MathMatrix

IEC 61131 Library for ACSELERATOR RTAC® Projects

SEL Automation Controllers

Table of Contents

Section 1: MathMatrix

Introduction 3
Supported Firmware Versions 4
Enumerations 4
Functions 5
Classes 13
Benchmarks 50
Examples 53
Release Notes 60

RTAC LIBRARY

MathMatrix

Introduction

The MathMatrix library allows for the creation of matrices of complex numbers. There are multiple desired workflows that exist when working with matrices and the library provides several options for working with them.

The library is designed to facilitate two basic types of matrix operations: operations that modify an existing matrix and operations that take one or more matrices as arguments and place the result in a different matrix. Operations that affect only the active matrix are completed using the methods on class_Matrix objects. Operations that affect two or more matrices are performed by external functions or special matrix manipulation classes.

The library also allows for operations of varying levels of required immediacy. For work on large or highly variant sized matrices that can be completed over multiple task cycles, it provides matrix manipulation classes that are loaded with operator and result matrices, stimulated to run to completion, and given a fixed number of steps or a fixed time slice. For operations that must complete immediately, ideally on fixed sized matrices so the computation time can be evaluated to validate timing requirements, functions provided by the library accomplish the same work while guaranteeing the completion of the algorithm before returning.

NOTE: Because of the cost of checking the system time, the time is not validated at each step in the algorithm but rather after multiple steps have been completed.

This library is dependent on the capabilities defined in the MathComplex library for all operations (see the MathComplex library documentation for more information on the operation of this library).

Special Considerations

- Copying classes from this library causes unwanted behavior. This means the following:

  1. The assignment operator ":=" must not be used on any class from this library; consider assigning pointers to the objects instead.

```
// This is bad and in most cases will provide a compiler error
    such as:
// "C0328: Assignment not allowed for type
    class_MathMatrixObject"
myMathMatrixObject := otherMathMatrixObject;
```

```
// This is fine
someVariable := myMathMatrixObject.value;
// As is this
pt_myMathMatrixObject := ADR(myMathMatrixObject);
```

2. Classes from this library must never be VAR_INPUT or VAR_OUTPUT members in function blocks, functions, or methods. Place them in the VAR_IN_OUT section or use pointers instead.

- Classes in this library have memory allocated inside them. As such, they should only be created in environments of permanent scope (e.g., Programs, Global Variable Lists, or VAR_STAT sections).
- Though this library provides the capability to dynamically resize, create, and destroy matrices, memory operations can be long running and care should be taken to avoid unnecessary use of this type of operation on a time-critical task.

Supported Firmware Versions

You can use this library on any device configured using ACSELERATOR RTAC® SEL-5033 Software with firmware version R143 or higher.

Versions 3.5.0.1 and older can be used on RTAC firmware version R132 and higher.

Enumerations

Enumerations make code more readable by allowing a specific number to have a readable textual equivalent.

enum_MatrixState

Enumeration	Description
NOT_INITIALIZED	This matrix has no memory assigned to it, call `SetSize()` to initialize.
NO_OPERATION	The matrix is not locked to any operation.
MATRIX_SCALE	The matrix is being scaled by one value.
EXTERNAL_OPERATION	The matrix is being operated on by an external class.
MATRIX_ROW_STEP_MULT	The matrix is multiplying a row by a scalar.
MATRIX_ROW_STEP_DIV	The matrix is dividing a row by a scalar.
MATRIX_ROW_STEP_ADD	The matrix is adding two rows together.
MATRIX_ROW_STEP_SUB	The matrix is subtracting one row from another.

Functions

fun_DeleteMatrix (Function)

The user should call this after completing work on a matrix received through `fun_NewMatrix()` before the matrix goes out of scope. It releases all system resources.

Inputs/Outputs

Name	IEC 61131 Type	Description
pt_matrix	POINTER TO class_Matrix	The matrix to be deleted. This pointer must be received through fun_NewMatrix.

Return Value

IEC 61131 Type	Description
BOOL	TRUE if the memory is successfully deallocated.

Processing

This function frees all system resources owned by this matrix. After completion of the function, pt_matrix is NULL(0).

fun_MatrixAdd (Function)

This function adds two matrices and places the result in a third. The entire operation will complete before the function returns.

Inputs/Outputs

Name	IEC 61131 Type	Description
matrix1	class_Matrix	The first addend.
matrix2	class_Matrix	The second addend.
result	class_Matrix	The sum of the two matrices.

Return Value

IEC 61131 Type	Description
BOOL	The matrix addition completed successfully.

Processing

This function sets the return value to TRUE if all conditions for performing the addition are met as follows:

- *matrix1* and *matrix2* are initialized.
- All three matrices are not busy performing a stepwise operation.
- *result* is a separate matrix from both *matrix1* and *matrix2*.
- *matrix1* and *matrix2* have the same dimensions.
- If necessary, *result* is successfully resized.

It then performs the addition.

fun_MatrixCopyColumn (Function)

This function copies one column from one matrix to a column in a second matrix. The entire operation will complete before the function returns.

Inputs

Name	IEC 61131 Type	Description
fromColumn	UINT	The column index of the column being copied from.
toColumn	UINT	The column index of the column being copied to.

Inputs/Outputs

Name	IEC 61131 Type	Description
fromMatrix	class_Matrix	The matrix being copied from.
toMatrix	class_Matrix	The matrix being copied to.

Return Value

IEC 61131 Type	Description
BOOL	The column copy completed successfully.

Processing

This function sets the return value to TRUE if all conditions for performing the copy are met as follows:

- Both matrices are initialized.
- Both matrices are not busy performing a stepwise operation.
- Both matrices have the same number of rows.
- The column indices provided are within the size of the matrices referenced.

It then performs the copy.

fun_MatrixDeterminant (Function)

This function calculates the determinant of a square matrix. The entire operation will complete before the function returns.

If the purpose behind calculating the determinant is a check before inverting a matrix or as part of the process of solving a system of equations this class is not the most optimal to use. In these cases it is better to use the fun_MatrixInvert or the fun_MatrixGaussianElim object instead as the overhead for all three is similar.

Inputs/Outputs

Name	IEC 61131 Type	Description
original	class_Matrix	The matrix to calculate the determinant of. This matrix is left unchanged.
workspace	class_Matrix	Memory to do the calculation in. If this is the same size as *original*, no memory allocation will occur in finding the determinant.

Outputs

Name	IEC 61131 Type	Description
determinant	struct_ComplexRect	The determinant of the matrix. Zero if the matrix is not invertible.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the operation was attempted.

Processing

1. This function sets the return value to TRUE if all conditions for performing the calculation are met as follows:
   - ➤ *original* is initialized.
   - ➤ *original* is a square matrix.
   - ➤ *workspace* is a separate matrix from *original*.
   - ➤ Neither matrix is busy performing some other operation.
   - ➤ If necessary, *workspace* is successfully resized.
2. Copies the contents of *original* into *workspace*.
3. Reduces *workspace* to an identity matrix using elementary row operations.
4. Calculates the determinant from the row operations performed.
5. If at any time the row operations cannot reduce *workspace* further and it is still not an identity matrix, the operation is terminated and *determinant* is set to zero.

fun_MatrixGaussianElim (Function)

This function simplifies a matrix to a diagonal ones matrix with trailing columns using Gaussian elimination. The contents of *coefficients* are destroyed and the contents of *solutions* are modified by this function. The entire operation will complete before the function returns.

Inputs/Outputs

Name	IEC 61131 Type	Description
coefficients	class_Matrix	The coefficients of the variables being solved for.
solutions	class_Matrix	The right hand side of the system of equations.

Outputs

Name	IEC 61131 Type	Description
error	BOOL	This algorithm cannot solve this system of equations.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the Gaussian elimination was attempted and the matrices could have been modified.

Processing

- This function sets the return value to TRUE if all conditions for performing the calculation are met as follows:
  - Both matrices are initialized.
  - Both matrices are not busy performing a stepwise operation.
  - *coefficients* is a separate matrix from *solutions*.
  - *coefficients* has at least as many columns as rows.
  - *solutions* has the same number of rows as *coefficients*.
- Reduces the first `Rows` • `Rows` of *coefficients* to an identity matrix using elementary row operations.
- Performs the same row operations on *solutions*.
- If at any time the row operations cannot reduce *coefficients* further and there is still not an identity matrix on the left the operation is terminated and *error* is set.
- The contents of *coefficients* are destroyed and the contents of *solutions* are modified by this method.

Output Combination Meanings

Error	Return	Description
FALSE	FALSE	This should never occur.
FALSE	TRUE	The Gaussian elimination completed successfully.
TRUE	FALSE	The matrix state prevented the Gaussian elimination request.
TRUE	TRUE	The matrix is not invertible and cannot be reduced by this Gaussian elimination algorithm.

fun_MatrixInvert (Function)

This function creates the inverse of a square matrix. *original* is destroyed in the process so if the data are still desired, they must be copied before the function is called. The entire operation will complete before the function returns.

One common use case for inverting a matrix is to solve a system of equations. In this library that use case is discouraged unless solving the same system for many solutions as Gaussian elimination performs the same functionality with less overhead.

