Formal Linear Algebra Methods Environment

Presented by Amir Kamil

Typical Software Development Process

1) Develop algorithm to solve problem.
2) Write code that “implements” the algorithm.
3) Insert assertions, debugging output, etc. to verify correctness of code.
4) Run on lots of test cases.

Problems with Typical Process

1) Develop algorithm to solve problem.
   - Difficult – requires expert in area.
2) Write code that “implements” the algorithm.
3) Insert assertions, debugging output, etc. to verify correctness of code.
4) Run on lots of test cases.

Problems with Typical Process

1) Develop algorithm to solve problem.
2) Write code that “implements” the algorithm.
   - Code looks nothing like the algorithm.
   - Hard to tell if code actually implements the algorithm.
   - Linear algebra code prone to index errors.
3) Insert assertions, debugging output, etc. to verify correctness of code.
4) Run on lots of test cases.

Problems with Typical Process

1) Develop algorithm to solve problem.
2) Write code that “implements” the algorithm.
3) Insert assertions, debugging output, etc. to verify correctness of code.
   - Code first, verify later.
   - Better to write predicates first, derive code from predicates.
4) Run on lots of test cases.

Problems with Typical Process

1) Develop algorithm to solve problem.
2) Write code that “implements” the algorithm.
3) Insert assertions, debugging output, etc. to verify correctness of code.
4) Run on lots of test cases.
   - Tedious, time consuming.
   - No guarantee of coverage, correctness.

A Better Process

1) Derive algorithm using a systematic process.
   - Determine pre/post-conditions, loop invariants.
   - Determine initialization, loop updates that preserve the above.
2) Implement using appropriate API so that the code looks like the algorithm.

Algorithm Skeleton

Step	Annotated Algorithm: $L := L^{-1}$
1a	$\left\{ L = \hat{L} \right\}$
4	Partition where
2	$\{P_{\text{inv}}\}$
3	while G do
2,3	$\{P_{\text{inv}} \wedge G\}$
5a	Repartition where
6	$\{P_{\text{before}}\}$
8	S_U
7	$\{P_{\text{after}}\}$
5b	Continue with
2	$\{P_{\text{inv}}\}$
	enddo
2,3	$\{P_{\text{inv}} \wedge \neg G\}$
1b	$\left\{ L = \hat{L}^{-1} \right\}$

Algorithm Derivation

- Eight step process to fill in algorithm skeleton.
- Example – in-place inversion of an *m x m* lower triangular matrix.

$$A := A^{-1}$$

Step 1: Pre- and Post-conditions

Step	Annotated Algorithm: $L := L^{-1}$
1a	$\left\{ L = \hat{L} \right\}$
4	Partition where
2	$\{P_{\text{inv}}\}$
3	while G do
2,3	$\{P_{\text{inv}} \wedge G\}$
5a	Repartition where
6	$\{P_{\text{before}}\}$
8	S_U
7	$\{P_{\text{after}}\}$
5b	Continue with
2	$\{P_{\text{inv}}\}$
	enddo
2,3	$\{P_{\text{inv}} \wedge \neg G\}$
1b	$\left\{ L = \hat{L}^{-1} \right\}$

- Determine pre- and post-conditions – essentially input and output of algorithm.
- Example:

  pre: $A = \hat{A}$

  post: $A = \hat{A}^{-1}$

  ($\hat{A}$ is initial matrix)

Step 2.1: Loop invariant

- Partition matrix, substitute into post-condition.
- Example: A → $\left(\begin{array}{c|c} A_{TL} & \\ \hline A_{BL} & A_{BR} \end{array}\right)$

  Plugging into post-condition:

$$\left(\begin{array}{c|c} A_{TL} & \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} \hat{A}_{TL} & \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)^{-1} = \left(\begin{array}{c|c} \hat{A}^{-1}{}_{TL} & \\ \hline -\hat{A}^{-1}{}_{BR}\, \hat{A}_{BL}\, \hat{A}^{-1}{}_{TL} & \hat{A}^{-1}{}_{BR} \end{array}\right)$$

