting

### Pattern Enumeration

#### Data 3

```plaintext
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

<table>
<thead>
<tr>
<th>Cut</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.00</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>24.75</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>17.50</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10.12</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8.75</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8.50</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7.75</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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
Pattern Generation

Cutting model

```plaintext
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

```plaintext
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;
```
**Scripting**

**Pattern Generation**

**Script (problems, initial patterns)**

```plaintext
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;
};
```
Scripting

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];
};
```
Scripting

Pattern Generation

Script (final integer solution)

```plaintext
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));
}
```
## Scripting

### Pattern Generation

#### Results (relaxation)

```plaintext
ampl: include cutpatgen.run

<table>
<thead>
<tr>
<th>Value</th>
<th>Coefficient</th>
<th>Rows</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.44</td>
<td>-1.53e-01</td>
<td>1</td>
<td>3  2  0</td>
</tr>
<tr>
<td>18.78</td>
<td>-1.11e-01</td>
<td>0</td>
<td>1  3  0</td>
</tr>
<tr>
<td>18.37</td>
<td>-1.25e-01</td>
<td>0</td>
<td>1  0  3</td>
</tr>
<tr>
<td>17.96</td>
<td>-4.17e-02</td>
<td>0</td>
<td>6  0  1</td>
</tr>
<tr>
<td>17.94</td>
<td>-1.00e-06</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Rounded up to integer: <strong>20 rolls</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cut</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>WASTE = 12.10%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Best integer: <strong>19 rolls</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cut</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>WASTE = 7.48%</td>
</tr>
</tbody>
</table>
Scripting

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**

```AMPL
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;
```
**AMPL API**

**Pattern Enumeration in MATLAB**

*Load & generate data, set up AMPL model*

```matlab
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');
```
**AMPL API**

**Pattern Enumeration in MATLAB**

**Send data to AMPL**

```matlab
% Send scalar values
ampl.getParameter('overrun').setValues(overrun);
ampl.getParameter('nPatters').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)
```
Pattern Enumeration in MATLAB

Solve and report

```matlab
% 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**

```matlab
function patmat = patternEnum(rollwidth, widths)
if length(widths) == 1
    patmat = floor(rollwidth/widths(1));
else
    patmat = [];
    for n = floor(rollwidth/widths(1)):1
        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*

```matlab
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))
```
Pattern Enumeration in MATLAB
**AMPL API**

**Pattern Enumeration in Java**

*Generate patterns, set up AMPL model*

```java
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

```java
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);
```
Pattern Enumeration in Java

Solve and report solution

```java
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**

```matlab
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);
```
**AMPL API**

**Pattern Generation in MATLAB**

**Send data to AMPL**

```matlab
% Send scalar values
ampl.getParameter('overrun').setValues(overrun);
ampl.getParameter('nPattersn').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);
```
**AMPL API**

**Pattern Generation in MATLAB**

**Set up for generation loop**

```matlab
% 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

```matlab
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
end
```
**AMPL API**

**Pattern Generation in MATLAB**

**Loop 2: Send new pattern to AMPL**

```matlab
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))
```
Pattern Generation in MATLAB
**AMPL API**

**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
**AMPL API**

**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
AMPL API

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
**AMPL Readings**


