# SpMV Sections

**ACM Reference Format:**
. 2019. SpMV Sections. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 SPARSE MATRIX-VECTOR MULTIPLICATION

Sparse linear algebra libraries and the methods they employ are vitally important in the domain of scientific computing (TODO: why?). Of particular interest is the sparse matrix-vector product, which solves $y = \boldsymbol{A}x$ where $\boldsymbol{A}$ is a sparse matrix and the vectors $y$ and $x$ are dense. Because SpMV usage is often highly repetitive within, e.g., iterative solvers, performance improvements via novel algorithms and the exploitation of modern hardware are areas of ongoing research (TODO: citations).

...why is SpMV difficult? Things like array indirection, varying sparsity patterns, it's memory bound...

As modern hardware introduces higher levels of parallelism and SpMV algorithms become more complex, the likelihood of introducing subtle bugs becomes decidedly higher. In this section we present a method for reasoning about the structural complexities and verifying the correctness of SpMV algorithms. We first model an abstract matrix-vector multiplication and subsequently demonstrate that a CSR SpMV algorithm is a valid functional refinement.

### 1.1 Abstract Matrix-Vector Multiplication

In order to demonstrate that an SpMV algorithm is correct, we must first create a model of matrix-vector multiplication to act as the arbiter of correctness. The result of the matrix-vector multiplication $y = \boldsymbol{A}x$ is a densely populated vector, $y$, in which each value is the dot product of a single row of the matrix $A$ with the vector $x$. A dot product is simply a sum of products in which each product is of two values located at the same index within two vectors, as shown in Figure 1b.

Thus far, our matrix and sparse matrix models are capable of representing the structure of the matrix and vectors, as well as the values contained in $\boldsymbol{A}$ and $x$. However, because we wish to model the *structural* behavior of the algorithm, the resulting vector, $y$, cannot simply contain abstract values. Given a solution generated by a matrix-vector multiplication, we need to be able to verify that the *composition* of a value is correct; that each value present in each dot product originates from the correct location. This is enabled by using the `SumProd` signature, introduced in Section 1.1.1, along

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \times \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{A_0 \cdot x_0} \\ A_1 \cdot x_1 \\ A_2 \cdot x_2 \end{Bmatrix}$$

**(a)**

$$\boldsymbol{A_0 \cdot x_0} = A_{00} * x_0 + A_{01} * x_1 + A_{02} * x_2$$

**(b)**

| | | |
|---|---|---|
| 0 | $A_{00}$ | $x_0$ |
| 1 | $A_{01}$ | $x_1$ |
| 2 | $A_{02}$ | $x_2$ |

**(c)**

**Figure 1: (a) a matrix-vector multiplication, (b) the components of the first dot product in (a), and (c) the relational form of the same dot product.**

with the following predicates: (1) to generate dot products, we introduce the `dotProd` predicate in Section 1.1.2, (2) to determine the equivalence of dot products we introduce the `valEqv` predicate in Section 1.1.3, and (3) to perform an abstract matrix-vector multiplication, we introduce the `MVM` predicate in Section 1.1.4.

*1.1.1 Sum of Products Model.* The `SumProd` signature found in Figure 2a represents the result of a dot product operation, a sum of products. Because each dot product evaluates to a single value that must be stored in one of our existing matrix models, the `SumProd` signature extends `Value`. Note that this evaluation is never actually realized, we merely wish to assemble values such that they represent a sum of products. To do so, `SumProd` contains a single field, `values`, that maps unique integers to `Value` pairs. We do not allow `SumProds` to nest, as we only wish to model dot products of numerical values, not complex numerical expressions, and so `SumProd` is removed from the set of values that may be stored by using the expression `(Value-SumProd)`. Using the `values` field, the `SumProd` signature is able to represent a sum of products as a table in which each row contains a product of two values, and the summation is composed of all rows, as shown in Figure 1c. Additionally, the integer value associated with each row indicates the index from which the values in that row originate.

*1.1.2 Dot Product Model.* The `dotProd` predicate found in Figure 2b assembles the dot product of `row` and `x` into the `SumProd` `b`. These three arguments warrant an explanation, as their usage may not be immediately clear.