Inputs/Outputs

Name	IEC 61131 Type	Description
original	class_Matrix	The matrix to invert.
result	class_Matrix	The inverted matrix.

Outputs

Name	IEC 61131 Type	Description
error	BOOL	The inversion could not be attempted or *original* cannot be inverted.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if inversion was attempted and the matrices could have been modified.

Processing

1. This function sets the return value to TRUE if all conditions for performing the calculation are met as follows:
   - *original* is initialized.
   - Both matrices are not busy performing a stepwise operation.
   - *result* is a separate matrix from *original*.
   - *original* is a square matrix.

- ➤ If necessary, *result* is successfully resized.

2. Sets *result* to an identity matrix.
3. Reduces *original* to an identity matrix using elementary row operations.
4. Performs the same row operations on *result* to create the inverse.
5. If at any time the row operations cannot reduce *original* further and it is still not an identity matrix, the operation is terminated and *error* is set.

Output Combination Meanings

Error	Return	Description
FALSE	FALSE	This should never occur.
FALSE	TRUE	The inversion completed successfully.
TRUE	FALSE	The matrix state prevented the inversion request.
TRUE	TRUE	The matrix is not invertible.

fun_MatrixMultiply (Function)

This function multiplies two matrices and places the result in a third. The entire operation will complete before the function returns.

Inputs/Outputs

Name	IEC 61131 Type	Description
matrix1	class_Matrix	The multiplier.
matrix2	class_Matrix	The multiplicand.
result	class_Matrix	The product of the two matrices.

Return Value

IEC 61131 Type	Description
BOOL	The matrix multiplication completed successfully.

Processing

This function sets the return value to TRUE if all conditions for performing the calculation are met as follows:

- ➤ *matrix1* and *matrix2* are initialized.
- ➤ All three matrices are not busy performing a stepwise operation.
- ➤ *result* is a separate matrix from both *matrix1* and *matrix2*.
- ➤ The column count of *matrix1* equals the row count of *matrix2*.

If necessary, it sets the size of *result*. It then performs the multiplication.

fun_MatrixSubtract (Function)

This function subtracts one matrix from another and places the result in a third. The entire operation will complete before the function returns.

Inputs/Outputs

Name	IEC 61131 Type	Description
matrix1	class_Matrix	The minuend.
matrix2	class_Matrix	The subtrahend.
result	class_Matrix	The difference of the two matrices.

Return Value

IEC 61131 Type	Description
BOOL	The matrix subtraction completed successfully.

Processing

This function verifies that the subtraction can be performed:

1. *matrix1* and *matrix2* are initialized.
2. All three matrices are not busy performing a stepwise operation.
3. *result* is a separate matrix from both *matrix1* and *matrix2*.
4. *matrix1* and *matrix2* have the same dimensions.

If necessary, the function resizes *result*. It then performs the subtraction.

fun_MatrixTranspose (Function)

This function places the transpose of a matrix into a result. The entire operation will complete before the function returns.

Inputs

Name	IEC 61131 Type	Description
conjugate	BOOL	The result of this operation will be the Hermitian Transpose. Before each element is placed in *result*, it will be conjugated.

Inputs/Outputs

Name	IEC 61131 Type	Description
original	class_Matrix	The matrix whose transpose is calculated.
result	class_Matrix	The transpose of the original matrix.

Return Value

IEC 61131 Type	Description
BOOL	The matrix transpose completed successfully.

Processing

This function verifies that the transpose can be performed:

1. *original* is initialized.
2. Both matrices are not busy performing a stepwise operation.
3. *result* is a separate matrix from *original*.

If necessary the function resizes *result*. It then performs the transpose. If *conjugate* is TRUE, conjugate each element in *result*.

fun_Matrix_ATA (Function)

This function performs an optimization of the operation (A^TA) transposing an input matrix and multiplying it by itself. The entire operation will complete before the function returns.

Inputs

Name	IEC 61131 Type	Description
conjugate	BOOL	The result of this operation will be calculated using the Hermitian Transpose. Before the transpose step is complete, each element will be conjugated.

Inputs/Outputs

Name	IEC 61131 Type	Description
original	class_Matrix	The matrix A.
result	class_Matrix	The matrix for the result.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the operation completed successfully.

Processing

This function verifies that the operation can be performed:

1. *original* is initialized.
2. Both matrices are not busy performing a stepwise operation.
3. *result* is a separate matrix from *original*.

If necessary, the function resizes *result*. It then performs the operation A^TA. If *conjugate* is TRUE, conjugate each element in the transpose before using it in the multiply.

fun_NewMatrix (Function)

Request a new matrix from the system with all required resources. Matrices received through this function must be freed through the `fun_DeleteMatrix()` function before they leave scope.

Inputs

Name	IEC 61131 Type	Description
rows	UINT	The number of rows in the new matrix.
cols	UINT	The number of columns in the new matrix.

Return Value

IEC 61131 Type	Description
POINTER TO class_Matrix	The address of the new class_Matrix.

Processing

This function allocates system resources for a *rows* by *cols* matrix of struct_ComplexRect objects.

Classes

This library provides the following classes as extensions of the IEC 61131 function block.

class_Matrix (Class)

This is the fundamental class for this library. It allows for the storage of struct_ComplexRect objects ordered by row and column. It manages all required system resources internally.

Initialization Inputs

Name	IEC 61131 Type	Description
rowCount	UINT	The number of rows this matrix begins with.
colCount	UINT	The number of columns this matrix begins with.

Outputs

Name	IEC 61131 Type	Description
pt_Data	POINTER TO POINTER TO struct_ComplexRect	Pointer to an array of pointers (one for each row). Allows accessing the matrix with [row][col] syntax. Indexing starts at zero. This pointer should be re-read before access after any resize operation.
Rows	UINT	The number of rows in the matrix.
Cols	UINT	The number of columns in the matrix.
State	enum_MatrixState	The active matrix operation.

Clear (Method)

This method returns all system resources internal to this matrix and sets its size to zero rows by zero columns. In addition it clears all locks on the matrix and resets all internal state machines.

This should typically be used only to free the system resources held by this matrix before it goes out of scope.

MatrixRowAdd (Method)

This method adds one row to another inside this matrix, replacing the content of the second row (Matrix[*toRow*] = Matrix[*toRow*] + Matrix[*fromRow*] • *scalar*). The entire operation will complete before the method returns.

Inputs

Name	IEC 61131 Type	Description
fromRow	UINT	The first addend.
toRow	UINT	The second addend and the location of the result.
scalar	struct_ComplexRect	A constant that is multiplied against the value of each entry in *fromRow* before adding it to the entry in *toRow*.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method performed the addition.

Processing

1. Validates that the matrix is initialized and not in the middle of a stepwise operation.
2. Validates that the provided row indices exist.
3. If the checks pass, multiplies *fromRow* by *scalar* and adds the result to *toRow*.
4. *fromRow* remains unchanged.

MatrixRowDivide (Method)

This method divides each element in *rowIndex* by *scalar* and stores the results back in *rowIndex* (Matrix[*rowIndex*] = Matrix[*rowIndex*] / *scalar*). The entire operation will complete before the method returns.

Inputs

Name	IEC 61131 Type	Description
rowIndex	UINT	The row to be modified.
scalar	struct_ComplexRect	A constant used as the divisor against the value of each entry in *rowIndex*.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method performed the division.

Processing

1. Validates that the matrix is initialized and not in the middle of a stepwise operation.
2. Validates the row index provided exists.
3. If the checks pass, divides each entry in *rowIndex* by *scalar*.

MatrixRowMultiply (Method)

This method multiplies each element in *rowIndex* by *scalar* and stores the results back in *rowIndex* (Matrix[*rowIndex*] = Matrix[*rowIndex*] • *scalar*). The entire operation will complete before the method returns.

Inputs

Name	IEC 61131 Type	Description
rowIndex	UINT	The row to be modified.
scalar	struct_ComplexRect	A constant used as the multiplier against the value of each entry in *rowIndex*.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method performed the multiplication.

Processing

1. Validates that the matrix is initialized and not in the middle of a stepwise operation.
2. Validates the row index provided exists.
3. If the checks pass, multiplies each entry in *rowIndex* by *scalar*.

MatrixRowSubtract (Method)

This method subtracts one row from another inside this matrix, replacing the content of the second row (Matrix[*toRow*] = Matrix[*toRow*] – Matrix[*fromRow*] • *scalar*). The entire operation will complete before the method returns.

Inputs

Name	IEC 61131 Type	Description
fromRow	UINT	The subtrahend.
toRow	UINT	The minuend and the location of the result.
scalar	struct_ComplexRect	A constant that is multiplied against the value of each entry in *fromRow* before subtracting it from the entry in *toRow*.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method performed the subtraction.

Processing

1. Validates that the matrix is initialized and not in the middle of a stepwise operation.
2. Validates the row indices provided exist.
3. If the checks pass, multiplies *fromRow* by *scalar* and subtracts the result from *toRow*.
4. *fromRow* remains unchanged.

MatrixScale (Method)

Multiplies each element in this matrix by a scalar. The entire operation will complete before the method returns.

