Formalized Linear Programming

Abstract

Correct and efficient linear programming, using the simplex algorithm, is of high importance not only in the context of operational research but also in programming language analysis. While existing solvers are robust, the lack of formally verified implementations prevents their use in computational proofs and rare corner-case bugs make the use of linear programming for automatic program generation difficult. Today, existing implementations are either written in low-level high-performance languages (e.g., C++), which are hard to verify, or focus on the verification aspects but are not intended to be executable in practice. We address this problem by implementing an executable and verified simplex solver in the Lean programming language. We ensure both ease of verification and reasonable execution performance by uses Bland's rule to select a pivot and by stating our pivot function without large and costly matrix multiplications as they are often used in textbook formalizations. Our research provides core foundations for verified linear programming and demonstrates the practicality of Lean as a language for fast verified constraint programming.

1 Introduction

Describe motivation of project. The aim is to provide a practically computable implementation of the simplex algorithm (Coq version did not aim to do this). It differs in that we use a more complex tableau data type (dead columns and unrestricted rows). We also use a different pivot rule (Bland's Rule - lexicographic rule seems to be an ambiguous name).

Description of why Lean is a good language to do this in.

2 Linear Programming

The standard simplex algorithm. No reference to Lean code

The simplex algorithm aims to optimize the value of an objective function within the constraints of a set of affine inequalities. It can be used to solve problems of the following form.

$$\begin{aligned} \text{Find } x \text{ that maximizes} \quad & c^T x \\ \text{subject to} \quad & Ax + b \geq 0 \\ \text{and} \quad & x \geq 0 \end{aligned} \tag{1}$$

Here, $x \in \mathbb{Q}^n$, $c \in \mathbb{Q}^n$, $b \in \mathbb{Q}^m$, and $A \in \mathbb{Q}^{m \times n}$. This can be written in an equivalent form The inclusion of z in the objective function is slightly unusual because obviously maximizing $c^T x$ is

the same as maximizing $c^T x + z$. However z is a cell in the tableau, it is the objective function value of the sample solution, and the explanation of the tableau is slightly easier with it

$$\begin{aligned} \text{Find } x_r \text{ and } x_c \text{ that maximize} \quad & c^T x_c + z \\ \text{subject to} \quad & x_r = Ax_c + b \\ \text{and} \quad & x_r \geq 0, x_c \geq 0 \end{aligned} \tag{2}$$

with $x_r \in \mathbb{Q}^m$, $x_c \in \mathbb{Q}^n$ and $z \in \mathbb{Q}$

The affine function $c^T + z$ is known as the *objective function.*

In the tableau representation of this problem, the vectors x_r and x_c are viewed as minors of a single variables x. The subset of $\mathbb{Q}^{m+n}$, $\{x : x_c = Ax_r + b \wedge x_c \geq 0 \wedge x_r \geq 0\}$ is stored as part of the simplex algorithm. The tableau consists of a matrix A of coefficients, a column vector b, a reduced cost vector c, and the constant part of the objective function z. Additionally two vectors of indices r and c defining the minors x_r and x_c are also stored.

$$\begin{array}{rcccccccccc} x_{r_1} & = & A_{11}x_{c_1} & + & A_{12}x_{c_2} & + & \ldots & + & A_{1n}x_{c_n} & + & b_1 \\ x_{r_2} & = & A_{21}x_{c_1} & + & A_{22}x_{c_2} & + & \ldots & + & A_{2n}x_{c_n} & + & b_2 \\ \vdots & & \vdots & & \vdots & & & & \vdots & \vdots & \\ x_{r_m} & = & A_{m1}x_{c_1} & + & A_{m2}x_{c_2} & + & \ldots & + & A_{mn}x_{c_n} & + & b_n \\ c^T x_c + z & = & c_1 x_{c_1} & + & c_2 x_{c_2} & + & \ldots & + & c_n x_{c_n} & + & z \end{array} \tag{3}$$

In this tableau representation the x_r are known as the *row* variables, and the x_c are the *column* variables. In a tableau the row variables are expressed as affine functions in terms of the column variables.