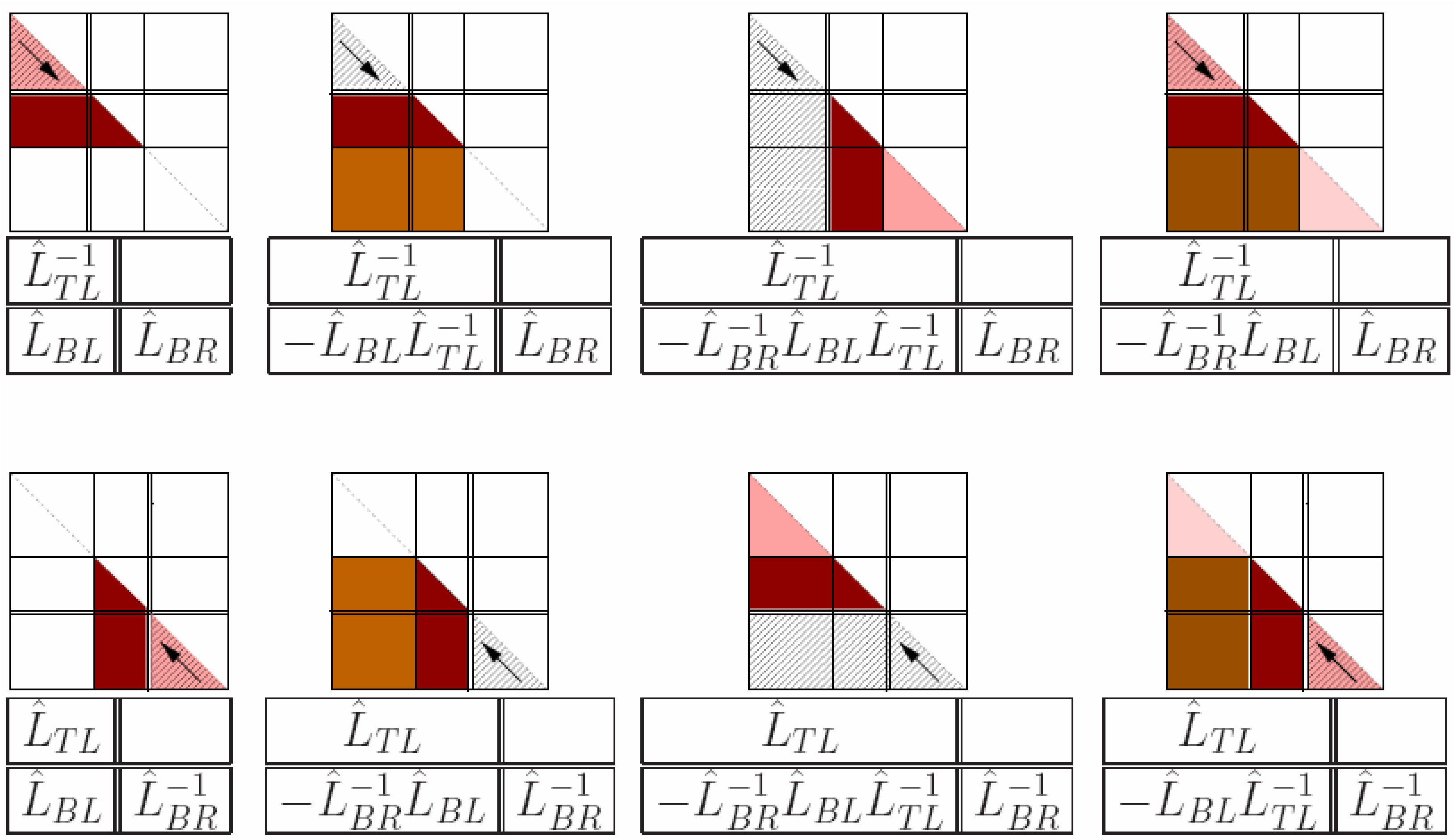
(RHS derived using linear algebra identities)

Step 2.2: Loop Invariant

- Consider individual operations – decide which ones correspond to intermediate results.
  - Programmer decision – different sets of completed operations correspond to different invariants.
- Example:

$$\left(\begin{array}{c|c} A_{TL} & \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} \hat{A}^{-1}{}_{TL} & \\ \hline -\hat{A}_{BL}\,\hat{A}^{-1}{}_{TL} & \hat{A}_{BR} \end{array}\right)$$

Possible Invariants



Steps 3-5

3) Determine loop guard.
4) Determine initialization.
5) Determine how to move partition boundary.

$$\left(\begin{array}{c|c} A_{TL} & \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & & \\ \hline A_{10} & A_{11} & \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

All of the above follow from the loop invariant.

Steps 6-7

Step	Annotated Algorithm: $L := L^{-1}$
1a	$\left\{ L = \hat{L} \right\}$
4	Partition where
2	$\{P_{\text{inv}}\}$
3	while G do
2,3	$\{P_{\text{inv}} \wedge G\}$
5a	Repartition where
6	$\{P_{\text{before}}\}$
8	S_U
7	$\{P_{\text{after}}\}$
5b	Continue with
2	$\{P_{\text{inv}}\}$
	enddo
2,3	$\{P_{\text{inv}} \wedge \neg G\}$
1b	$\left\{ L = \hat{L}^{-1} \right\}$

6) Determine predicate before loop update.
7) Determine predicate after loop update.