Inputs

Name	IEC 61131 Type	Description
scalar	struct_ComplexRect	A constant that is multiplied against the value of each entry this matrix.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method scaled the matrix.

Processing

1. Validates that the matrix is initialized and not in the middle of a stepwise operation.
2. If the checks pass, multiplies each element in the matrix by *scalar*, placing the result in the same location.

MatrixStepRowAdd (Method)

This method adds one row to another inside this matrix, replacing the content of the second row (Matrix[*toRow*] = Matrix[*toRow*] + Matrix[*fromRow*] • *scalar*).

The operation will perform the next *steps* operations of the complete addition. This design allows for the completion of the algorithm over the course of multiple task cycles for matrices large enough for completion time to be a concern.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of operations to attempt this task cycle.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method completed the addition.

Processing

1. Validates that the matrix is initialized and has begun a stepwise addition.
2. If the checks pass, this method uses the values provided in `StartMatrixOperation()` to multiply *fromRow* by *scalar* and add the result to *toRow*.
3. *fromRow* remains unchanged.
4. Performs, at most, the next *steps* operations toward completing the addition algorithm.
5. Decrements *steps* by the number of operations consumed.
6. Returns TRUE and unlocks the matrix if the addition completed.
7. Returns FALSE if the addition was not attempted or *steps* was exhausted before the algorithm completed.

MatrixStepRowDivide (Method)

This method divides each element in *toRow* by *scalar* and stores the results back in *toRow* (Matrix[*toRow*] = Matrix[*toRow*] / *scalar*).

The operation will perform the next *steps* operations of the complete division. This design allows for the completion of the algorithm over the course of multiple task cycles for matrices large enough for completion time to be a concern.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of operations to attempt this task cycle.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method completed the division.

Processing

1. Validates that the matrix is initialized and has begun a stepwise division.
2. If the checks pass, this method uses the values provided in `StartMatrixOperation()` to divide each element in *toRow* by *scalar*.
3. Performs, at most, the next *steps* operations toward completing the division algorithm.
4. Decrements *steps* by the number of operations consumed.
5. Returns TRUE and unlocks the matrix if the division completed.
6. Returns FALSE if the division was not attempted or *steps* was exhausted before the algorithm completed.

MatrixStepRowMultiply (Method)

This method multiplies each element in *toRow* by *scalar* and stores the results back in *toRow* (Matrix[*toRow*] = Matrix[*toRow*] • *scalar*).

The operation will perform the next *steps* operations of the complete multiplication. This design allows for the completion of the algorithm over the course of multiple task cycles for matrices large enough for completion time to be a concern.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of operations to attempt this task cycle.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method completed the multiplication.

Processing

1. Validates that the matrix is initialized and has begun a stepwise multiplication.
2. If the checks pass, this method uses the values provided in `StartMatrixOperation()` to multiply each entry in *toRow* by *scalar*.
3. Performs, at most, the next *steps* operations toward completing the multiplication algorithm.
4. Decrements *steps* by the number of operations consumed.
5. Returns TRUE and unlocks the matrix if the multiplication completed.
6. Returns FALSE if the multiplication was not attempted or *steps* was exhausted before the algorithm completed.

MatrixStepRowSubtract (Method)

This method subtracts one row from another inside this matrix, replacing the content of the second row (Matrix[*toRow*] = Matrix[*toRow*] – Matrix[*fromRow*] • *scalar*).

The operation will perform the next *steps* operations of the complete subtraction. This design allows for the completion of the algorithm over the course of multiple task cycles for matrices large enough for completion time to be a concern.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of operations to attempt this task cycle.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method completed the subtraction.

Processing

1. Validates that the matrix is initialized and has begun a stepwise subtraction.
2. If the checks pass, this method uses the values provided in `StartMatrixOperation()` to multiply *fromRow* by *scalar* and subtract the result from *toRow*.
3. *fromRow* remains unchanged.
4. Performs, at most, the next *steps* operations toward completing the subtraction algorithm.
5. Decrements *steps* by the number of operations consumed.

6. Returns TRUE and unlocks the matrix if the subtraction completed.
7. Returns FALSE if the subtraction was not attempted or *steps* was exhausted before the algorithm completed.

MatrixStepScale (Method)

Multiplies each element in this matrix by a scalar.

The operation will perform the next *steps* operations of the complete scaling operation. This design allows for the completion of the algorithm over the course of multiple task cycles for matrices large enough for completion time to be a concern.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of operations to attempt this task cycle.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method completed the scaling operation.

Processing

1. Validates that the matrix is initialized and has begun a stepwise scaling operation.
2. If the checks pass, this method uses the values provided in `StartMatrixOperation()` to multiply each element in the matrix by *scalar*.
3. Performs, at most, the next *steps* operations toward completing the scaling algorithm.
4. Decrements *steps* by the number of operations consumed.
5. Returns TRUE and unlocks the matrix if the scaling operation completed.
6. Returns FALSE if the scaling operation was not attempted or *steps* was exhausted before the algorithm completed.

MatrixTimedRowAdd (Method)

This method adds one row to another inside this matrix, replacing the content of the second row (Matrix[*toRow*] = Matrix[*toRow*] + Matrix[*fromRow*] • *scalar*).

The operation will perform work for the next *duration* microseconds toward completion of the addition. This design allows for the completion of the algorithm over the course of multiple task cycles for matrices large enough for completion time to be a concern.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method completed the addition.

Processing

1. Validates that the matrix is initialized and has begun a stepwise addition.
2. If the checks pass, this method uses the values provided in `StartMatrixOperation()` to multiply *fromRow* by *scalar* and add the result to *toRow*.
3. *fromRow* remains unchanged.
4. Performs work toward completing the addition algorithm, in groups of steps, until *duration* microseconds is exceeded.
5. Decrements *duration* by the microseconds consumed.
6. Returns TRUE and unlocks the matrix if the addition completed.
7. Returns FALSE if the addition was not attempted or *duration* was exhausted before the algorithm completed.

MatrixTimedRowDivide (Method)

This method divides each element in *toRow* by *scalar* and stores the results back in *toRow* (Matrix[*toRow*] = Matrix[*toRow*] / *scalar*).

The operation will perform work for the next *duration* microseconds toward the complete division. This design allows for the completion of the algorithm over the course of multiple task cycles for matrices large enough for completion time to be a concern.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method completed the division.

Processing

1. Validates that the matrix is initialized and has begun a stepwise division.
2. If the checks pass, this method uses the values provided in `StartMatrixOperation()` to divide each element in *toRow* by *scalar*.
3. Performs work toward completing the division algorithm, in groups of steps, until *duration* microseconds is exceeded.
4. Decrements *duration* by the microseconds consumed.

5. Returns TRUE and unlocks the matrix if the division completed.
6. Returns FALSE if the division was not attempted or *duration* was exhausted before the algorithm completed.

MatrixTimedRowMultiply (Method)

This method multiplies each element in *toRow* by *scalar* and stores the results back in *toRow* (Matrix[*toRow*] = Matrix[*toRow*] • *scalar*).

The operation will perform work for the next *duration* microseconds toward the complete multiplication. This design allows for the completion of the algorithm over the course of multiple task cycles for matrices large enough for completion time to be a concern.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method completed the multiplication.

Processing

1. Validates that the matrix is initialized and has begun a stepwise multiplication.
2. If the checks pass, this method uses the values provided in `StartMatrixOperation()` to multiply each entry in *toRow* by *scalar*.
3. Performs work toward completing the multiplication algorithm, in groups of steps, until *duration* microseconds is exceeded.
4. Decrements *duration* by the microseconds consumed.
5. Returns TRUE and unlocks the matrix if the multiplication completed.
6. Returns FALSE if the multiplication was not attempted or *duration* was exhausted before the algorithm completed.

MatrixTimedRowSubtract (Method)

This method subtracts one row from another inside this matrix, replacing the content of the second row (Matrix[*toRow*] = Matrix[*toRow*] – Matrix[*fromRow*] • *scalar*).

The operation will perform work for the next *duration* microseconds toward the complete subtraction. This design allows for the completion of the algorithm over the course of multiple task cycles for matrices large enough for completion time to be a concern.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method completed the subtraction.

Processing

1. Validates that the matrix is initialized and has begun a stepwise subtraction.
2. If the checks pass, this method uses the values provided in `StartMatrixOperation()` to multiply *fromRow* by *scalar* and subtract the result from *toRow*.
3. *fromRow* remains unchanged.
4. Performs work toward completing the subtraction algorithm, in groups of steps, until *duration* microseconds is exceeded.
5. Decrements *duration* by the microseconds consumed.
6. Returns TRUE and unlocks the matrix if the subtraction completed.
7. Returns FALSE if the subtraction was not attempted or *duration* was exhausted before the algorithm completed.

MatrixTimedScale (Method)

Multiplies each element in this matrix by a scalar.

The operation will perform work for the next *duration* microseconds toward the complete scaling operation. This design allows for the completion of the algorithm over the course of multiple task cycles for matrices large enough for completion duration to be a concern.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method completed the scaling operation.

