Writing .nl Files

David M. Gay
Optimization and Uncertainty Estimation
Sandia National Laboratories [*]
Albuquerque, NM

November 30, 2005

Abstract

AMPL® is a language and environment for describing mathematical programming problems and solving them. The AMPL processor interprets the AMPL language and invokes separate solvers to actually solve problems. It conveys problem information to solvers in "`.nl`" files. This paper describes the structure of a `.nl` file in enough detail to permit writing such a file without using the AMPL processor.

1 Introduction

AMPL [2, 4] facilitates expressing, solving, and analyzing mathematical programming problems, i.e., problems of the form

$$\begin{array}{rc} \text{minimize} & f(x) \\ \text{subject to} & \ell \le c(x) \le u \end{array}$$

in which $x \in \mathbf{R}^n$, $f : \mathbf{R}^n \to \mathbf{R}^1$ and $c : \mathbf{R}^n \to \mathbf{R}^m$, with f and c given algebraically and perhaps some x_i restricted to integer values. AMPL communicates with solvers by writing a "`.nl`" file, which contains a problem representation, including the numbers of various kinds of variables, constraints, and objectives, the linear terms of constraints and objectives, and expression graphs for the nonlinear parts of constraints, objectives, and "defined variables" (named common expressions).

To write a `.nl` file manually, it is perhaps simplest to invoke the `fg_write()` routine in the AMPL/solver interface library (hereafter called the ASL, documented in [5], with source freely available from *netlib*, as described in [5]). This could be complicated from, say, Java, as the ASL is only designed for direct

[*]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000. This document is released as SAND2005-7907P.

```
set I = 1..9;
var x{I};
function myfunc;

var t{i in 1..3} = x[i]^2 + 1 + sum{j in 8..9} (i+j)*x[j];
var u{i in 1..2} = x[7+i]^2 + 2 + sinh(x[1] + 2*t[2] + 6*x[6]);

maximize zip: if t[2] >= 0 then -t[2]^3 else -t[2]^2;
minimize zap: sin(t[1]) + cos(2*t[2]) + 4*x[4] + 5*x[5] + x[6]^2 + x[7]^2;
minimize zot: cosh(<<1,2;3,4,5>>x[6]);

s.t. c1: t[2] + sin(t[3]) <= 4;
s.t. c2: x[5] + cos(x[6]) >= 3;
s.t. c3: sum{i in 3..7} i*x[i] = 1;
s.t. c4: 4.3 <= x[5] + myfunc(t[2], x[3]*x[6], "some string") <= 15.5;
s.t. b45{i in 4..5}: x[i] >= i;
s.t. b1: x[1] <= 3.5;
s.t. b2: -1 <= x[2] <= 2;
s.t. b3{i in 8..9}: 0 <= x[i] <= 0.1*i;
s.t. lc{i in 1..2}: x[6] + x[7] >= 2.5 ==> (x[5] + x[6]^2)^2 + u[i] <= 35;

suffix zork;
let{i in 2.._nvars} _var[i].zork := i;
option nl_comments 1, auxfiles rc;
write gsilly;
```

Figure 1: sample AMPL Model and commands: silly.amp

use with C and C++. Java could use the Java Native Interface (JNI) to talk directly with the ASL, but that would make it necessary to construct expression graphs in the right form. The rest of this document summarizes how to write a `.nl` file "from scratch".

Each `.nl` file begins with a header that is discussed in the next section, followed by segments of various kinds. Several kinds of segments include expression graphs, which are described in §3. A description of the various segments then appears in §4.

Seeing examples may prove helpful. Figure 1 shows an AMPL model and commands (file `silly.amp`) that give rise to a `.nl` file, `silly.nl`, portions of which we use in illustrations below. The `write` command at the end of Figure 1 causes the writing of an ASCII `silly.nl`. The preceding `option` command requests (via "`nl_comments 1`") comments in `silly.nl`. The comments are the text in many of the tables below, such as Table 3, that begins with `#` and extends to the end of the line. The option command also requests (via "`auxfiles rc`") the writing of auxiliary files `silly.row` and `silly.col` with the AMPL names of the constraints, objectives and variables. (The suffixes `.row` and `.col` hearken back to linear programming, where variables correspond to columns of the constraint matrix and constraints and objectives to rows. Solvers ordinarily

silly.row	silly.col
c1	x[2]
c2	x[6]
c4	x[7]
c3	x[8]
lc[1]	x[9]
lc[2]	x[3]
zap	x[5]
zot	x[1]
	x[4]

