XIRTAM Language Proposal

Annie Wang (aw3168) - manager
Andrew Peter Yevsey Gorovoy (apg2165) - manager/architect (environment selection)
Shida Jing (sj2670) - tester
Bailey Nozomu Hwa (bnh2128) - language
Lior Attias (lra2135) - architect (compiler design)

1. Overview

The XIRTAM language is a statically typed and compiled language focused on the manipulation of matrix (XIRTAM) objects. The purpose of XIRTAM is to provide functionality akin to Python's numpy matrix manipulation module in addition to flexible matrix operations not possible in traditional languages. The overall syntax is quite similar to C. XIRTAM is statically typed and forces the user to write the types out in declarations and then the compiler will validate those types. XIRTAM models inherent matrix data as instance variables existent on the matrix object itself. This object-oriented approach to matrix representation allows users of XIRTAM to quickly get useful mathematical information about their matrix that would otherwise require computation. This information allows XIRTAM further optimize performance by choosing the most computationally efficient matrix operation algorithms based on matrix size. Our language aims to create a language which is as functionally convenient as numpy, syntactically more streamlined, and allows for more flexible matrix design. XIRTAM is ideal for:

- Neural Networks: Unlike traditional arrays, XIRTAM objects are mutable. This makes it easy to increase and decrease the size of matrixes for operations such as building mutable neural networks, which can be cumbersome in traditional languages like C++
- Pure Mathematical Manipulations: XIRTAM's powerful built-in methods makes realizing and experimenting with matrix related theorems and algorithms easy.

2. Language Details

2.1 Data Types and Operations

Data Type	Description:	Operations (unary and binary):	Examples:
numeric	Catch-all data type for numbers. So there is no issue with float matrices vs int. By default a float.	=, ==, !=, +, -, *, /, %, ++, –, >>, <<, +=, -=, <, >, =<, >=	num a = 1; num a = 1.0;
char	A character type, 1 byte	=, ==, !=, +, ++, –, <, >, =<, >=	char a = "h";
string	An immutable array of chars	=, ==, !=, <, >, =<, >= (lexicographical), + (concatenate)	String s1 = "hello";

bool	A boolean literal, with value either true or false	=, ==, !=, !, &&, ||	x = true ! x // Returns false
int	Integral type - 4 bytes	=, ==, !=, +, -, *, /, %, ++, –, >>, <<, +=, -=, <, >, =<, >=	a1 < a2; a1 = a1 + 7;
float	Float type - 8 bytes	=, ==, !=, +, -, *, /, %, ++, –, +=, - =, <, >, =<, >=	f1 == f2; float myFloat = 36.6;
matrix	Matrices are represented as XIRTAM objects, the following operations are all inherent in XIRTAM objects or built in as static functions	See section 3	See section 3

2.2 Keywords

The following words are reserved in the XIRTAM language. The reserved words parallel reserved words found in C.

if, else – Used for decision control programming structure.

break – Used with any loop OR switch case.

int, float, char, numeric, matrix – These are the data types and used during variable declaration.

for, while – types of loop structures in C.

void – One of the return type

return – This keyword is used for returning a value.

continue – It is generally used with for, while and do while loops, when the compiler encounters this statement it performs the next iteration of the loop, skipping rest of the statements of the current iteration.

enum – Set of constants.

sizeof – It is used to know the size.

struct, typedef – Both of these keywords used in structures (Grouping of data types in a single record).

2.3 Control Flow

The following keywords are reserved for control flow: **if**..**else**, **for**, **while**
The control flow will follow C-style if..else blocks, for loops, and while loops

```
for (num i = 0; i< 5; i++){
      if ((i % 2) ==0){
            print("even");
      }
}
```

2.4 Functions

XIRTAM supports function declarations in a standard C style with

```
return_type function_name(parameter list) {
      body of the function
}
```

For example, to write a function that adds two integers in XIRTAM it looks like this:

```
num foo (num a, num b){
      return (a+b);
}
```

Functions can be declared before the actual body of the function is defined. Functions can be called after a function declaration and before the body of the function is defined.

```
num foo (num a, num b);

num main (){
      num a = 100;
      num b = 10.5;
      num result;

      result = foo(a,b);

      return result;
}

num foo (num a, num b){
      return (a+b);
}
```