Processing

1. Validates that the matrix is initialized and has begun a stepwise scaling operation.

2. If the checks pass, this method uses the values provided in `StartMatrixOperation()` to multiply each element in the matrix by *scalar*.
3. Performs work toward completing the scaling algorithm, in groups of steps, until *duration* microseconds is exceeded.
4. Decrements *duration* by the microseconds consumed.
5. Returns TRUE and unlocks the matrix if the scaling operation completed.
6. Returns FALSE if the scaling operation was not attempted or *duration* was exhausted before the algorithm completed.

RowSwap (Method)

This method exchanges the position of two rows in a given matrix.

Inputs

Name	IEC 61131 Type	Description
row1	UINT	The first row to swap.
row2	UINT	The second row to swap.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method performed the row swap.

Processing

1. Validates that the matrix is initialized and not performing any stepwise operation.
2. If the checks pass, swaps the positions of *row1* and *row2*.
3. Returns TRUE if the swap succeeded.
4. Returns FALSE if the swap failed.

SetSize (Method)

This method changes the storage capacity of the matrix modifying `Rows` and `Cols`.

Inputs

Name	IEC 61131 Type	Description
rowCount	UINT	The new number of rows.
colCount	UINT	The new number of columns.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method resized the matrix.

Processing

1. Validates that the matrix is not performing any stepwise operation.
2. If either *rowCount* or *colCount* is zero, sets `Rows` and `Cols` to zero and frees all system resources.
3. If both *rowCount* equals `Rows` and *colCount* equals `Cols`, leaves the matrix unchanged.
4. If *rowCount* is greater than `Rows`, adds zeroed rows to the bottom of the matrix.
5. If *rowCount* is less than `Rows`, removes rows from the bottom of the matrix.
6. If *colCount* is greater than `Cols`, adds zeros to the end of each row.
7. If *colCount* is less than `Cols`, truncates each row to the new count.
8. Returns TRUE if the matrix is the newly requested size.
9. Returns FALSE if the matrix is not the newly requested size.
10. If the resize fails, the matrix retains all old values.

StartMatrixOperation (Method)

This method must be called to configure any stepwise or timed operation on only this matrix. It accepts and stores the values used during the operation.

Inputs

Name	IEC 61131 Type	Description
operation	enum_MatrixState	The stepwise operation to begin.
fromRow	UINT	The row to use in the modification. Used only in addition and subtraction.
toRow	UINT	The row to be modified.
scalar	struct_ComplexRect	The constant value to be used during the operation.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the method locked the matrix to the requested operation.

Processing

1. Validates that the matrix is initialized and not performing any stepwise operation.

2. Validates row indices required for the operation requested. For addition and subtraction, both indices must be within the matrix. For multiplication and division, only *toRow* is validated. For scaling operations, no row index is validated
3. Stores *scalar* for use during the operation.
4. Locks the matrix to prevent other operations from occurring.
5. Returns TRUE if the operation is primed.
6. Returns FALSE if anything prevents the operation from being primed.

class_MatrixAdd (Class)

This class handles the locking handshakes and the state required to add two class_Matrix objects over the course of multiple scans.

Outputs

Name	IEC 61131 Type	Description
Busy	BOOL	This class has locked class_Matrix instances and is in the middle of a calculation.

LockMatrices (Method)

This method primes the class to perform a new addition (result = matrix1 + matrix2). It must be called before each addition of two matrices to be performed.

Inputs/Outputs

Name	IEC 61131 Type	Description
matrix1	class_Matrix	The first addend.
matrix2	class_Matrix	The second addend.
result	class_Matrix	The sum of *matrix1* and *matrix2*.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the addition operation is now ready.

Processing

1. Returns FALSE if either *matrix1* or *matrix2* is not initialized.
2. Returns FALSE if *result* is not a separate matrix from both *matrix1* and *matrix2*.
3. Returns FALSE if any of the three matrices is busy doing any stepwise operation.
4. Returns FALSE if the dimensions of *matrix1* do not match those of *matrix2*.

5. Returns FALSE if *result* cannot be made to be the same dimensions as the other two matrices.
6. Returns FALSE if it cannot lock all matrices involved in the operation.
7. If all other checks succeeded, then stores required references, sets *Busy* to TRUE, and returns TRUE.
8. The contents of *result* are destroyed by this method.

ProcessSteps (Method)

This method performs the addition algorithm on three already locked-in matrices.

The operation will perform the next *steps* sub-operations of the complete addition algorithm.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of sub-operations to attempt this task cycle.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the addition completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Adds each element in *matrix1* to its corresponding element in *matrix2* and stores the sum in *result*.
3. Decrements *steps* by the number of operations performed.
4. Returns TRUE if the addition algorithm completed before *steps* was exhausted.
5. Returns FALSE if `LockMatrices()` has not been called or *steps* was exhausted before completing the algorithm.

ProcessTimed (Method)

This method performs the addition algorithm on three already locked-in matrices.

The operation will perform work for the next *duration* microseconds toward the complete addition algorithm.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the addition completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Adds each element in *matrix1* to its corresponding element in *matrix2* and stores the sum in *result*.
3. Performs work toward completing the addition algorithm, in groups of steps, until *duration* microseconds is exceeded.
4. Decrements *duration* by the microseconds consumed.
5. Returns TRUE if the addition algorithm completed before *duration* was exhausted.
6. Returns FALSE if `LockMatrices()` has not been called or *duration* was exhausted before completing the algorithm.

UnlockMatrices (Method)

This method unlocks any matrices locked by `LockMatrices()`. It only needs to be called by the user if the matrix operation has been terminated early by calling `Clear()` on any of the dependent matrices. In all other cases, the matrices will be unlocked on completion of the algorithm.

Processing

1. Requests that all locked matrices free themselves for other operations.
2. Sets *Busy* to FALSE.

class_MatrixCopyColumn (Class)

This class handles the locking handshakes and the state required to copy a column from one class_Matrix object to another over the course of multiple scans.

Outputs

Name	IEC 61131 Type	Description
Busy	BOOL	This class has locked class_Matrix instances and is in the middle of a calculation.

LockMatrices (Method)

This method primes the class to perform a new column copy. It must be called before each column copy to be performed.

Inputs

Name	IEC 61131 Type	Description
fromColumn	UINT	The index of the column to copy from.
toColumn	UINT	The index of the column to copy to.

Inputs/Outputs

Name	IEC 61131 Type	Description
fromMatrix	class_Matrix	The matrix to be copied from.
toMatrix	class_Matrix	The matrix to be copied to.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the copy operation is now ready.

Processing

1. Returns FALSE if either *fromMatrix* or *toMatrix* is not initialized.
2. Returns FALSE if either of the matrices is busy doing any stepwise operation.
3. Returns FALSE if `Rows` of *matrix1* does not match `Rows` of *matrix2*.
4. Returns FALSE if either index provided is outside of the corresponding matrix.
5. Returns FALSE if it cannot lock all matrices involved in the operation.
6. If all other checks succeeded, then stores required references, sets *Busy* to TRUE, and returns TRUE.

ProcessSteps (Method)

This method performs the copy algorithm on two already locked-in matrices.

The operation will perform the next *steps* sub-operations of the complete copy.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of sub-operations to attempt this task cycle.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the copy completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Copies each element in column *fromColumn* of *fromMatrix* to its corresponding element in column *toColumn* of *toMatrix*.
3. Decrements *steps* by the number of sub-operations performed.
4. Returns TRUE if the copy algorithm completed before *steps* was exhausted.
5. Returns FALSE if `LockMatrices()` has not been called or *steps* was exhausted before completing the algorithm.

ProcessTimed (Method)

This method performs the copy algorithm on two already locked-in matrices.

The operation will perform work for the next *duration* microseconds toward the complete copy.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the copy completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Copies each element in column *fromColumn* of *fromMatrix* to its corresponding element in column *toColumn* of *toMatrix*.
3. Performs work toward completing the copy algorithm, in groups of steps, until *duration* microseconds is exceeded.
4. Decrements *duration* by the microseconds consumed.
5. Returns TRUE if the copy algorithm completed before *duration* was exhausted.
6. Returns FALSE if `LockMatrices()` has not been called or *duration* was exhausted before completing the algorithm.

UnlockMatrices (Method)

This method unlocks any matrices locked by `LockMatrices()`. It only needs to be called by the user if the matrix operation has been terminated early by calling `Clear()` on any of the dependent matrices. In all other cases, the matrices will be unlocked on completion of the algorithm.

Processing

1. Requests that all locked matrices free themselves for other operations.
2. Sets *Busy* to FALSE.

class_MatrixDeterminant (Class)

This class handles the locking handshakes and the state required to calculate the determinant of a matrix over the course of multiple scans.

If the purpose behind calculating the determinant is a check before inverting a matrix or as part of the process of solving a system of equations this class is not the most optimal to use. In these cases it is better to use the class_MatrixInvert or the class_MatrixGaussianElim object instead as the overhead for all three is similar.

Outputs

Name	IEC 61131 Type	Description
Busy	BOOL	This class has locked class_Matrix instances and is in the middle of a calculation.

LockMatrices (Method)

This method primes the class to calculate the determinant of a new matrix. It must be called before each operation to be performed.

