SLang - the Next Generation

Module TMATH

Christian Bucher, Sebastian Wolff
Center of Mechanics and Structural Dynamics
Vienna University of Technology

May 28, 2010

Contents

1 Module tmath

1.1 Overview

The module `tmath` provides data types and algorithms for basic linear algebra. It was attempted to provide a natural access to the mathematical grammar, thus, merging the programming idioms of Lua, C++ and arithmetic languages like MATLAB (TM).

The package is, however, embedded into the language Lua and is, thus, dependent on its logic and syntax. Therefore, sometimes the syntax appears uncommon. For example, any object has a certain datatype. Functions can only be applied to objects of specific datatypes. The same is true for operators and methods which are tied to their lefthand argument. For example, the "*" operator is attached to its left hand neighbour. In programming languages, the terms "A*B" and "A:operator*(B)" are equivalent. Since we can define operators for our own datatypes, but not for Lua's internal "number" type, we can provide the operator "matrix*number", but not "number*matrix".

1.2 Dense linear algebra

1.2.1 Creating matrices

There are several ways to create matrices. The simpliest way is to call its constructor

```
A = tmath.Matrix(3,4) -- create 3x4 matrix
B = tmath.Matrix(3)   -- create 3x1 vector (of type Matrix)
C = tmath.Vector(3)   -- create 3x1 vector (of type Matrix)
```

Calling the constructor will only create and return an object of type "Matrix" of the specified size, but without initializing the values. Although the uninitialized values are around zero on most computers, initial values must be assigned by a separate command, e.g.

```
A = tmath.Matrix(3,4)
value = 2.1;
value1 = 1;
value2 = 2;
A:SetZero()                    -- zero matrix
A:SetOnes()                    -- all elements are "1"
A:SetConstant(value)           -- all elements are ="value"
A:SetIdentity()                -- (rectangular) identity matrix
A:SetLinearCols(value1,value2)  -- linear columns from value1 to value2
A:SetLinearRows(value1,value2)  -- linear rows
A:SetRandom()                  -- random numbers (0..1)
```

To simplify creation and initialization convenience functions are defined such as

```
A = tmath.Identity(3) -- 3x3 identity matrix
A = tmath.ZeroMatrix(3,4)
B = tmath.ZeroVector(4)
```

It is also possible to read the contents of matrices from input

```
A = tmath.Matrix(2,3)
tmath.Read(A,
      1,2,3,
      4,5,6);
-- even better (using 2-dimensional Lua tables as input)
B = tmath.Matrix(
    {{1,2,3},
     {4,5,6}}
     );
```

When defining a matrix by Lua tables, it is possible to combine existing matrix objects. These objects will be interpreted as row vectors:

```
A = tmath.ZeroVector(4);
B = tmath.Identity(2);
C = tmath.Matrix({
  A,           -- 0 0 0 0
  {5,6,7,8},   -- 5 6 7 8
  B            -- 1 0 0 1
  });
```

1.2.2 Assigning values

Assigning values to objects may differ in various programming languages. In C++ and MATLAB the contents of an object is copied into the other. In Lua, however, only a new identifier is created for the right hand object, i.e. Lua's assign command

```
A = tmath.ZeroVector(3) -- create a new Matrix object and assign it to ident "A"
B = A                   -- assign the object behind "A" to the ident "B"
```

will create the identifier "B" which refers to the same matrix object as "A" (The first command will create a Matrix object on the right hand side and assign it to the identifier "A" on its left side).

As long as new objects will be created on the right side, the assign operator is equal to what is known from C++ , i.e.

```
A = tmath.Identity(3)
B = A*(-3) + tmath.Identity(3)
```

Herein, the arithmetic operators always create and return new temporary objects of type Matrix. The last created object will then be assigned to the identifier "B".