First, the relation `row` represents a single row from an abstract `Matrix`. Recall that a `Matrix` stores values using an `Int→Int→Value`

```
sig SumProd extends Value {
    values: Int
        → lone (Value-SumProd)
        → (Value-SumProd)
}{
    all i: values.univ.univ | i ≥ 0
}
```

(a)

```
pred dotProd [row: Int→Value, x: Matrix, b: SumProd]
{
    all col: row.univ |
        b.values[col] = row[col]→x.values[col][0]
    all col: Int- row.univ |
        no b.values[col]
}
```

(b)

```
pred valEqv [x, y: Value] {
    (x = y)
    or (x in SumProd and isZero[x] and y = Zero)
    or (y in SumProd and isZero[y] and x = Zero)
    or (isZero[x] and isZero[y])
    or (x in SumProd and
         y in SumProd and
         removeZeros[x] = removeZeros[y])
}
```

(c)

**Figure 2: (a) The `SumProd` signature, representing a summation of products, (b) the `dotProd` predicate, which assembles a dot product, and (c) the `valEqv` predicate, which determines if two values are equivalent.**

relation, in which the first and second integer represent the row and column, respectively, in which the corresponding value falls. Therefore, when we reference all values in a single row, the first integer is removed, leaving us with a relation of the form required by `row`, which is `Int→Value`. In this relation, the integer represents the column in which the corresponding value falls.

Second, the argument `x` must be a `Matrix`, representing the column vector that is being multiplied by the matrix in our matrix-vector multiplication. Note that we do not enforce certain properties of `x`, namely that it contain a single column and the same number of rows as the matrix has columns. It is assumed that these checks are performed in concert with the use of the `dotProd` predicate.

Finally, the argument `b` is a `SumProd`, representing the dot product of `row` and `x`. The set of values contained by this `SumProd` is fully enumerated within the predicate using the two `all` statements. The first one states that for each column index, `col`, in the current row of the matrix, the `col`-th value of the dot product summation is equal to the row value at index `col` times the vector value at (`col`, 0). The second states that any integer that is *not* a column index also does not show up in the set of `SumProd` values.

```
pred MVM [A, x, b: Matrix] {
    ...
    all i: rowInds[A] |
        dotProd[A.values[i], x, b.values[i][0]]
}
```

**Figure 3: The abstract matrix-vector multiplication model.**

*1.1.3 Dot Product Equivalence.* When we perform a matrix-vector multiplication, the result is a column vector in which each value is a single `SumProd`. So, in order to compare the results of two different matrix-vector multiplication algorithms, we need to be able to determine the equivalence of individual `SumProd`s. Additionally, we must be able to determine if a `SumProd` is equal to `Zero`, so as not to preclude an algorithm from skipping the assembly of a dot product should it know that a row of the matrix is empty. Finally, because differing algorithms may be operating on storage structures that contain differing numbers of zeros, we expect the composition of dot products to also contain differing numbers of zeros. However, we do expect all dot products to contain equivalent sets on nonzero values originating from the same locations. Therefore, to determine the equivalence of two values, the `valEqv` predicate, shown in Figure 2c, compares only nonzero values when it encounters a `SumProd` value. Note the use of a few helper functions, namely `isZero` and `removeZeros`. The `isZero` function evaluates to true if the `SumProd` contains no nonzero values, and the `removeZeros` function returns the set of nonzero values contained in a `SumProd`. These and other helper functions can be found in the supplemental repository [1].

*1.1.4 Matrix-Vector Multiplication Model.* Finally, we are able to model a matrix-vector multiplication using the `MVM` predicate shown in Figure 3. This predicate receives three matrices, `A`, `x`, and `b`. For brevity, a number of lines have been removed from the `MVM` predicate. These lines, which can be found in the supplemental repository [1], simply ensure that `x` and `b` each only contain a single column and that the dimensions of all three matrices are able to represent a valid matrix-vector multiplication. Additionally, we enforce that there are no `SumProd`s present in either `A` or `x`. As mentioned previously, we are only interested in modeling matrix-vector multiplications that consist of simple numerical values.