Determined by substituting result of step 5 into loop invariant and simplifying.

Step 8: Loop Update

Step	Annotated Algorithm: $L := L^{-1}$
1a	$\left\{ L = \hat{L} \right\}$
4	Partition where
2	$\{P_{\text{inv}}\}$
3	while G do
2,3	$\{P_{\text{inv}} \wedge G\}$
5a	Repartition where
6	$\{P_{\text{before}}\}$
8	S_U
7	$\{P_{\text{after}}\}$
5b	Continue with
2	$\{P_{\text{inv}}\}$
	enddo
2,3	$\{P_{\text{inv}} \wedge \neg G\}$
1b	$\left\{ L = \hat{L}^{-1} \right\}$

- Compare before and after predicates, determine what must be changed between the two.

End Result

- Predicates can be removed, since they aren't needed by the algorithm.

Partition $L = \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$

where L_{TL} is 0×0

while $\neg \mathsf{SameSize}(L, L_{TL})$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$$

where L_{11} is $b \times b$

$L_{21} := -L_{21}\ L_{11}^{-1}$

$L_{20} := L_{20} + L_{21}\ L_{10}$

$L_{10} := L_{11}^{-1}\ L_{10}$

$L_{11} := L_{11}^{-1}$

Continue with

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$$

enddo

Implementation

- FLAME provides a set of APIs that provide high level linear algebra operations.
  - Available for many languages.
- Algorithm can be coded by transforming operations into calls to FLAME.
  - Code can also be generated automatically from algorithm.
- Implementation actually looks like algorithm.

Actual Code

```
void Trinv ( FLA_Obj L, int nb )
{
  FLA_Part_2x2 ( L, &LTL, /**/ &LTR,
                /* *************** */
                   &LBL, /**/ &LBR,
           /* with */ 0, /* x */ 0, /* quadrant */ FLA_TL );

  while (b = min ( FLA_Obj_length ( LBR ), nb ) ){

    FLA_Repart_2x2_to_3x3 ( LTL, /**/ LTR,     &L00, /**/ &L01, &L02,
                          /* ************* */ /* ******************** */
                                                &L10, /**/ &L11, &L12,
                             LBL, /**/ LBR,     &L20, /**/ &L21, &L22,
                        /* with */ b, /* x */ b, /* L11 from */ FLA_BR );

    /* ------------ Compute ------------------------------------------ */
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG, MINUS_ONE, L11, L21 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE, ONE, L21, L10, ONE, L20 );
    FLA_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG, MINUS_ONE, L11, L10 );
    Trinv_unblocked( L11 );
    /* -------------------------------------------------------------- */

    FLA_Cont_with_3x3_to_2x2 ( &LTL, /**/ &LTR,      L00, L01, /**/ L02,
                                                     L10, L11, /**/ L12,
                             /* ************** */ /* ***************** */
                               &LBL, /**/ &LBR,      L20, L21, /**/ L22,
                             /* with L11 added to */ FLA_TL );
  }
}
```

Automation

- Much of the above process can be automated.
  - For some simple problems, entire process can be done automatically.

Automated Steps 1-2

- (Step 1) Pre- and post-conditions define problem, so must be provided.
- (Step 2.1) Partitioned expression must be provided.
  - (Step 2.2) Loop invariant can be automatically derived from partitioned expression – select subset of operations.

Automated Steps 3-5

- (Step 3) Loop guard can be derived from invariant – essentially when partition reaches the matrix boundaries.
- (Step 4) Initialization usually only requires placing partition line at the boundaries.
- (Step 5) How to move partition boundaries can be determined by comparing initialization to loop guard.

Automated Steps 6-7

- (Step 6) To obtain before predicate:
  - Substitute result of step 5 into loop invariant – can be done automatically.
  - Simplify – requires symbolic manipulation (e.g. Mathematica)
- (Step 7) After predicate is similar to above.

Automated Step 8

- (Step 8) Determine loop update by comparing before and after predicates.
  - Automation requires pattern matching, symbolic manipulation, library of transformation rules.

Status of Automation

- Fully automated system exists for limited set of linear algebra problems.
- Prototype semi-automated system for general problems:
  - Loop invariant required as input.
  - Output is partially filled skeleton – all but loop update computed, and hints provided for what needs to be done in loop update.

Benefits of FLAME

- Algorithm derivation and implementation can now be done in hours instead of months.
  - Can be done by non-experts.
- Performance comparable to vendor-supplied libraries.
- Implementation is provably correct.
- Parallelization of code is trivial – just call parallel version of FLAME API.

Future Work

- Improve prototype system to generate complete algorithm.
- Implement stability and performance analysis of algorithms.