The simplex algorithm iteratively performs the *pivot* operation on a tableau. The pivot operation takes a row and a column index as an argument, and swaps their positions in the tableau. If x_{r_i} and x_{c_j} are pivoted, then in the new tableau x_{r_i} becomes a row variable and x_{c_j} becomes a column variable. The variables $(x_{r_1}, x_{r_2}, \ldots, x_{c_j}, \ldots x_{r_m})$ are expressed as affine functions of $(x_{c_1}, x_{c_2}, \ldots, x_{r_i}, \ldots, x_{c_n})$. The matrix A, and the vectors c and b are updated to A', c' and b', such that the tableau represents the same subset of $\mathbb{Q}^{m+n}$, and the same optimization problem. Additionally the vectors r and c are updated to vectors $r' := (r_1, r_2, \ldots c_j, \ldots, r_m)$ and $c' := (c_1, c_2 \ldots r_i, \ldots, c_n)$. Oops, two things are called c Preserving the same polyhedron is possible if $A_{ij} \neq 0$.

$$\begin{array}{rcccccccccc} x_{r_1} & = & A'_{11}x_{c_1} & + & A'_{12}x_{c_2} & + & \ldots & + & A'_{1j}x_{r_i} & + & \ldots & + & A'_{1n}x_{c_n} & + & b'_1 \\ x_{r_2} & = & A'_{21}x_{c_1} & + & A'_{22}x_{c_2} & + & \ldots & + & A'_{2j}x_{r_i} & + & \ldots & + & A'_{2n}x_{c_n} & + & b'_2 \\ \vdots & & \vdots & & \vdots & & & & \vdots & & & & \vdots & & \vdots \\ x_{c_j} & = & A'_{i1}x_{c_1} & + & A'_{i2}x_{c_2} & + & \ldots & + & A'_{ij}x_{r_i} & + & \ldots & + & A'_{in}x_{c_n} & + & b'_i \\ \vdots & & \vdots & & \vdots & & & & \vdots & & & & \vdots & & \vdots \\ x_{r_m} & = & A'_{m1}x_{c_1} & + & A'_{m2}x_{c_2} & + & \ldots & + & A'_{mj}x_{r_i} & + & \ldots & + & A'_{mn}x_{c_n} & + & b'_n \\ c^T x + z & = & c'_1 x_{c_1} & + & c'_2 x_{c_2} & + & \ldots & + & c'_j x_{r_i} & + & \ldots & + & c'_n x_{c_n} & + & z' \end{array} \tag{4}$$

The expressions for the updated matrix A', and the updated vectors c', b' are given by the

following formulas [4].

$$A'_{i'j'} = \begin{cases} A_{ij}^{-1} & \text{if } i' = i \text{ and } j' = j \\ -A_{ij'}/A_{ij} & \text{if } i' = i \text{ and } j' \neq j \\ A_{i'j}/A_{ij} & \text{if } i' \neq i \text{ and } j' = j \\ A_{i'j'} - A_{i'j}A_{ij'}/A_{ij} & \text{if } i' \neq i \text{ and } j' \neq j \end{cases} \tag{5}$$

$$b'_{i'} = \begin{cases} -b_i/A_{ij} & \text{if } i' = i \\ b_{i'} - A_{i'j}b_i/A_{ij} & \text{if } i' \neq i \end{cases} \tag{6}$$

$$c'_{j'} = \begin{cases} c_j/A_{ij} & \text{if } j' = j \\ c_{j'} - c_j A_{ij'}/A_{ij} & \text{if } j' \neq j \end{cases} \tag{7}$$

$$z' = z - c_j b_i/A_{ij} \tag{8}$$

Note that b, c and z are updated as though they are a continuation of A; b is updated in the same way as a column of A, and c is updated in the same way as a row of A. This is quite important in the understanding of our version of the simplex. In our version we just optimize a variable x_i, rather than having an objective function, but this is actually the exact same thing.

Given any assignment of values to the column variables of a tableau, there is a unique x satisfying $x_r = Ax_c + b$ The *sample solution* of a tableau is the solution of $x_r = Ax_c + b$ found by setting x_c to zero. For this solution $x_r = b$. Similarly, the objective function must have value z in the *sample solution*. A tableau is *feasible* if the sample solution also satisfies $x \geq 0$. This is equivalent to saying $b \geq 0$.