Table 1: `.row` and `.col` files from `silly.amp`.

```
g3 1 1 0 # problem silly
 9 4 3 1 1 2 # vars, constraints, objectives, ranges, eqns, lcons
 3 3 # nonlinear constraints, objectives
 0 0 # network constraints: nonlinear, linear
 7 8 5 # nonlinear vars in constraints, objectives, both
 0 1 0 1 # linear network variables; functions; arith, flags
 0 0 0 0 0 # discrete variables: binary, integer, nonlinear (b,c,o)
 17 5 # nonzeros in Jacobian, gradients
 5 4 # max name lengths: constraints, variables
 2 0 0 3 1 # common exprs: b,c,o,c1,o1
```

Figure 2: header of `silly.nl`

do not use the `.row` and `.col` files, so AMPL does not write them by default, but the names in them can be useful in error messages.) Table 1 shows the contents of `silly.row` and `silly.col`, which show how AMPL has permuted the constraints and variables.

2 Header

The first 10 lines of a `.nl` file are a header that gives problem statistics in ASCII format. The rest is either more ASCII text or a binary equivalent (as indicated on the `.nl` file's first line). For example, Figure 2 shows the header that results from `silly.amp` (Figure 1).

The first character of a `.nl` file is 'g' if the whole file is in ASCII format and 'b' if the file is in binary format (after the first 10 lines). AMPL always writes explanatory comments — text starting with '#' — in the header, regardless of the setting of option `nl_comments`, but such comments are not required and are ignored by the ASL. The numbers on the first line matter to AMPL; for other uses, it is best simply to supply the ones shown above.

The rest of the header gives problem statistics, as described in the comment

name	*meaning*
`n_var`	number of variables
`n_con`	number of algebraic constraints (1)
`n_obj`	number of objectives
`n_lcon`	number of logical constraints

Table 2: names of some header statistics.

on each line. The ASL considers the *i*-th algebraic constraint to be in the form

$$\ell_i \le c_i(x) \le u_i \qquad (1)$$

where ℓ_i and u_i are constants, possibly with $\ell_i = -\infty$ and/or $u_i = +\infty$, and $c_i(x)$ is the *body* of the constraint. Equations are constraints with $\ell_i = u_i$, and range constraints are those with $-\infty < \ell_i < u_i < +\infty$. The second line gives the numbers of variables, constraints, and objectives (of which there can be none, one, or several), and the numbers of range and equality constraints; both are included in the total number of algebraic constraints. Variables are ordered as described in Tables 3 and 4 of [5], because some solvers treat linear constraints and variables specially. You only need to worry about such permutations if you are using a solver that cares about them.

Logical constraints are an AMPL extension described in [3]. The final integer on line 2 of Figure 2 is the number of logical constraints, which are not included in the count of algebraic constraints. AMPL omits this final number if no logical constraints are present. In Figure 1, `lc[1]` and `lc[2]` are logical constraints.

For simplicity, the descriptions that follow assume an ASCII `.nl` file. Binary files consist of corresponding sequences of `int`, `double`, and string values, where a string consists of an `int` length followed by that many characters. (The newlines mentioned below do not appear in binary `.nl` files, and the decimal integers and floating-point values mentioned below become binary `int` (4 byte) and `double` (8 byte) values, respectively.)

Below, it will be convenient to refer to some of the problem statistics from the `.nl` header. For consistency with the ASL, we use the names summarized in Table 2 for these statistics.

3 Expression Graphs

Expression graphs in a `.nl` file convey the nonlinear parts of constraints, objectives, and defined variables. Expression graphs are expressed in Polish prefix notation: operator followed by operands. A numeric constant is denoted by "n" (the operator) followed by a decimal floating-point value (the operand, a possibly signed decimal string possibly followed by "e" and a signed integer). A reference to variable *i* is denoted by "v" followed by a decimal integer; for $0 \le i <$ `n_var`, v*i* is the value of a decision variable that the solver manipulates;

```
f0 3        #myfunc
v10         #t[2]
n1.23
h11:some string
```

Table 3: `.nl` fragment from `myfunc(t[2], 1.23, "some string")`

n	oper.	n	oper.	n	oper.
13	floor	14	ceil	15	abs
16	neg	34	not	37	tanh
38	tan	39	sqrt	40	sinh
41	sin	42	log10	43	log
44	exp	45	cosh	46	cos
47	atanh	49	atan	50	asinh
51	asin	52	acosh	53	acos