Inputs/Outputs

Name	IEC 61131 Type	Description
original	class_Matrix	The matrix to calculate the determinant of. This matrix is left unchanged.
workspace	class_Matrix	Memory to do the calculation in. If this is the same size as *original*, no memory allocation will occur in finding the determinant.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the operation is now ready.

Processing

1. Returns FALSE if *original* is not initialized.
2. Returns FALSE if *workspace* is not a separate matrix from *original*.
3. Returns FALSE if either of the matrices is busy doing any stepwise operation.
4. Returns FALSE if *original* is not a square matrix.

5. Returns FALSE if *workspace* cannot be resized to the correct dimensions.
6. Returns FALSE if it cannot lock all matrices involved in the operation.
7. If all other checks succeeded, then stores required references, sets *Busy* to TRUE, and returns TRUE.
8. The contents of *workspace* are destroyed by this method.

ProcessSteps (Method)

This method computes the determinant of an already locked-in matrix.

The operation will perform the next *steps* sub-operations of the complete task.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of sub-operations to attempt this task cycle.

Outputs

Name	IEC 61131 Type	Description
determinant	struct_ComplexRect	The determinant of the matrix. Zero if the calculation is incomplete or the matrix is not invertible.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the operation completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Decrements *steps* by the number of sub-operations performed.
3. Returns TRUE and outputs the calculated *determinant* if the algorithm completed before *steps* was exhausted.
4. Returns FALSE and outputs a *determinant* of zero if `LockMatrices()` has not been called or *steps* was exhausted before completing the algorithm.
5. In the case that the matrix is not invertible, outputs a *determinant* of zero.

ProcessTimed (Method)

This method computes the determinant of an already locked-in matrix.

The operation will perform work for the next *duration* microseconds toward the complete matrix operation.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Outputs

Name	IEC 61131 Type	Description
determinant	struct_ComplexRect	The determinant of the matrix. Zero if the calculation is incomplete or the matrix is not invertible.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the operation completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Performs work toward completing the algorithm, in groups of steps, until *duration* microseconds is exceeded.
3. Decrements *duration* by the microseconds consumed.
4. Returns TRUE and outputs the calculated *determinant* if the operation completed before *duration* was exhausted.
5. Returns FALSE and outputs a *determinant* of zero if `LockMatrices()` has not been called or *duration* was exhausted before completing the algorithm.
6. In the case that the matrix is not invertible, outputs a *determinant* of zero.

UnlockMatrices (Method)

This method unlocks any matrices locked by `LockMatrices()`. It only needs to be called by the user if the matrix operation has been terminated early by calling `Clear()` on any of the dependent matrices. In all other cases, the matrices will be unlocked on completion of the algorithm.

Processing

1. Requests that all locked matrices free themselves for other operations.
2. Sets *Busy* to FALSE.

class_MatrixGaussianElim (Class)

This class handles the locking handshakes and the state required to simplify the matrix to a diagonal ones matrix with trailing columns using Gaussian elimination over the course of multiple scans. The contents of *coefficients* are destroyed and the contents of *solutions* are modified by this class.

Outputs

Name	IEC 61131 Type	Description
Busy	BOOL	This class has locked class_Matrix instances and is in the middle of a calculation.

LockMatrices (Method)

This method primes the class to perform the Gaussian elimination. It must be called before each calculation to be performed.

Inputs/Outputs

Name	IEC 61131 Type	Description
coefficients	class_Matrix	The coefficients of the variables being solved for.
solutions	class_Matrix	The right hand side of the system of equations.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the Gaussian elimination is now ready.

Processing

1. Returns FALSE if either matrix is not initialized.
2. Returns FALSE if *coefficients* is not a separate matrix from *solutions*.
3. Returns FALSE if either matrix is busy doing any stepwise operation.
4. Returns FALSE if *coefficients* has fewer columns than rows.
5. Returns FALSE if *solutions* is not one column with the same number of rows as *coefficients*.
6. Returns FALSE if it cannot lock both matrices involved in the operation.
7. If all other checks succeeded, then stores required references, sets *Busy* to TRUE, and returns TRUE.

ProcessSteps (Method)

This method performs Gaussian elimination on two already locked-in matrices.

The operation will perform the next *steps* sub-operations of the complete calculation.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of sub-operations to attempt this task cycle.

Outputs

Name	IEC 61131 Type	Description
error	BOOL	This algorithm cannot solve this system of equations.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the Gaussian elimination completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Reduces the first `Rows` • `Rows` of *coefficients* to an identity matrix using elementary row operations.
3. Performs the same row operations on *solutions*.
4. If at any time the row operations cannot reduce *coefficients* further and there is still not an identity matrix on the left, the operation is terminated and *error* is set.
5. Decrements *steps* by the number of operations performed.
6. Returns TRUE if the Gaussian elimination completed before *steps* was exhausted.
7. Returns FALSE if `LockMatrices()` has not been called or *steps* was exhausted before completing the algorithm.
8. The contents of *coefficients* are destroyed and the contents of *solutions* are modified by this method.

Output Combination Meanings

Error	Return	Description
FALSE	FALSE	This should never occur.
FALSE	TRUE	The Gaussian elimination completed successfully.
TRUE	FALSE	The matrix state prevented the Gaussian elimination request.
TRUE	TRUE	The matrix is not invertible and cannot be reduced by this Gaussian elimination algorithm.

ProcessTimed (Method)

This method performs Gaussian elimination on two already locked-in matrices.

The operation will perform work for the next *duration* microseconds toward the complete inversion algorithm.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Outputs

Name	IEC 61131 Type	Description
error	BOOL	This algorithm cannot solve this system of equations.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the Gaussian elimination completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Reduces the first `Rows` • `Rows` of *coefficients* to an identity matrix using elementary row operations.
3. Performs the same row operations on *solutions*.
4. If at any time the row operations cannot reduce *coefficients* further and there is still not an identity matrix on the left, the operation is terminated and *error* is set.
5. Performs work toward completing the algorithm, in groups of steps, until *duration* microseconds is exceeded.
6. Decrements *duration* by the microseconds consumed.
7. Returns TRUE if the Gaussian elimination completed before *duration* was exhausted.
8. Returns FALSE if `LockMatrices()` has not been called or *duration* was exhausted before completing the algorithm.
9. The contents of *coefficients* are destroyed and the contents of *solutions* are modified by this method.

The table, listed for the previous method, is provided as reference for interpreting output combinations.

UnlockMatrices (Method)

This method unlocks any matrices locked by `LockMatrices()`. It only needs to be called by the user if the matrix operation has been terminated early by calling `Clear()` on any of the dependent matrices. In all other cases, the matrices will be unlocked on completion of the algorithm.

Processing

1. Requests that all locked matrices free themselves for other operations.
2. Sets *Busy* to FALSE.

class_MatrixInvert (Class)

This class handles the locking handshakes and the state required to create the inverse of a square matrix over the course of multiple scans. The contents of *original* are destroyed in the process so if the data are still desired, they must be copied before this class is used. The entire operation will complete before the function returns.

One common use case for inverting a matrix is to solve a system of equations. In this library that use case is discouraged unless solving the same system for many solutions as Gaussian elimination performs the same functionality with less overhead.

Outputs

Name	IEC 61131 Type	Description
Busy	BOOL	This class has locked class_Matrix instances and is in the middle of a calculation.

LockMatrices (Method)

This method primes the class to invert a matrix. (result = original^{-1}). It must be called before each inversion to be performed.

Inputs/Outputs

Name	IEC 61131 Type	Description
original	class_Matrix	The matrix to invert.
result	class_Matrix	The inverted matrix.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the inversion operation is now ready.

Processing

1. Returns FALSE if *original* is not initialized.
2. Returns FALSE if *result* is not a separate matrix from *original*.
3. Returns FALSE if any of the matrices is busy doing any stepwise operation.
4. Returns FALSE if *original* is not a square matrix.
5. Returns FALSE if *result* cannot be sized correctly to store the product.
6. Returns FALSE if it cannot lock all matrices involved in the operation.
7. If all other checks succeeded, then stores required references, sets *Busy* to TRUE, and returns TRUE.
8. The contents of *result* are destroyed by this method.

ProcessSteps (Method)

This method performs the inversion algorithm on two already locked-in matrices.

The operation will perform the next *steps* sub-operations of the complete inversion algorithm.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of sub-operations to attempt this task cycle.

Outputs

Name	IEC 61131 Type	Description
error	BOOL	The matrix is not invertible.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the inversion completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Sets *result* to an identity matrix.
3. Reduces *original* to an identity matrix using elementary row operations.
4. Performs the same row operations on *result* to create the inverse.
5. If at any time the row operations cannot reduce *original* further and it is still not an identity matrix, the operation is terminated and *error* is set.
6. Decrements *steps* by the number of operations performed.

7. Returns TRUE if the inversion algorithm completed before *steps* was exhausted.
8. Returns FALSE if `LockMatrices()` has not been called or *steps* was exhausted before completing the algorithm.
9. The contents of *original* are destroyed by this method.

ProcessTimed (Method)

This method performs the inversion algorithm on two already locked-in matrices.