If the value should be assigned (and not the object itself) then `tmath` provides copy constructors for many data types, i.e.

```
A = tmath.ZeroVector(3) -- create a new Matrix object and assign it to ident "A"
B = tmath.Matrix(A)     -- create a new Matrix object which has equal content with "A
    " and assign it to ident "B"
```

There are some case, where the "=" operator is not applicable. Then the only way to copy data is to use the "Assign" method. This may happen if you want to assign a value to a matrix which is part of another userdata object. For example, there is a finite element object which stores a force vector and gives access to it via a referencing method:

```
force = fem_object:RestoringForce() -- call the method to return a reference to a
    Matrix object stored in "fem_object"
force = tmath.Matrix(fem_object:RestoringForce()) -- create a copy of the internal
    force vector
B = tmath.Vector(force:Rows())
fem_object:RestoringForce() = B -- will produce an error because the left side is a
    userdata
fem_object:RestoringForce():Assign( B ) -- will copy the contents from "B" to "
    fem_object:RestoringForce()"
```

For such cases, the Matrix class is equipped with the method "Assign" which directly assigns the given value to itself.

A very fast way to transfer data is the "Swap" method which swaps the pointer to the data buffer of two Matrix objects:

```
A = tmath.Identity(3)
B = tmath.ZeroVector(4)
B:Swap(A); -- "A" will now be a zero vector, "B" is an identity matrix
```

1.2.3 Matrix blocks

`tmath` provides the data type MatrixBlock which is a view on parts of existing Matrix objects. MatrixBlock does not have its own data buffer, but it behaves like an independent Matrix object providing its own arithmetic operators, set methods, index operators, etc. A matrix block is created by and can be used as, for example

```
A = tmath.Identity(4)
col = A:Col(2) -- create a matrix block for the 3rd column of A
row = A:Row(1) -- create a matrix block for the 2nd row of A
mat = A:Block(1,0,2,2) -- create a block of size 2x2, starting at {1,0}
mat = tmath.MatrixBlock(A,1,0,2,2) -- does the same


mat:SetOnes() -- sets all elements of "A" in the index range {1..2,0..1} to "1"
B = col*2.5 -- create a matrix from an arithmetic operation with the 3rd column of A
A:Block(2,1,2,2):Row(1):SetOnes() -- set the elements of "A" at {3,1} and {3,2} to
    "1"
```

Blocks use a reference counting system regarding the parent matrix. That is, the parent matrix will not be garbage collected as long as any block is referencing it.

1.2.4 Matrix keys

Keys provide elementwise access to the data buffer of matrices and matrix blocks. Unlike in Lua where table indices range from `1` to `#table`, `tmath` uses indices ranging from `0` to `#buffer-1`, i.e. from `{0,0}` to `{Rows()-1,Cols()-1}`. There exist two index operators: Either your key denotes the position in the internal one-dimensional column-major data buffer, or it denotes a two-dimensional matrix index. The one-dimensional key n will be translated into matrix notation $\{i, j\}$ via the relation $n = i + j * Rows()$. The key operator can be used to get and set elements of a matrix or matrix block:

```
A = tmath.Identity(3)
A[{2,1}] = 2 -- sets the element at {2,1} to "2"
print(A[5]) -- prints the element at {2,1} (="2")
```

It is also possible to insert the contents of matrices and blocks:

```
A = tmath.ZeroMatrix(4,4);
B = tmath.Vector(2);
B[{0,0}] = 1;
B[{1,0}] = 2;
A[{1,1}] = B;
print(A)
--[[
0       0       0       0
0       1       0       0
0       2       0       0
0       0       0       0
--]]
```

Herein, the given index is the topleft position where the matrix will be inserted.