Table 4: unary operators.

for larger i, vi is the value of a defined variable. A call on an imported function is denoted by "f" followed by two decimal integers, i and n, which are the function number i given in a previous F segment and the number of arguments n passed in this call. Then come n expressions for the arguments. Among these argument expressions can be strings, introduced by "h" followed by a decimal integer length ℓ and a colon, followed by the ℓ characters in the string. For example, assuming the declarations in Figure 1, Table 3 shows the `.nl` fragment that would result from the invocation

```
myfunc(t[2], 1.23, "some string")
```

An operation on the results of other operations (including variables, numeric constants, imported-function evaluations) is introduced by "o" followed by a decimal integer n (the operation number) followed by expression graphs for the operation's operands. Operations fall into several classes, described below. Class numbers are given in solver interface file `op_type.hd` [6], which also mentions some classes that do not appear in `.nl` files, but are used by the `.nl` file readers of [5].

Unary operators (class 1) take one operand and are summarized in Table 4. The *not* operator turns zero into 1 and nonzero values into zero. *Neg* is the unary "minus" operator, i.e., negation. *Log* is the natural logarithm, while *log10* is the base-10 logarithm. The other unary operators are for standard mathematical functions. For example, the invocation `cos(x[6])` that appears in constraint `c2` of Figure 1 gives rise to the `.nl` fragment shown in Table 5.

Binary operators (class 2) and are summarized in Table 6 and, as the name suggests, take two operands. The *plus*, *minus*, *mult*, and *div* operators are the

```
o46        #cos
v1         #x[6]
```

Table 5: `.nl` fragment from `cos(x[6])`

n	oper.	n	oper.	n	oper.
0	plus	1	minus	2	mult
3	div	4	rem	5	pow
6	less	20	or	21	and
22	lt	23	le	24	eq
28	ge	29	gt	30	ne
48	atan2	55	intdiv	56	precision
57	round	58	trunc	73	iff

Table 6: binary operators.

usual arithmetic $+$, $-$, $\times$, $\div$, i.e., addition, subtraction, multiplication, and division operators. *Intdiv* is for integer division, i.e., the integer that agrees in sign with the quotient of the left (first) and right (second) operands and is the largest such integer in absolute value that does not exceed the absolute value of the quotient. *Rem* is the remainder operation, and *pow* is exponentiation, raising the left operand to the power of the right operand. The *or* operator returns 1 if either operand is nonzero and returns zero otherwise. Analogously, the *and* operator returns 1 if both operands are nonzero and returns zero otherwise. The *lt*, *le*, *eq*, *ge*, *gt*, and *ne* operators are the comparisons $<$, $\le$, $=$, $\ge$, $>$, and $\ne$, respectively, returning 1 if the indicated relation holds between the left and right operand, and zero otherwise. The *iff* operator returns 1 if both operands are nonzero or both are zero and returns zero otherwise. The *precision* operator rounds the left operand to the number of significant decimal digits given by the right operand, and the *trunc* and *round* operators truncate or round the first operand after the number of decimal places after (or, for negative right operand, before) the decimal point specified by the right operand.

As an example of binary operations, Table 7 shows the expression graph for `(x[5] + x[6]^2)^2`, which appears in the `lc` constraints of Figure 1.

Classes 3, 6, and 11 are for n-ary operators, which take several operands. The class numbers vary to help the `.nl` readers arrange derivative computations, but they all have the same form in a `.nl` file: an integer n follows the operation number. After the integer come n expression graphs, one for each operand. Table 8 summarizes the n-ary operations. The *min*, *max*, and *sum* operators are self-explanatory. *Count* returns the number of nonzero operands. The *numberof* operators return the number of values among the second and subsequent operands that equal the first; *numberof* has numeric arguments, while *numberofs* has string arguments (which can only be constants or the results of

```
o5        #^
o0        # +
v6        #x[5]
o5        #^
v1        #x[6]
n2
n2
```

Table 7: `.nl` fragment from `(x[5] + x[6]^2)^2`

n	oper.	n	oper.	n	oper.
11	min	12	max	54	sum
59	count	60	numberof	61	numberofs
70	and	71	or	74	alldiff

Table 8: n-ary operators.

if-then-else expressions). The n-ary *and* returns 1 if all operands are nonzero and zero otherwise; the n-ary *or* returns 1 if any operand is nonzero and zero otherwise. The *alldiff* operator returns 1 if all operands are distinct and 0 if at least two coincide.