The operation will perform work for the next *duration* microseconds toward the complete inversion algorithm.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Outputs

Name	IEC 61131 Type	Description
error	BOOL	The matrix is not invertible.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the inversion completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Sets *result* to an identity matrix.
3. Reduces *original* to an identity matrix using elementary row operations.
4. Performs the same row operations on *result* to create the inverse.
5. If at any time the row operations cannot reduce *original* further and it is still not an identity matrix, the operation is terminated and *error* is set.
6. Performs work toward completing the algorithm, in groups of steps, until *duration* microseconds is exceeded.
7. Decrements *duration* by the microseconds consumed.
8. Returns TRUE if the inversion algorithm completed before *duration* was exhausted.
9. Returns FALSE if `LockMatrices()` has not been called or *duration* was exhausted before completing the algorithm.
10. The contents of *original* are destroyed by this method.

UnlockMatrices (Method)

This method unlocks any matrices locked by `LockMatrices()`. It only needs to be called by the user if the matrix operation has been terminated early by calling `Clear()` on any of the dependent matrices. In all other cases, the matrices will be unlocked on completion of the algorithm.

Processing

1. Requests that all locked matrices free themselves for other operations.
2. Sets *Busy* to FALSE.

class_MatrixMultiply (Class)

This class handles the locking handshakes and the state required to multiply two class_Matrix objects over the course of multiple scans.

Outputs

Name	IEC 61131 Type	Description
Busy	BOOL	This class has locked class_Matrix instances and is in the middle of a calculation.

LockMatrices (Method)

This method primes the class to perform a new multiply (result = matrix1 • matrix2). It must be called before each multiply to be performed.

Inputs/Outputs

Name	IEC 61131 Type	Description
matrix1	class_Matrix	The multiplicand.
matrix2	class_Matrix	The multiplier.
result	class_Matrix	The matrix to store the product.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the multiply operation is now ready.

Processing

1. Returns FALSE if either *matrix1* or *matrix2* is not initialized.
2. Returns FALSE if *result* is not a separate matrix from both *matrix1* and *matrix2*.
3. Returns FALSE if any of the matrices are busy doing any stepwise operation.

4. Returns FALSE if `Cols` of *matrix1* does not match `Rows` of *matrix2*.
5. Returns FALSE if *result* cannot be sized correctly to store the product.
6. Returns FALSE if it cannot lock all matrices involved in the operation.
7. If all other checks succeeded, then stores required references, sets *Busy* to TRUE, and returns TRUE.
8. The contents of *result* are destroyed by this method.

ProcessSteps (Method)

This method performs the multiplication algorithm on three already locked-in matrices.

The operation will perform the next *steps* sub-operations of the complete multiplication algorithm.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of sub-operations to attempt this task cycle.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the multiplication completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Multiplies *matrix1* by *matrix2*.
3. Decrements *steps* by the number of operations performed.
4. Returns TRUE if the multiply algorithm completed before *steps* was exhausted.
5. Returns FALSE if `LockMatrices()` has not been called or *steps* was exhausted before completing the algorithm.

ProcessTimed (Method)

This method performs the multiplication algorithm on three already locked-in matrices.

The operation will perform work for the next *duration* microseconds toward the complete multiplication algorithm.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the multiplication completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Multiplies *matrix1* by *matrix2*.
3. Performs work toward completing the multiplication algorithm, in groups of steps, until *duration* microseconds is exceeded.
4. Decrements *duration* by the microseconds consumed.
5. Returns TRUE if the multiply algorithm completed before *duration* was exhausted.
6. Returns FALSE if `LockMatrices()` has not been called or *duration* was exhausted before completing the algorithm.

UnlockMatrices (Method)

This method unlocks any matrices locked by `LockMatrices()`. It only needs to be called by the user if the matrix operation has been terminated early by calling `Clear()` on any of the dependent matrices. In all other cases, the matrices will be unlocked on completion of the algorithm.

Processing

1. Requests that all locked matrices free themselves for other operations.
2. Sets *Busy* to FALSE.

class_MatrixSubtract (Class)

This class handles the locking handshakes and the state required to subtract one class_Matrix object from another over the course of multiple scans.

Outputs

Name	IEC 61131 Type	Description
Busy	BOOL	This class has locked class_Matrix instances and is in the middle of a calculation.

LockMatrices (Method)

This method primes the class to perform a new subtraction (result = matrix1 – matrix2). It must be called before each subtraction to be performed.

Inputs/Outputs

Name	IEC 61131 Type	Description
matrix1	class_Matrix	The minuend.
matrix2	class_Matrix	The subtrahend.
result	class_Matrix	The difference of *matrix1 – matrix2*.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the subtraction operation is now ready.

Processing

1. Returns FALSE if either *matrix1* or *matrix2* is not initialized.
2. Returns FALSE if *result* is not a separate matrix from both *matrix1* and *matrix2*.
3. Returns FALSE any of the matrices are busy doing any stepwise operation.
4. Returns FALSE if the dimensions of *matrix1* do not match those of *matrix2*.
5. Returns FALSE if *result* cannot be resized to match the dimensions of the other two matrices.
6. Returns FALSE if it cannot lock all matrices involved in the operation.
7. If all other checks succeeded, then stores required references, sets *Busy* to TRUE, and returns TRUE.
8. The contents of *result* are destroyed by this method.

ProcessSteps (Method)

This method performs the subtraction algorithm on three already locked-in matrices.

The operation will perform the next *steps* sub-operations of the complete subtraction algorithm.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of sub-operations to attempt this task cycle.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the subtraction completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Subtracts each element in *matrix2* from its corresponding element in *matrix1* and the difference in *result*.
3. Decrements *steps* by the number of sub-operations performed.
4. Returns TRUE if the subtraction algorithm completed before *steps* was exhausted.
5. Returns FALSE if `LockMatrices()` has not been called or *steps* was exhausted before completing the algorithm.

ProcessTimed (Method)

This method performs the subtraction algorithm on three already locked-in matrices.

The operation will perform work for the next *duration* microseconds toward the complete subtraction algorithm.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the subtraction completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Subtracts each element in *matrix2* from its corresponding element in *matrix1* and stores the difference in *result*.
3. Performs work toward completing the subtraction algorithm, in groups of steps, until *duration* microseconds is exceeded.
4. Decrements *duration* by the microseconds consumed.
5. Returns TRUE if the subtraction algorithm completed before *duration* was exhausted.
6. Returns FALSE if `LockMatrices()` has not been called or *duration* was exhausted before completing the algorithm.

UnlockMatrices (Method)

This method unlocks any matrices locked by `LockMatrices()`. It only needs to be called by the user if the matrix operation has been terminated early by calling `Clear()` on any of the dependent matrices. In all other cases, the matrices will be unlocked on completion of the algorithm.

Processing

1. Requests that all locked matrices free themselves for other operations.
2. Sets *Busy* to FALSE.

class_MatrixTranspose (Class)

This class handles the locking handshakes and the state required to transpose a class_Matrix object over the course of multiple scans.

Outputs

Name	IEC 61131 Type	Description
Busy	BOOL	This class has locked class_Matrix instances and is in the middle of a calculation.

LockMatrices (Method)

This method primes the class to perform a new matrix transpose. It must be called before each transpose to be performed.

Inputs

Name	IEC 61131 Type	Description
conjugate	BOOL	The result of this operation will be the Hermitian Transpose. Before each element is placed in *result*, it will be conjugated.

Inputs/Outputs

Name	IEC 61131 Type	Description
original	class_Matrix	The matrix to copy from.
result	class_Matrix	The matrix to copy to.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the transpose operation is now ready.

Processing

1. Returns FALSE if *original* is not initialized.
2. Returns FALSE if *result* is not a separate matrix from *original*.
3. Returns FALSE if either of the matrices is busy doing any stepwise operation.
4. Returns FALSE if *result* cannot be resized to the correct dimensions.

5. Returns FALSE if it cannot lock all matrices involved in the operation.
6. If all other checks succeeded, then stores required references, sets *Busy* to TRUE, and returns TRUE.
7. The contents of *result* are destroyed by this method.

ProcessSteps (Method)

This method transposes an already locked-in matrix.

The operation will perform the next *steps* sub-operations of the complete matrix transpose.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of sub-operations to attempt this task cycle.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the transpose completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Copies each element (i, j) from *original* to element (j, i) of *result*.
3. If *conjugate* was TRUE, conjugate each element in *result*.
4. Decrements *steps* by the number of sub-operations performed.
5. Returns TRUE if the transpose algorithm completed before *steps* was exhausted.
6. Returns FALSE if `LockMatrices()` has not been called or *steps* was exhausted before completing the algorithm.

ProcessTimed (Method)

This method transposes an already locked-in matrix.

The operation will perform work for the next *duration* microseconds toward the complete matrix transpose.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the transpose completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Copies each element (i, j) from *original* to element (j, i) of *result*.
3. If *conjugate* was TRUE, conjugate each element in *result*.
4. Performs work toward completing the transpose, in groups of steps, until *duration* microseconds is exceeded.
5. Decrements *duration* by the microseconds consumed.
6. Returns TRUE if the transpose algorithm completed before *duration* was exhausted.
7. Returns FALSE if `LockMatrices()` has not been called or *duration* was exhausted before completing the algorithm.