Given a feasible tableau, the simplex algorithm iteratively pivots whilst maintaining or increasing the objective function value of the sample solution z, as well as maintaining feasibility of the tableau. The chosen pivot row and pivot column i and j of a tableau always satisfy the following properties.

$$\begin{aligned} c_j &> 0 \\ A_{ij} &< 0 \\ \forall i', A_{i'j} < 0 \implies |b_i/A_{ij}| &\leq \left|b_{i'}/A_{i'j}\right| \end{aligned} \tag{9}$$

If the condition $A_{ij} < 0$ is met by some index i, then the condition $\forall i', A_{i'j} < 0 \implies |b_i/A_{ij}| \leq \left|b_{i'}/A_{i'j}\right|$, must also be met, since every finite set contains its greatest lower bound.

If the pivot row and pivot column satisfy these properties then using the expressions in equations (6) and (8), it can be shown the pivoted tableau is both feasible and that z has either increased or stayed the same.

The choice of pivot row and column satisfying (9) is non-unique, and must be chosen carefully in order to guarantee termination. Different implementations of the simplex algorithm use different pivot rules to guarantee termination. If z was strictly increasing it would straightforward to prove the algorithm terminates, since there are only finitely many partitions of the variables into row and column variables, and the simplex algorithm cannot repeat a partition if z is strictly increasing. However when $b_i = 0$ in the pivot row i, z does not increase, and ensuring termination is more difficult. Our implementation uses Bland's Rule to ensure termination. [1].

If there is no column j satisfying $c_j > 0$, then the sample solution must maximize $c^T x_c$. This is because $c^T \leq 0$ and $x_c \geq 0$, so $c^T x_c \leq 0$, and the maximum value of this is given by setting x_c to zero.

Explanation below could be better

If there is a column j satisfying $c_j > 0$, but no row i satisfying $A_{ij} < 0$ then the objective function must be unbounded. The variable in column j is unbounded; for any $k > 0$, the solution found by setting the variable in column j to k and all other column variables to zero does not break the nonnegativity constraints of the row variables. The expression for the variable in row i is $A_{ij}k + b_i$, which is nonnegative. The objective function is $c_j k + z$, which is unbounded since k is unbounded.

The simplex algorithm terminates when there is no pivot row and column satisfying the conditions in (9). When it terminates it has always either detected unboundedness or found an optimal solution.

3 Description of our version of the linear programming problem

Our version, dead columns, unrestricted variables, stopping early. Also very little reference to Lean code.

The implementation in Lean follows the implemenentation in the Simplify prover [3]. This implementation represents a polyhedron in a slightly different way. There are two additional sets stored, a set of *restricted* variables and a set of *dead* columns.

The optimization problem for this implementation is given by

$$\begin{aligned} \text{Find } x \text{ that maximizes} &\quad x_{r_k} \\ \text{such that} &\quad x_r = Ax_c + b \\ \text{and} &\quad \forall i \in \mathit{restricted}, x_i \geq 0 \\ \text{and} &\quad \forall j \in \mathit{dead}, x_{c_j} = 0 \end{aligned} \tag{10}$$

The inclusion of the two sets *dead* and *restricted* changes the conditions that any pivot row and column must meet. If it is trying to optimize the variable in row k then a pivot row i and pivot column j must satisfy the following criteria.

$$\begin{aligned} A_{kj} &\neq 0 \\ c_j \in \mathit{restricted} &\implies A_{kj} > 0 \\ j &\notin \mathit{dead} \\ i &\neq k \\ r_i &\in \mathit{restricted} \\ A_{ij}/A_{kj} &< 0 \\ \forall i', r_{i'} \in \mathit{restricted} \wedge A_{i'j}/A_{kj} < 0 &\implies |b_i/A_{ij}| \leq \left|b_{i'}/A_{i'j}\right| \end{aligned} \tag{11}$$

Once the pivot row and column is chosen the tableau is updated in the same way as in (5) and (6).

Like the classical simplex, if there is no pivot column then the sample solution is optimal, and if there is no pivot row the objective variable is unbounded. The proofs of these are minor adaptations of the proofs for the classical simplex.