As an example of the n-ary *sum* operator, Table 9 shows a fragment of `silly.nl` corresponding to the `sinh(...)` term in objective `zot` of Figure 1.

Class 4 is for piecewise-linear terms that AMPL did not linearize. They arise only as operands to nonlinear functions or in objectives or constraints when one specifies "`option pl_linearize 0;`" before writing a `.nl` file. Piecewise-linear terms have operation number 64 and consist of an integer n for the number of slopes, followed by $2n-1$ floating-point numbers, which are alternating slope and breakpoint values (starting and ending with a slope). Following these numeric

```
o40       #sinh
o54       #sumlist
3
v7        #x[1]
o2        #*
n2
v10       #t[2]
o2        #*
n6
v1        #x[6]
```

Table 9: `.nl` fragment from `sinh(x[1] + 2*t[2] + 6*x[6])`

```
o45      #cosh
o64      #<<<>>>
3
n3
n1
n4
n2
n5
v1       #x[6]
```

Table 10: `.nl` fragment from `cosh(<<1,2;3,4,5>>x[6])`

n	oper.	n	oper.	n	oper.
35	if	65	ifs	72	implies

Table 11: n-ary operators.

values is a variable reference, i.e., a `v` followed by an integer variable number number; the piecewise-linear expression is applied to the indicated variable (see [4]). For instance, Table 10 shows a fragment of `silly.nl` corresponding to the `cosh(...)` term in objective `zot` of Figure 1.

Finally, class 5 is for *if-then-else* expressions. Table 11 summarizes the relevant operations. All are followed by three operands, the first of which is a logical expression (the result of one of the above operators that is specified to return zero or 1; in the ASL, these operators return floating-point values). The second and third operands are expression graphs for the *then* and *else* expressions, both giving results of the same type. The *if* operator has numeric *then* and *else* expressions, the *ifs* has string values for its *then* and *else* expressions, and the *implies* operator has logical expressions for its *then* and *else* expressions.

As an example of an *if-then-else* expression, Table 12 shows the objective segment in `silly.nl` corresponding to objective `zip` of Figure 1.

4 Segments

Following the header in a `.nl` file are various segments, introduced by one of the distinguishing key letters shown in Table 13. The segments only appear if nonempty and generally appear in the order shown above, though other orders are possible, subject only to the restrictions noted in Table 13. Some segments (the ones with lower-case key letters: d, x, r, b, k) appear at most once. The others appear as often as necessary. For example, if the solver sees three algebraic constraints, then there will be three C segments. G and J lines supply linear terms for the corresponding objectives and algebraic constraints.

The V, C, L, and O segments contain expression graphs in Polish prefix notation, as discussed in §3. Explanations of each kind of segment follow, in the order shown in Table 13.

F segments provide information about imported functions and consist of "F" followed by three integers, say i, j, and k, and the name of the function (an unquoted string), where i is the function number (0 for the first), j is 1 if string arguments are allowed and is 0 otherwise, and k indicates the number of arguments: if $k < 0$, then the function has at least $-(k+1)$ arguments, and if $k \geq 0$, then the function has exactly k arguments. For example, the segment for function `myfunc` in Figure 1 consists of the single line

```
F0 1 -1 myfunc
```

S segments convey declared and implicit suffixes. Implicit suffixes include values AMPL uses for linearizing nonconvex piecewise-linear terms (which requires a solver that handles integer variables) and `.sstatus` values for conveying basis information. The latter are usually only nonzero after a preliminary "solve" by a simplex-based linear-programming solver or by a nonlinear solver that similarly uses basis information, such as MINOS. Declared suffixes convey auxiliary information to some solvers; for example, some integer-programming solvers get branching priorities from `.priority` suffix values. Other solvers make no use of suffix values; there is no need to provide suffix values that will not be used. In general, solvers look only for suffixes they care about and ignore any others that might be present.

Following the "S" that introduces a suffix segment are two integers, k and n, and the name of the suffix (a string value containing no white space, followed by a newline). The first two bits of k indicate the kind of entity to which the suffix applies, as indicated in Table 14 (in which the first two bits of k are called Kind; in C, these bits are computed as "`Kind = k & 3`"). The "4" bit of k indicates whether the suffix is real (i.e., double) valued or integer valued: $(k\&4) \neq 0 \rightarrow$ real valued. Only nonzero suffix values are transmitted; n is the number of such values. Following the "S" line are n lines of the form

$$i \; v_i$$

where i is the offset of suffix value v_i. (Thus a suffix value on the first variable or constraint would have $i = 0$.)