UnlockMatrices (Method)

This method unlocks any matrices locked by `LockMatrices()`. It only needs to be called by the user if the matrix operation has been terminated early by calling `Clear()` on any of the dependent matrices. In all other cases, the matrices will be unlocked on completion of the algorithm.

Processing

1. Requests that all locked matrices free themselves for other operations.
2. Sets *Busy* to FALSE.

class_Matrix_ATA (Class)

This class handles the locking handshakes and the state required for an optimization of the operation (A^TA) transposing the input matrix and multiplying it by the original matrix over the course of multiple scans.

Outputs

Name	IEC 61131 Type	Description
Busy	BOOL	This class has locked class_Matrix instances and is in the middle of a calculation.

LockMatrices (Method)

This method primes the class to perform a new matrix operation A^TA. It must be called before each operation to be performed.

Inputs

Name	IEC 61131 Type	Description
conjugate	BOOL	The result of this operation will be calculated using the Hermitian Transpose. Before the transpose step is complete, each element will be conjugated.

Inputs/Outputs

Name	IEC 61131 Type	Description
original	class_Matrix	The matrix A.
result	class_Matrix	The matrix for the result.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the operation is now ready.

Processing

1. Returns FALSE if *original* is not initialized.
2. Returns FALSE if *result* is not a separate matrix from *original*.
3. Returns FALSE if either of the matrices is busy doing any stepwise operation.
4. Returns FALSE if *result* cannot be resized to the correct dimensions.
5. Returns FALSE if it cannot lock all matrices involved in the operation.
6. If all other checks succeeded, then stores required references, sets *Busy* to TRUE, and returns TRUE.
7. The contents of *result* are destroyed by this method.

ProcessSteps (Method)

This method computes A^TA of an already locked-in matrix.

The operation will perform the next *steps* sub-operations of the complete task.

Inputs/Outputs

Name	IEC 61131 Type	Description
steps	UDINT	The number of sub-operations to attempt this task cycle.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the operation completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Decrements *steps* by the number of sub-operations performed.
3. Returns TRUE if the algorithm completed before *steps* was exhausted.
4. Returns FALSE if `LockMatrices()` has not been called or *steps* was exhausted before completing the algorithm.

ProcessTimed (Method)

This method computes A^TA of an already locked-in matrix.

The operation will perform work for the next *duration* microseconds toward the complete matrix operation.

Inputs/Outputs

Name	IEC 61131 Type	Description
duration	UDINT	The number of microseconds to spend on this calculation.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the operation completed.

Processing

1. Validates that `LockMatrices()` has been successfully called.
2. Performs work toward completing the algorithm, in groups of steps, until *duration* microseconds is exceeded.
3. Decrements *duration* by the microseconds consumed.
4. Returns TRUE if the operation completed before *duration* was exhausted.
5. Returns FALSE if `LockMatrices()` has not been called or *duration* was exhausted before completing the algorithm.

UnlockMatrices (Method)

This method unlocks any matrices locked by `LockMatrices()`. It only needs to be called by the user if the matrix operation has been terminated early by calling `Clear()` on any of the dependent matrices. In all other cases, the matrices will be unlocked on completion of the algorithm.

Processing

1. Requests that all locked matrices free themselves for other operations.
2. Sets *Busy* to FALSE.

Benchmarks

Benchmark Platforms

The benchmarking tests recorded for this library are performed on the following platforms.

- SEL-3505
  - R134 firmware
- SEL-3530
  - R134 firmware
- SEL-3555
  - Dual-core Intel i7-3555LE processor
  - 4 GB ECC RAM
  - R134-V1 firmware

Benchmark Test Descriptions

Each benchmark is run on three different matrices: a 2 by 2, an 8 by 8, and a 64 by 64. All matrices used in the benchmarks are sized such that no memory allocations occur during the benchmark run.

fun_DeleteMatrix

The posted time is the average execution time of 100 consecutive calls when deleting a matrix.

fun_MatrixAdd

The posted time is the average execution time of 100 consecutive calls when adding two matrices.

fun_MatrixCopyColumn

The posted time is the average execution time of 100 consecutive calls when copying a column from one matrix to another.

fun_MatrixDeterminant

The posted time is the average execution time of 100 consecutive calls when operating on a valid invertable matrix.

fun_MatrixGaussianElim

The posted time is the average execution time of 100 consecutive calls when operating on a valid matrix that allows the algorithm to run to completion.

fun_MatrixInvert

The posted time is the average execution time of 100 consecutive calls when inverting a matrix.

fun_MatrixMultiply

The posted time is the average execution time of 100 consecutive calls when multiplying two matrices.

fun_MatrixSubtract

The posted time is the average execution time of 100 consecutive calls when subtracting two matrices.

fun_MatrixTranspose

The posted time is the average execution time of 100 consecutive calls when transposing a matrix.

fun_MatrixTranspose (Hermitian)

The posted time is the average execution time of 100 consecutive calls when calculating the Hermitian transpose of a matrix.

fun_Matrix_ATA

The posted time is the average execution time of 100 consecutive calls when calculating A^TA.

fun_Matrix_ATA (Hermitian)

The posted time is the average execution time of 100 consecutive calls when calculating A^TA when using the Hermitian transpose.

fun_NewMatrix

The posted time is the average execution time of 100 consecutive calls when allocating a new matrix.

Benchmark Results

Operation Tested	Platform (time in μs)		
	SEL-3505	SEL-3530	SEL-3555
fun_DeleteMatrix - 2x2	118	54	6
fun_DeleteMatrix - 8x8	105	49	4
fun_DeleteMatrix - 64x64	119	57	4
fun_MatrixAdd - 2x2	17	7	1
fun_MatrixAdd - 8x8	66	46	4
fun_MatrixAdd - 64x64	5 408	3 299	244
fun_MatrixCopyColumn - 2x2	7	2	1
fun_MatrixCopyColumn - 8x8	10	3	1
fun_MatrixCopyColumn - 64x64	61	36	1
fun_MatrixDeterminant - 2x2	69	41	3
fun_MatrixDeterminant - 8x8	1 118	665	53
fun_MatrixDeterminant - 64x64	515 458	292 846	26 267
fun_MatrixGaussianElim - 2x2	76	39	2
fun_MatrixGaussianElim - 8x8	1 392	731	61
fun_MatrixGaussianElim - 64x64	516 370	296 721	26 789
fun_MatrixInvert - 2x2	74	48	3
fun_MatrixInvert - 8x8	2 092	1 243	105
fun_MatrixInvert - 64x64	1 034 455	581 496	52 648
fun_MatrixMultiply - 2x2	26	15	2
fun_MatrixMultiply - 8x8	1 002	626	51
fun_MatrixMultiply - 64x64	554 947	325 479	26 534
fun_MatrixSubtract - 2x2	13	6	1
fun_MatrixSubtract - 8x8	90	41	4
fun_MatrixSubtract - 64x64	5 341	3 066	242
fun_MatrixTranspose - 2x2	8	4	1
fun_MatrixTranspose - 8x8	29	12	1
fun_MatrixTranspose - 64x64	2 447	1 731	71
fun_MatrixTranspose (Hermitian) - 2x2	14	7	1
fun_MatrixTranspose (Hermitian) - 8x8	97	48	3
fun_MatrixTranspose (Hermitian) - 64x64	6 071	3 699	192
fun_Matrix_ATA - 2x2	26	15	1
fun_Matrix_ATA - 8x8	563	318	29
fun_Matrix_ATA - 64x64	308 482	179 945	13 186
fun_Matrix_ATA (Hermitian) - 2x2	34	19	1

Operation Tested	Platform (time in μs)		
	SEL-3505	SEL-3530	SEL-3555
fun_Matrix_ATA (Hermitian) - 8x8	877	455	39
fun_Matrix_ATA (Hermitian) - 64x64	431 940	239 407	17 341
fun_NewMatrix - 2x2	340	98	10
fun_NewMatrix - 8x8	87	39	5
fun_NewMatrix - 64x64	744	327	13

Examples

These examples demonstrate the capabilities of this library. Do not mistake them as suggestions or recommendations from SEL.

Implement the best practices of your organization when using these libraries. As the user of this library, you are responsible for ensuring correct implementation and verifying that the project using these libraries performs as expected.

Solving a System of Equations

Objective

The user desires to repeatably solve a system of equations for some set of outputs. This example solves three equations for three unknowns.

For example, on a given scan the system of equations could be:

$$\begin{cases} x + 2y + 3z = 2 \\ 2x + 3y + z = 2 \\ 3x + 2y + z = 10 \end{cases}$$

This becomes a 3x4 matrix which, after Gaussian elimination, appears as follows:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 2 \\ 2 & 3 & 1 & 2 \\ 3 & 2 & 1 & 10 \end{array}\right] => \left[\begin{array}{ccc|c} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & -3 \\ 0 & 0 & 1 & 1 \end{array}\right]$$

By inspection the solution becomes:

$$\begin{cases} x = 5 \\ y = -3 \\ z = 1 \end{cases}$$

Assumptions

Each scan the user has placed the values to use into a pair of arrays of struct_ComplexRect objects, Values and Answers, before this program is called.