The `all` statement in the `MVM` predicate performs the actual matrix-vector multiplication. It simply states that the value located at row `i` in the solution vector `b` is equal to the dot product of row `i` of the matrix `A` with the vector `x`.

## 1.2 CSR Matrix-Vector Multiplication

In this section we model a CSR matrix-vector multiplication algorithm and demonstrate that it is a valid functional refinement of the matrix-vector multiplication algorithm modeled in Section 1.1.

Recall that the CSR format contains three arrays. The arrays `A` and `JA` are equal in length and contain values and the column indices at which those values appear, respectively. The third array, `IA`, contains integers used to index into `A` and `JA`. Each consecutive pair

```
pred MVM [C: CSR, x, b: Matrix] {
    ...
    all i: rowInds[C] |
        let row = getrow[C, i] |
            dotProd[row, x, b.values[i][0]]
}
```

**(a)**

```
fun getrow [c: CSR, row: Int]: Int→Value {
    let cols = rowcols[c, row],
        vals = rowvals[c, row] | ~cols.vals
}
```

**(b)**

```
assert refines {
    all C: CSR, M, x, mb, cb: Matrix |
        ...
        alpha[C, M] and
        MVM[C, x, cb] and
        MVM[M, x, mb] ⇒ matEqv[cb, mb]
}
```

**(c)**

**Figure 4: (a) The CSR matrix-vector multiplication model, (b) the `getrow` helper function, which extracts the column→value pairs of a row in a CSR matrix, and (c) the predicate used to check for refinement.**

of indices represents a subset of the `A` and `JA` arrays corresponding to a single row of the matrix. The first pair of indices represents the first row, the second pair represents the second row, and so on.

```
1 for $i$ in range($nrows$):
2     for $j$ in range(IA[$i$], IA[$i+1$]):
3         val = A[$j$]
4         col = JA[$j$]
5         b[$i$] = b[$i$] + val * x[col]
```

**Algorithm 1:** The CSR MVM algorithm.

Algorithm 1 performs the matrix-vector multiplication of a CSR matrix with the vector `x`, resulting in the vector `b`. In this algorithm, the inner loop is responsible for iterating over consecutive pairs of indices in the `IA` array. Within this inner loop, the current value and column are extracted from the `A` and `JA` arrays, and the dot-product is accumulated at the appropriate location.

We model Algorithm 1 in the `MVM` predicate found in Figure 4a. This predicate shares much in common with the abstract model, still requiring that matrix and vector dimensions are valid for a matrix-vector multiplication. These lines, just as in the abstract model, have been removed for brevity.

The row loop in the CSR model is modeled in the same way as the row loop in the abstract model, by using an `all` expression. Before the inner loop, however, this model uses the `getrow` helper function to extract the columns and values from the CSR data structure. This helper function, found in Figure 4b, models the array dereferencing found in lines 3 and 4 of Algorithm 1. Additionally, it returns the full set of column→value pairs for a given row. This allows us to reuse the `dotProd` predicate found in Figure 2b. Note that the `getrow` function makes use of two additional helper functions, `rowcols` and `rowvals`, which extract the sequence of columns and values, respectively, for a given column. Both can be found in the supplemental repository [1].

Finally, we check that our algorithm correctly performs a matrix-vector multiplication in the `refines` assertion, found in Figure 4c. First, the lines that have been removed simply enforce the representation invariant for all of the arguments passed to the predicate. Next, two matrix-vector multiplications are assembled: $\boldsymbol{M} * x = m_b$ and $\boldsymbol{C} * x = c_b$. Additionally, the CSR matrix $\boldsymbol{C}$ and abstract matrix $\boldsymbol{M}$ are made equivalent using the abstraction function `alpha`. The assertion states that if $\boldsymbol{M}$ and $\boldsymbol{C}$ represent the same matrix and are multiplied by the same vector, $x$, then their respective solution vectors, $m_b$ and $c_b$ are equivalent.

This assertion has been tested for up to 5×5 matrices, and is found to be valid. Therefore, we can conclude that within scope Algorithm 1 is correct.

## REFERENCES

[1] T. Dyer, "Article github repository," 2019. [Online]. Available: https://github.com/atdyer/alloy-lib