As an example, Table 15 shows the S segment for suffix `zork` of Figure 1.

V segments provide definitions of defined variables, which amount to named common subexpressions. Defined variables that appear in more than one constraint or objective come first, before any C, L or O segments. V segments for defined variables that appear only in a single constraint or objective come just before the C, L or O segment for the constraint or objective.

Three integers follow the "V" that introduces a V segment: i, j, k. The first, i, is the variable number for this defined variable: in subsequent expression graphs vi will denote the value of this defined variable. The second integer, j, is the number of linear terms that immediately follow, and the third, k, is 0 if vi appears in more than one constraint or objective; if it only appears in constraint m (with $m = 0$ for the first constraint), then $k = m + 1$; and if vi only appears in objective m (0 for the first), then k = `n_con` + `n_lcon` + m

(see Table 2). Immediately after the "Vi j k" line come j lines of the form

$$p_\ell \; c_\ell$$

where p_ℓ is an integer with $0 \le I_\ell <$ `n_var` and c_ℓ is a floating-point number (such as 1.23 or -4.56e7). These j lines represent $\sum_{\ell=0}^{j-1} c_\ell \cdot v_{p_\ell}$. The value v_i of `v`i is this sum plus the value of the expression in the following expression graph (§3).

To illustrate V segments, Table 16 shows the V segments for defined variable `t[2]` of Figure 1, which AMPL has split into two parts, the purely nonlinear part and the complete `t[2]` (denoted `v9` and `V10`, respectively, in `silly.nl`). AMPL does this for efficiency, to allow combining linear contributions where appropriate, but such splitting is not required.

The body of algebraic constraint i, $0 \le i <$ `n_con`, is introduced by a line of the form `C`i, which is followed by an expression graph for the nonlinear part of the body. The linear part comes later, in the J segment for this constraint, and the complete constraint body is the linear part plus the nonlinear part. Similarly, logical constraint i, $0 \le i <$ `n_lcon`, is introduced by a line of the form `L`i, followed by an expression graph for the complete logical constraint. And objective i, $0 \le i <$ `n_obj`, is introduced by a line of the form "`O`i σ", followed by an expression graph for the nonlinear part of the objective, with $\sigma = 0$ if the objective is to be minimized and $\sigma = 1$ if it is to be maximized. In analogy with constraints, the linear part of the objective is conveyed in the subsequent G segment corresponding to the objective, and the total objective value is the linear part plus the nonlinear part. Table 12 illustrates the O segment for objective `zip` of Figure 1.

Initial values for the dual and primal variables are conveyed in d and x segments. The number of entries in each segment, say m, immediately follows the introductory "d" or "x". Then come m (offset, value) pairs, where the integer offset is zero for the first constraint or variable.

Ranges for the constraints are conveyed in an r segment. Following the "r" are `n_con` lines, each consisting of an integer followed by zero, one, or two floating-point numbers, or, for complementarity constraints, two integers, as indicated in Table 17. For a complementarity constraint to hold, if v_{i-1} is at its lower bound, then $body \ge 0$; if v_{i-1} is at its upper bound, then $body \le 0$; and if v_{i-1} is strictly between its bounds, then $body = 0$. The integer k in a complementarity constraint line indicates which bounds on v_{i-1} are finite: 1 and 3 imply a finite lower bound; 2 and 3 imply a finite upper bound; 0 (which should not occur) would imply no finite bounds, i.e., $body = 0$ must always hold.

Table 18 is an example r segment, corresponding as usual to Figure 1.

Bounds on the variables are conveyed in a b section, which has `n_var` lines after the initial "b". These lines are as in Table 17, with the variable in place of "*body*" and with no complementarity constraints (i.e., lines starting with 5).

It should be clear that constraints and objectives are similar, except that constraints have ranges, i.e., upper and lower bounds, associated with them. Another distinction is that `.nl` files contain extra information about the sparsity of constraints, to facilitate computing a Jacobian matrix stored columnwise, since many nonlinear solvers want to see Jacobians stored this way. The k segment contains this extra information. Following "k" is the integer `n_var` $- 1$ followed by that many integers, the cumulative sums of the numbers of nonzeros in the first `n_var` $- 1$ columns of the Jacobian matrix.