4 Lean implementation

Only describe Lean code where I had to do something in a way that was not the obvious way. The termination proof is interesting since the majority of the length of the proof is about stuff that was not mentioned on paper. Also 'pequiv' made a few things a little easier and is not obvious. Also state correctness theorems

4.1 Correctness statements

The tableau data structure is implemented in Lean as a record of the matrix in the optimization problem, a partition of row and column variables, a constant column, a set of restricted variables, and a set of dead columns.

```
structure tableau (m n : ℕ) extends partition m n :=
(to_matrix  : matrix (fin m) (fin n) ℚ)
(const      : cvec m)
(restricted : finset (fin (m + n)))
(dead       : finset (fin n))
```

Given a tableau it is possible to define the polyhedron corresponding to the tableau. First the `flat` is defined, the affine subset of $\mathbb{Q}^{m+n}$ that satisfies the affine equalities.

```
def flat : set (cvec (m + n)) :=
{ x | T.to_partition.rowp.to_matrix · x =
  T.to_matrix · T.to_partition.colp.to_matrix · x + T.const }
```

`T.to_partition.rowp.to_matrix` is an $m \times (m+n)$ matrix, which corresponds to the matrix R in (12). Similarly `T.to_partition.colp.to_matrix` is the matrix C in (12).

The other relevant sets are the `res_set`, the intersection of the flat and the set such that all restricted variables are nonnegative, and the `dead_set`, the intersection of the flat and the set such that the variables assigned to all dead columns are equal to zero. Finally the main object of study is the `sol_set`; the intersection of the `res_set` and the `dead_set`. In the definition of `sol_set`, `T.to_partition.colg j` returns the variable assigned to the column `j` in `T`.

```
def res_set : set (cvec (m + n)) := flat T ∩ { x | ∀ i, i ∈ T.restricted → 0 ≤ x i 0 }

def dead_set : set (cvec (m + n)) :=
flat T ∩ { x | ∀ j, j ∈ T.dead → x (T.to_partition.colg j) 0 = 0 }

def sol_set : set (cvec (m + n)) :=
res_set T ∩ { x | ∀ j, j ∈ T.dead → x (T.to_partition.colg j) 0 = 0 }
```

Using these two sets we can also define the predicates `is_optimal` and `is_unbounded_above`

```
def is_optimal (x : cvec (m + n)) (i : fin (m + n)) : Prop :=
x ∈ T.sol_set ∧ ∀ y, y ∈ sol_set T → y i 0 ≤ x i 0

def is_unbounded_above (i : fin (m + n)) : Prop :=
∀ q : ℚ, ∃ x : cvec (m + n), x ∈ sol_set T ∧ q ≤ x i 0
```

The type `cvec (m + n)` is notation for $m + n \times 1$ matrices. `x i 0` is the notation for the ith element of the vector `x`.

The Lean simplex algorithm takes as arguments a tableau, the row that should be optimized and a boolean function, that gives the user the option of terminating the algorithm early. This can be used for example to determine whether the maximum value of a variable is positive, without actually computing the maximum. The simplex algorithm itself returns both a tableau and the reason for termination of the algorithm, either `unbounded`, `optimal`, or `while`. The

```
def simplex (w : tableau m n → bool) (obj : fin m) : Π (T : tableau m n) (hT : feasible T)
    ,
  tableau m n × termination n
```

`termination n` is an inductive type with three constructors. In the case that the variable is unbounded, the piivot column that was chosen before the algorithm failed to find a pivot row is also returned.

```
inductive termination (n : ℕ) : Type
| while {}          : termination
| unbounded (c : fin n) : termination
| optimal {}        : termination
```

The full correctness statement is actually quite complicated, but there are simpler versions that state correctness of the most important parts of the algorithm (it returns the correct one out of optimal unbounded or while)If the simplex algorithm is correct, then the tableau returned should be a feasible tableau, representing the same polyhedron as the starting tableau. Most importantly we also need to prove that it returns the correct condition out of `while`, `unbounded`, and `optimal`.

```
@[simp] lemma sol_set_simplex (T : tableau m n) (hT : feasible T) (w : tableau m n →
    bool)
  (obj : fin m) : (T.simplex w obj hT).1.sol_set = T.sol_set

lemma termination_eq_while_iff {T : tableau m n} {hT : feasible T} {w : tableau m n →
    bool}
  {obj : fin m} : (T.simplex w obj hT).2 = while ↔ w (T.simplex w obj hT).1 = ff

lemma termination_eq_optimal_iff {T : tableau m n} {hT : feasible T}
  {w : tableau m n → bool} {obj : fin m} : (T.simplex w obj hT).2 = optimal ↔
  w (T.simplex w obj hT).1 = tt ∧
  is_optimal T ((T.simplex w obj hT).1.of_col 0) (T.to_partition.rowg obj)

lemma termination_eq_unbounded_iff {T : tableau m n} {hT : feasible T}
  {w : tableau m n → bool} {obj : fin m} {c : fin n}: (T.simplex w obj hT).2 = unbounded
    c ↔
  w (T.simplex w obj hT).1 = tt ∧ is_unbounded_above T (T.to_partition.rowg obj)
  ∧ c ∈ pivot_col (T.simplex w obj hT).1 obj
```