2.5 Comments

Comments also follow C's syntax where a single-line comment is denoted with (//) and a multi-line comments are denoted with /* */ as shown below:

```
// This is a single line comment

/* This
is
A
multiline
comment */
```

2.6 Memory

For XIRTAM specific data objects and functions, memory is handled on the backend and no memory management is needed by the user . If the user wants to use XIRTAM in conjunction with C or C linked libraries, normal C rules should be followed (i.e., when needed, the user should Malloc and realloc memory as they would do naturally in C). For XIRTAM objects and functions, there will not be an automated garbage collector.

3. XIRTAM Specifics and Example Code

3.0 XIRTAM basics

XIRTAM is strongly and statically typed. The purpose of XIRTAM is to allow for flexible matrix operations that are not possible in traditional languages. For these reasons, XIRTAM forces the user to write the types out in declarations (and then the compiler will validate those types on compile-time) and does not automatically convert between types or provide generic object types. Please note that while XIRTAM provides a matrix data type with rich built-in functionality, the purpose of XIRTAM is not to be a general object oriented language.

3.1 XIRTAM instance data

XIRTAM models inherent matrix data as instance variables existent on the matrix object itself. This approach to matrix representation allows users of XIRTAM to quickly get useful mathematical information about their matrix that would otherwise require computation.

These data types can be accessed via the XIRTAM object directly. To illustrate these we will provide some examples assuming the following sample matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
Xirtam matrix // assume this matrix has already been populated
int rows = matrix.rows;  //returns 3
int cols = matrix.cols;  //returns 3
int [] dimensions = matrix.dimensions;   //returns [3, 3]
int [] l_diagonal = matrix.left_diagonal;  //returns [1,5,9]
int [] r_diagonal = matrix.right_diagonal; //returns [3,5,7]
int [] col_1 = matrix.cols(1);    //returns [1, 4, 7]
int [][] cols_1_2 = matrix.cols(1:2)); //returns columns 1 through 2
inclusive
int [] rows_1 = matrix.rows(1);   //returns [1 2 3]
int [][] rows_2:3 = matrix.rows(2:3); // returns a 2D array rows 2
through 3 inclusive
int [][] sub_matrix = matrix.dims(1:2, 2:3); // returns a 2D array
representing {{2, 3}, {5, 6}} (the intersection of rows 1 - 2 and
cols 2 - 3)
int max = matrix.max_val;
int min = matrix.min_val;
```

3.2 XIRTAM transformations and discussion of shape flexibility

The XIRTAM language also provides for standard library operations that are not tied to a single data type. One example is the creation of XIRTAM objects through several different avenues. The example below shows the ability to create a XIRTAM object directly from another matrix (in which the XIRTAM object is a subset of the larger matrix). Alternatively, a XIRTAM object can also be instantiated from an N-D array of any size (1-D, 2-D, 3-D, etc.).

Below, the asXIRTAM built in is meant to act similar to a 'toString(var int)' method. The asXIRTAM function converts its parameters to a Xirtam object.

```
Xirtam small_matrix = asXIRTAM(matrix.cols(1:2));
Xirtam sub_matrix = asXIRTAM(matrix.dims(1:2, 2:3)); //returns a new
matrix of rows 1 through 2 incl., and columns 2 through 3 incl. {{2,
3}, {5, 6}}
Xirtam s_matrix = asXIRTAM(cols_1_2);
```

Unlike traditional arrays, XIRTAM objects are mutable. This makes it easy to increase and decrease the size of matrixes for operations such as building mutable neural networks, which can be cumbersome in traditional languages like C++.

For flexible design of neural networks, XIRTAM allows users to build non-square matrices using the `without_error_detection` functionality.