To illustrate k segments, Table 19 shows the sparsity of the Jacobian matrix corresponding to `silly.amp` (Figure 1), and Table 20 shows the corresponding k section.

For each constraint or objective, a corresponding J or G segment indicates which variables appear in the constraint or objective and supplies coefficients for the linear part of the constraint's body or for the objective. The initial J or G is followed by two integers, i and k, where i indicates the constraint or objective (0 for the first) and k the number of variables on which it depends. Following these integers are k (offset, coefficient) pairs (j, c) indicating variable j and linear coefficient c.

As a final example, Table 21 shows the J segments for the first two constraints (`c1` and `c2`) in Figure 1.

5 Making Your Own Examples

You may find it helpful to use AMPL to write your own sample `.nl` files. Student versions of AMPL suffice for this purpose and are available from platform-specific subdirectories of *netlib*'s `ampl/student` directory [1].

References

[1] Netlib's `ampl/student` directory. `http://www.netlib.org/ampl/student` or `http://netlib.bell-labs.com/netlib/ampl/student`.

[2] R. Fourer, D. M. Gay, and B. W. Kernighan. A modeling language for mathematical programming. *Management Science*, 36(5):519–554, 1990. (The URL is for the longer technical report cited in the *Management Science* paper.).

[3] Robert Fourer and David M. Gay. Extending an algebraic modeling language to support constraint programming. *INFORMS Journal on Computing*, 14(4):322–344, 2002.

[4] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press/Brooks/Cole Publishing Co., second edition, 2003.

[5] David M. Gay. Hooking your solver to AMPL. Numerical Analysis Manuscript No. 93-10, AT&T Bell Laboratories, Murray Hill, NJ, 1993, revised 1997. `http://www.ampl.com/REFS/hooking2.ps.gz`.

[6] AMPL/solver interface library source file `op_type.hd`. Available as `http://www.netlib.org/ampl/solvers/op\_type.hd` or `http://netlib.bell-labs.com/netlib/ampl/solvers/op\_type.hd.gz`.

```
O0 1        #zip
o35         # if
o28         # >=
v10         #t[2]
n0
o16         #-
o5          #^
v10         #t[2]
n3
o16         #-
o5          #^
v10         #t[2]
n2
```

Table 12: objective segment in `silly.nl` for objective `zip`

.

Key letter	Description
F	imported function description
S	suffix values
V	defined variable definition (must precede V,C,L,O segments where used)
C	algebraic constraint body
L	logical constraint expression
O	objective function
d	dual initial guess
x	primal initial guess
r	bounds on algebraic constraint bodies ("ranges")
b	bounds on variable
k	Jacobian column counts (must precede all J segments)
J	Jacobian sparsity, linear terms
G	Gradient sparsity, linear terms

Table 13: segment types.

Kind	Entity
0	variables
1	constraints
2	objectives
3	problem

Table 14: kinds of suffixes.

```
S0 8 zork
0 2
1 6
2 7
3 8
4 9
5 3
6 5
8 4
```

Table 15: S segment for suffix `zork` of `silly.amp`

```
V9 0 0     #nl(t[2])
o5         #^
v0         #x[2]
n2
V10 2 0     #t[2]
3 10
4 11
o0         # +
v9         #nl(t[2])
n1
```

Table 16: V segments for `t[2]` of `silly.amp`

Line	Interpretation
0 ℓ u	$\ell \leq \mathit{body} \leq u$
1 u	$\mathit{body} \leq u$
2 ℓ	$\ell \leq \mathit{body}$
3	no constraints on *body*
4 c	$\mathit{body} = c$
5 k i	*body* complements variable v_{i-1}

Table 17: Lines in r segments.

```
r         #4 ranges (rhs's)
1 3
2 3
0 4.3 15.5
4 1
```

Table 18: r segment from `silly.nl`

v0	v1	v2	v3	v4	v5	v6	v7	v8
x			x	x	x			
	x					x		
x	x		x	x	x	x		
	x	x			x	x		x

Table 19: Jacobian matrix sparsity for `silly.amp`

```
k8    #intermediate Jacobian column lengths
2
5
6
8
10
13
16
16
```

Table 20: k section from `silly.nl`

```
J0 4
0 0
3 10
4 11
5 0
J1 2
1 0
6 1
```

Table 21: J segments for `c1` and `c2` in `silly.nl`