4.2 pequiv

I can't really claim confidently that this was any better than any other way of doing this. My initial instinct when starting the project was that proving properties of matrices defined

using `if ... then ... else ...` would be extremely difficult, however the pivot defined like this was easier to use, and I think the proofs mentioned in this section would be shorter overall with `if ... then ... else ...` as well, because of the lines spent proving the properties of the matrices R and C mentioned below. The simplex algorithm requires a lot of reasoning about minors of matrices or vectors. Taking a minor of a matrix is a linear map, and can be expressed as a matrix multiplication. Within the simplex algorithm there are two matrix minors that are often taken, the minor of a vector within a polyhedron corresponding to the column variables, and the minor for the row variables.

Suppose R^T is the matrix mapping x to x_r, and C^T is the matrix mapping x to x_c. Then R and C have the following properties.

$$\begin{aligned} R^T R &= 1 \\ C^T C &= 1 \\ R^T C &= 0 \\ C^T R &= 0 \\ RR^T + CC^T &= 1 \end{aligned} \tag{12}$$

These matrices are used in the definitions of the polyhedron corresponding to a tableau, and also the definition of the function `of_col` can be made short by always expressing minors using matrix multiplication. Given x_c, and a tableau, the function $ofcol$ computes x, satisfying $x_r = Ax_c + b$. The definition is given by

$$ofcol(x_c) := Cx_c + R(Ax_c + b) \tag{13}$$

In this case we are not using C and R to take minors, but rather using C to assign values to the column variables, and leave zeros elsewhere, and R to assign values to the row variables and zeros elsewhere. The basic properties of this function can be easily proved using the identities in (12).

$$\begin{aligned} C^T ofcol(x_c) &= x_c \\ R^T ofcol(x_c) &= Ax_c + b \\ R^T ofcol(x_c) &= AC^T ofcol(x_c) + b \end{aligned} \tag{14}$$

Do I want to go into as much detail as below

The definition of the matrices R and C in Lean uses the concept of a partial equivalence. A partial equivalence between two sets X and Y is a bijection between a subset of X and a subset of Y. In the same way that there is a group homomorphism from the set of permutations of $(1, \ldots, n)$ into the group of invertible $n \times n$ matrices, there is a functor from the category of partial equivalences on finite sets into the category of matrices.

The partial equivalence corresponding to R is given by the vector r, mapping i to r_i. This is an injective function and therefore a bijection with its own image. The matrix R is given by

$$R_{ij} = \begin{cases} 1 & \text{if } j = r_i \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

4.3 Bland's Rule

How much detail to go into in this proof? This was one of the more challenging parts of the formalization, however, the proof is actually not very nicely written at the moment, for example

the definition of fickle is not quite the same as in Chvatal. I used a weaker notion that seemed good enough,but then one variable that should be fickle ended up not being fickle, and I had to get round this in a slightly messy way.

The simplex implemented in Lean uses Bland's rule to ensure termination. If the pivot column is restricted then we choose the column owned by a variable with the smallest index out of the columns that satisfy the condition specified in [refer to earlier section]. We do the same for the rows that satisfy the specified condition.

In order to prove that this rule will terminate, it suffices to prove that the simplex does not repeat a tableau. This is because there are only finitely many tableaux that can be visited by the simplex algorithm; at most one for each of the finitely many partitions of the variables. Supposing that the simplex algorithm does repeat then there is a finite set of "fickle" variables that are pivoted during a cycle. By always choosing the variable with the smallest index, we know that the largest of the fickle variables was the unique fickle column variable satisfying [refer to earlier section] in some tableau in the cycle, and the unique fickle row variable in some other tableau in this cycle. It is possible to derive a contradiction from this, though the proof is omitted here [2].