Solution

The user can call this program each scan to receive a solution for the provided inputs, as shown in *Code Snippet 1*.

Code Snippet 1 prg_MatrixSolver

```
PROGRAM prg_MatrixSolver
VAR
    (* Here are sample values to generate a matrix with a known solution
       [[ 1 2 3 | 2 ]
        [ 2 3 1 | 2 ]
        [ 3 2 1 | 10 ]] *)
    Values : ARRAY [0 .. 8] OF struct_ComplexRect :=
        [(Re := 1), (Re := 2), (Re := 3),
         (Re := 2), (Re := 3), (Re := 1),
         (Re := 3), (Re := 2), (Re := 1)];
    AnswerCol : ARRAY [0 .. 2] OF struct_ComplexRect :=
        [(Re := 2), (Re := 2), (Re := 10)];
    // The result should be [5, -3, 1]
    Solution : ARRAY [0 .. 2] OF struct_ComplexRect;

    CoefficientsMatrix : class_Matrix(3, 3);
    SolutionsMatrix : class_Matrix(3, 1);
    pt_Data : POINTER TO POINTER TO struct_ComplexRect;
    Unsolved : BOOL;

    Row : UINT;
    Col : UINT;
END_VAR
```

```
//Load each row of the matrix
FOR Row := 0 to CoefficientsMatrix.Rows - 1 DO
    pt_Data := CoefficientsMatrix.pt_Data;
    //Load all but the answer column of the matrix
    FOR Col := 0 TO CoefficientsMatrix.Cols - 1 DO
        pt_Data[Row][Col] := Values[Row*(CoefficientsMatrix.Cols) + Col];
    END_FOR
    //Load the answer column after the final increment above
    pt_Data := SolutionsMatrix.pt_Data;
    pt_Data[Row][0] := AnswerCol[Row];
END_FOR

fun_MatrixGaussianElim(CoefficientsMatrix, SolutionsMatrix, error =>
    Unsolved);
FOR Row := 0 to SolutionsMatrix.Rows - 1 DO
    //If a solution was successfully found update the solution array.
    IF NOT Unsolved THEN
        pt_Data := SolutionsMatrix.pt_Data;
        Solution[Row] := pt_Data[Row][0];
    END_IF
END_FOR
```

Manipulating a Matrix Across Multiple Scans

Objective

The user needs to manipulate a set of data in matrix form, but has some set of timing constraints that cause concern regarding the completion of the operations.

Assumptions

The user has created an enumeration to assist in managing the data flow to the desired outcome.

Code Snippet 2 enum_States

```
TYPE enum_States :
(
    IDLE,
    BUILD_MATRICES,
    SUM_MATRICES,
    SCALE_RESULT,
    STORE_RESULT,
    ERROR
);
END_TYPE
```

Solution

The user can call this program each scan, as shown in *Code Snippet 3*, to receive a solution for the provided inputs. When *Begin* is set to true, the calculation will commence. When the program has copied the answer into *Solution*, the program sets the *Complete* flag to true.

Code Snippet 3 prg_MatrixManipulation

```
PROGRAM prg_MatrixManipulation
VAR CONSTANT
    c_StepsPerScan : UDINT := 5;
END_VAR
VAR
    State : enum_States := IDLE;

    Values1 : ARRAY [0 .. 11] OF struct_ComplexRect;
    Values2 : ARRAY [0 .. 11] OF struct_ComplexRect;
    Solution : ARRAY [0 .. 11] OF struct_ComplexRect;

    Matrix1 : class_Matrix(4, 3);
    Matrix2 : class_Matrix(4, 3);
    MatrixEnd : class_Matrix(4, 3);
    Adder : class_MatrixAdd;
    Scalar : struct_ComplexRect := (Re := 2, Im := 0);
    pt_Data1 : POINTER TO POINTER TO struct_ComplexRect;
    pt_Data2 : POINTER TO POINTER TO struct_ComplexRect;

    Row : UINT;
    Col : UINT;
    Steps : UDINT;
    Scans : UDINT;
    Begin : BOOL;
    Complete : BOOL;
END_VAR
```

Code Snippet 3 prg_MatrixManipulation (Continued)

```
Steps := c_StepsPerScan;
Scans := Scans + 1;
WHILE Steps > 0 DO
    CASE State OF
    IDLE:
        IF Begin THEN
            State := BUILD_MATRICES;
            Row := 0;
            Col := 0;
            Scans := 1;
            pt_Data1 := Matrix1.pt_Data;
            pt_Data2 := Matrix2.pt_Data;
            Begin := FALSE;
            Complete := FALSE;
            Steps := Steps - 1;
        ELSE
            Scans := Scans - 1;
            Steps := 0;
        END_IF
    BUILD_MATRICES:
        //This is a state machine to load the matrix a few values at a time
        Steps := Steps - 1;
        pt_Data1[Row][Col] := Values1[Row*(Matrix1.Cols) + Col];
        pt_Data2[Row][Col] := Values2[Row*(Matrix1.Cols) + Col];
        Col := Col + 1;
        IF Col = Matrix1.Cols THEN
            Col := 0;
            Row := Row + 1;
            IF Row = Matrix1.Rows THEN
                IF Adder.LockMatrices(Matrix1, Matrix2, MatrixEnd) THEN
                    State := SUM_MATRICES;
                ELSE
                    State := ERROR;
                END_IF
            END_IF
        END_IF
    SUM_MATRICES:
        IF Adder.ProcessSteps(steps) THEN
            IF MatrixEnd.StartMatrixOperation(MATRIX_SCALE, 0, 0, Scalar)
                THEN
                State := SCALE_RESULT;
            ELSE
                State := ERROR;
            END_IF
        END_IF
    SCALE_RESULT:
        IF MatrixEnd.MatrixStepScale(steps) THEN
            State := STORE_RESULT;
            Row := 0;
            Col := 0;
        END_IF
```

Code Snippet 3 prg_MatrixManipulation (Continued)

```
    STORE_RESULT:
        Steps := Steps - 1;
        Solution[Row*(MatrixEnd.Cols) + Col] := MatrixEnd.pt_Data[Row][Col];
        Col := Col + 1;
        IF Col = MatrixEnd.Cols THEN
            Col := 0;
            Row := Row + 1;
            IF Row = MatrixEnd.Rows THEN
                State := IDLE;
                Complete := TRUE;
                Steps := 0;
            END_IF
        END_IF
    ERROR:
        Steps := 0;
    END_CASE
END_WHILE
```

Troubleshooting a Matrix

Objective

The user has designed a solution with matrices to perform some set of calculations and something is not going as desired. The user would like to have additional insight into the matrix element values for online troubleshooting.

Assumptions

This solution assumes a static matrix size. This is not required but if the *Rows* and *Cols* variables of the matrix do not match the sizes provided for the troubleshooting variable, the user must realize that only data up to the size of the matrix are valid.

Solution

The user can add an additional pointer variable to provide additional insight during runtime. The syntax for this pointer is shown in *Code Snippet 4*.

Code Snippet 4 prg_MatrixTroubleshoot

```
PROGRAM prg_MatrixTroubleshoot
VAR CONSTANT
    c_Rows : UINT := 2;
    c_Cols : UINT := 6;
END_VAR
VAR
    Values1 : ARRAY [0 .. 11] OF struct_ComplexRect;

    Matrix1 : class_Matrix(c_Rows, c_Cols);

    (*This is the troubleshooting variable that has been added.
      To be valid it must be reassigned each time the memory allocated to
      the matrix could change, so the safest usage is to assign it
          immediately
      before using it.*)
    pt_Raw : POINTER TO ARRAY [0 .. c_Rows-1] OF
             POINTER TO ARRAY [0 .. c_Cols-1] OF struct_ComplexRect;
END_VAR
```

```
//Load the matrix
(*The SysMemCpy command allows the movement of large quantities of
    contiguous
  data with a single instruction. This can greatly increase the performance
  of large data copies. If the destination and the source could overlap
  then the SysMemMove call facilitates this with a little more overhead.*)
SysMemCpy(Matrix1.pt_Data[0], ADR(Values1),
    c_Cols*SIZEOF(struct_ComplexRect));
SysMemCpy(Matrix1.pt_Data[1], ADR(Values1[6]),
    c_Cols*SIZEOF(struct_ComplexRect));

(*Here is where we find some meaningful work up to the point of interest
    for troubleshooting.*)

//Assign the troubleshooting variable. Now the data can be seen in
//online mode.
//This line is where a breakpoint would be added.
pt_Raw := Matrix1.pt_Data;

(*There is probably additional work to be accomplished after the point of
    interest as well*)
```

Release Notes

Version	Summary of Revisions	Date Code
3.5.1.1	➤ Allows new versions of ACSELERATOR RTAC to compile projects for previous firmware versions without SEL IEC types “Cannot convert” messages. ➤ Must be used with R143 firmware or later.	20180921
3.5.0.1	➤ Initial release.	20150722