```
Xirtam matrix; // assume populated with the original values as above
matrix.append_row_values(0); // populates a row at the bottom of the
matrix containing all zeros
matrix.prepend_row_values(1); // populates a row at the bottom of the
matrix containing all ones
matrix.append_col_values(1); // populates a column to the left of the
matrix of all 1s
matrix.prepend_col_values(0); // populates a column to the right of
the matrix of all 1's
Xirtam matrix2 = {{2,2,2}, {2,2,2}};
matrix.add(matrix2, "top");
matrix.add(matrix2, "bottom");
matrix.overalay_with_error_detection(matrix2, overide_rows 1);
matrix.overaly_without_error_detectoin(matrix2, overide_rows 1);
//allows for purposeful creation of non-square matrices
matrix_add_without_error_detection(matrix2, "left");
amatrix_add_without_error_detection(matrix2, "right");
```

XIRTAM matrices can contain any type of data. The default type of a XIRTAM matrix holds numerics. However, XIRTAM matrices can hold any type of data (doubles, chars, strings, longs, floats, etc.) which can be specified by the user at initialization. Once the type of the XIRTAM matrix is set, it is not possible to modify the type contained in the matrix. Additionally, while XIRTAM objects are mutable and can be appended with other XIRTAM objects, arrays, and matrixes, it is not possible to append sub-matrices, sub arrays, or sub XIRTAM objects of different types to the original XIRTAM object.

```
Xirtam int_matrix = new matrix(); // a default, int containing matrix.
This matrix starts out empty
Xirtam char_matrix = new matrix("char");
Xirtam string_matrix = new matrix("string");
Xirtam xirtam_matrix = new matrix("xirtam");
Xirtam array_matrix = new matrix("array");
Xirtam list_matrix = new matrix ("list");
```

A note on `min_val` and `max_val`-- if the matrix holds string or char values, the highest ascii value argument is denoted as the max value. For matrixes that hold objects, min and max values must be programmed into the object itself by the user. For XIRTAM matrixes that hold other XIRTAM matrixes, min and max values are a global min and max.

3.3 XIRTAM object initialization

XIRTAM objects can be initialized from a char-delimited file (such as a CSV or space separated file, where each new line (carriage return) represents a new row. XIRTAM objects can also be instantiated directly with a matrix using { { }, { } } notation. XIRTAM objects can also be instantiated via default dimensions and values, including NULL values. Finally, XIRTAM matrices can be instantiated via “special type” such as `identity` .

```
Xirtam matrix = new matrix_from_file (filepath, ','); // char delimiter
Xirtam matrix = new matrix_from_file (filepath, "string_delimiter");
Xirtam matrix = new matrix({ {1, 2, 3}, {4, 5, 6}, {7, 8, 9} });
Xirtam matrix = new matrix(2, 3, NULL); // a 2 row, 3 column matrix
containing NULL values
Xirtam matrix = new matrix(2, 3, 0); // a 2 row, 3 column matrix
containing the int value 0

Xirtam matrix = new matrix("identity"); //the identity matrix
```

As discussed above, XIRTAM objects can also be instantiated via any n-dimensional array via a XIRTAM transformation.

```
int [] 1D = {1, 2, 3, 4};
int [] [] 2D = {{1, 2, 3, 4}, 1};
Xirtam matrix = asXIRTAM(1D);
Xirtam matrix = asXIRTAM(2D);
```

3.3 XIRTAM internal transformations

The following operations allow for internal data representations of XIRTAM objects

```
Xirtam matrix;
matrix.remove(0); // replaces all 0 values in the matrix with NULL, for
example for JPEG compression
matrix.remove(0:10); // replaces all values 0 - 10 in the matrix with
NULL
matrix. remove([0, 10, 11, 12]); // replaces specific values the
matrix with NULL
matrix.replace(0, 10); // replaces all 0 values with 10
matrix.replace(0:10, 100); // replace all 0 through 10 (inclusive)
values in the matrix with the value 100

matrix.replace_with_gradient(value:value range end, gradient:
gradient range end);
matrix.replace_with_gradient(0:10, 25:100); // replaces all values 0 -
10 with an even gradient of values 25 to 100 in ascending order based
on the amount of values being replaced
```

3.4 Standard Library Functions

XIRTAM (in all caps) is used to denote functions that are static functions— in other words functions that do not belong to a specific Xirtam object. For example, XIRTAM.multiply( ) is a built in method that takes in two independent Xirtam objects and returns a third Xirtam object. Here, multiple is not resident on a specific Xirtam object. Contrast this with the syntax for class methods, such as Xirtam m = new(); m.remove(0). Here, the remove function is resident on and acts on the Xirtam object m.

XITAM language has several pre-built functions that return a new XIRTAM object, with the parameters unmodified. These include:

a. Addition
b. Matrix multiplication
c. Tensor Product
d. Transpose
e. Conjugate transpose
f. Check to see if the columns/rows are orthogonal
g. Inverse of square matrix
h. Determinant of matrix
i. Reshaping Matrix
j. Reduced Row Echelon Form

XIRTAM writes such expressions as shown below:

```
Xirtam matrix1;
Xirtam matrix2;
Xirtam matrix_addition = XIRTAM.add(matrix1, matrix2);
Xirtam matrix_multipication = XIRTAM.multiply(matrix1, matrix2);
Xirtam matrix_tensor = XIRTAM.tensor(matrix1, matrix2);
Xirtam matrix_transpose = XIRTAM.transpose(matrix1);
Xirtam conjucate_transpose = XIRTAM.conjugate_transpose(matrix1);
bool orthogonal_rows = XIRTAM.is_matrix_orthogonal(matrix1, "rows");
bool orthogonal_cols = XIRTAM.is_matrix_orthogonal(matrix1, "cols");
XIRTAM.determinant(matrix1);
Xirtam translated_matrix = XIRTAM.translate(matrix1);
Xirtam rotated_matrix_90 = XIRTAM.rotate(matrix1, 90); // rotate by 90
degrees. The only possible arguments are 90 or 180)
Xirtam reflected_matrix = XIRTAM.reflect(matrix1, "x-axis"); //
reflects on the x or y axis
Xirtam reshaped_matrix = XIRTAM.reshape(matrix1);
Xirtam inverse_sq = XIRTAM.inverse_of_square(matrix1);
Xirtam rreo_form = XIRTAM.reduced_row_echelon_form(matrix1);
```

3.5 Example XIRTAM Programs

3.5.1 Fibonacci Sequence

Generating the nth number of a general Fibonacci Sequence. Parameters: a_1, a_0 are the starting values of the sequence.

```
num nthFibo(num num n, num num a_1, num num a_0){
      Xirtam m = new matrix({ {1, 1}, {1, 0} });
      Xirtam exp_m = new matrix("identity");
      for (num i = 0; i< n; i++){
            exp_m = XIRTAM.multiply(new_m, m);
      }
      Xirtam init = new matrix({ {a_1}, {a_0} });
      Xirtam result = XIRTAM.multiply(exp_m, init);
      return result[1];
}
```

3.5.2 Entangled Qubits

The function postMeasurementState takes in a pair of entangled qubits expressed in terms of a vector in C^4 and a measurement result expressed in terms of a basis. It returns the post-measurement state of the qubit pair if the first qubit is measured and the result is the result vector provided.

```
Xirtam postMeasurementState(Xirtam entangled_state, Xirtam result){
      Xirtam identity_matrix = new matrix("identity");
      Xirtam projectioin_matrix = XIRTAM.multiply(result,
XIRTAM.conjugate_transpose(result));
      Xirtam post_meas_state =
XIRTAM.multiply(XIRTAM.tensor(projection_matrix, identity_matrix),
entangled_state);
      return post_meas_state;
}
```

3.5.3 Total Positivity

Check for total positivity. A minor of a matrix A is a square matrix which is formed by selecting a subset of the rows of A, then selecting a subset of the columns of A, then taking the entries from the intersection. A matrix is totally positive if all of its minors have positive determinant. If we have a 2 by n matrix A and we want to check for total positivity, there are (n choose 2) many non-trivial minors to check. However, a theorem states that we only need to check 2n-3 many minors to conclude total positivity.

```
bool 2bynTotalPositivity(Xirtam m){
      bool isTotallyPositive = True;

      for (int i = 0; i < 2 * m.cols - 3; i++){
            list_of_cols = [m.cols(0), m.cols(i)];
            Xirtam minor = asXIRTAM(list_of_cols);
            isTotallyPositive = isTotallyPositive &&
(XIRTAM.determinant(minor) > 0);
      }
      return isTotallyPositive;
}
```

Total positivity is important in **algebraic combinatorics** but historically it stems from real analysis, because totally positive matrices have real distinct positive eigenvalues, and therefore they are diagonalizable.