In order to provide a proof of termination of the simplex algorithm in Lean, it is necessary to give a relation, a proof of well foundedness of this relation, and a proof that the sequence of tableaux accessed by the simplex algorithm is decreasing according to this relation. For this proof of termination, there is no natural choice of relation, so the relation is just defined using the pivot rule. Given tableaux `T'` and `T`, `rel obj T' T` is a relation saying that if the simplex algorithm visits `T`, then at some point after it will visit `T'`. It is defined inductively.

```
inductive rel : tableau m n → tableau m n → Prop
| pivot : ∀ {T}, feasible T → ∀ {i j}, pivot_col T obj = some j →
  pivot_row T obj j = some i → rel (T.pivot i j) T
| trans_pivot : ∀ {T₁ T₂ i j}, rel T₁ T₂ → pivot_col T₁ obj = some j →
  pivot_row T₁ obj j = some i → rel (T₁.pivot i j) T₂
```

Two tableau `T` and `T'` are related if either `T' = T.pivot i j` where `i` and `j` are the pivot row and column selected according to Bland's rule, or if there exists another tableau T_1 such that `rel obj` T_1 `T` and `T' =` T_1`.pivot i j`.

Various properties of the tableau then have to be proven by induction on this relation. The majority of the length of the termination proof was taken up by proving properties of related tableaux and tableaux within a cycle.

5 Refined algorithm

Say that we refined it and it is vaguely fast. Do I go into detail about how we did it. Probably quite a short section.

The simplex algorithm was originally written on an extremely slow implementation of matrices. The implementation in the Lean maths library [cite mathlib]is as the type of binary functions from two finite sets into a ring, in our case $\mathbb{Q}$. This implementation is convenient for writing proofs, but very slow to compute. In order to have a practical executable version of the simplex algorithm, it was necessary to transfer the algorithm and proofs onto a faster implementation of matrices using arrays. Ideally this would be done using a refinement framework brief explanation

of what that means here. Would it be possible to generalize proof onto a type class rather than a concrete type using this framework?. This is not currently available in Lean, so it was instead done in an ad hoc way, using a tableau type class. The proof of the simplex algorithm can be generalized onto any type for which there is an instance of the `is_tableau` type class.

```
class is_tableau (X : ℕ → ℕ → Type) : Type :=
(to_tableau {m n : ℕ} : X m n → tableau m n)
(pivot {m n : ℕ} : X m n → fin m → fin n → X m n)
(pivot_col {m n : ℕ} (T : X m n) (obj : fin m) : option (fin n))
(pivot_row {m n : ℕ} (T : X m n) (obj : fin m) : fin n → option (fin m))
(to_tableau_pivot {m n : ℕ} (T : X m n) (i : fin m) (j : fin n) :
  to_tableau (pivot T i j) = (to_tableau T).pivot i j)
(to_tableau_pivot_col {m n : ℕ} (T : X m n) :
  pivot_col T = (to_tableau T).pivot_col)
(to_tableau_pivot_row {m n : ℕ} (T : X m n) :
  pivot_row T = (to_tableau T).pivot_row)
```

By using the `is_tableau` type class, implementing and proving correctness of a fast implementation of the simplex algorithm, is reduced to defining an instance of this class. To define an instance of this class, it is necessary to provide a function `to_tableau` from a fast tableau type into `tableau m n`, and three functions `pivot_col`, `pivot_row`, and `pivot`. The only proofs about these functions that must be provided are that they agree with the functions on `tableau m n`. This sentence needs to be changed

6 Conclusion

Follow up work. Difficulties I came across. It would have been nice to have a generic theory of linear programming. Some of what I wrote could be applied to other implementations of simplex, but not much.

References

[1] New finite pivoting rules for the simplex method. *Math. Oper. Res.*, 2(2):103–107, May 1977.

[2] Vašek Chvátal. *Linear programming*. A series of books in the mathematical science. Freeman, New York, 1983.

[3] David Detlefs, Greg Nelson, and James Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52, 09 2003.

[4] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.