For matrices, the size of the matrix will be automatically increased if the inserted block exceeds its size. This resize operation is allowed for appending rows to vectors and for appending columns for arbitrarily shaped matrices. Appending a row to a matrix with column-major storage format is always an expensive operation because resizing requires a reordering of matrix elements. It, therefore, is not allowed.

```
-- append row to vector
A = tmath.ZeroVector(4);
A[1] = 1;
B = tmath.Vector(2);
B[{0,0}] = 1;
B[{1,0}] = 2;
A[{3,0}] = B;
print("Matrix: ",A:Rows(),A:Cols());
print(A:Transpose())
--[[
Matrix:         5       1
0 1 0 1 2
--]]
```

```
-- append column to matrix
A = tmath.ZeroMatrix(4,4);
B = tmath.ZeroVector(4)
B[{0,0}] = 1;
B[{1,0}] = 2;
A[{0,4}] = B;
print("Matrix: ",A:Rows(),A:Cols());
print(A)
--[[
Matrix:         4       5
0       0       0       0       1
0       0       0       0       2
0       0       0       0       0
0       0       0       0       0
--]]
```

```
-- insert a block which will increase number of columns of a matrix (including some
    white space)
A = tmath.ZeroMatrix(4,3);
B = tmath.Matrix(2,2);
B[{0,0}] = 1;
B[{1,0}] = 2;
B[{0,1}] = 3;
B[{1,1}] = 4;
A[{1,2}] = B;
print("Matrix: ",A:Rows(),A:Cols());
print(A)
--[[
Matrix:          4       4
0       0       0       0
0       0       1       3
0       0       2       4
0       0       0       0
--]]
```

1.2.5 Component wise operations

The data type MatrixCWise provides component wise operations to matrices. Objects of this kind are created by

```
A = tmath.Identity(4);
cw = A:CW() -- short hand notation
cw = tmath.MatrixCWise(A) -- creation using constructor
```

They denote a scalar view onto all elements of the parent matrix. The class provides numerous scalar functions as builtin methods, for example

- trigonometric: Sin, Cos, Atan2
- scalar arithmetic operators: +-*/
- powers: operator ^, Square, Cube, Sqrt, Exp, Log
- auxiliary functions: absolute value (Abs, Abs2), Min, Max, Sign
- etc.

There also exists a way to apply any scalar valued function $y = f(x)$ to a matrix component wise:

```
-- Define some scalar function
function f(x)
  return x^2+1;
end
A = tmath.Identity(4); -- create a matrix
B = tmath.CWise(A,f) -- apply the user defined function f(x) component wise to "A"
print(B)
```

1.2.6 Arithmetic operators

Multiplication, special products and powers

```
-- define objects:
A = tmath.ZeroMatrix(3,4)
B = tmath.ZeroMatrix(4,2)
s = 1
-- operators:
C = A*B      -- matrix product (vectors are considered as matrices)
B = A*s      -- product of a matrix with a scalar
A:Mul(s)     -- inplace product with a scalar (like A=A*s, but faster)
```

```
-- define objects:
A = tmath.Identity(4,4)
B = tmath.Identity(4,4)
v = tmath.ZeroVector(4)
w = tmath.ZeroVector(4)
s = 1
-- operators:
s = tmath.Dot(v,w)          -- scalar product of 2 vectors s=v'*w
s = v:Dot(w)                -- scalar product of 2 vectors s=v'*w
s = tmath.Dott(A,v)         -- weighted scalar product s=v'*A*v
C = tmath.OuterProd(A,B)    -- outer matrix product C=A*B'
C = tmath.InnerProd(A,B)    -- inner matrix product C=A'*B
B = tmath.MatrixEigenSym(A)^s      -- matrix power A^s of symmetric A
B = tmath.MatrixEigenSym(A):Exp() -- matrix exponential exp(A) of symmetric A
```

```
-- define objects:
A = tmath.Identity(4,4)
B = tmath.Identity(4,4)
s = 1
-- operators:
C = A:CW()*B        -- cwise matrix product
A:CW():Mul(B)       -- cwise inplace matrix product (A = A:CW()*B)
B = A:CW()^s        -- componentwise power (B_ij = (A_ij)^s)
B = A:CW():Exp()    -- componentwise expinential (B_ij = exp(A_ij))
```

Division and inversion

```
-- define objects:
A = tmath.Identity(4,4)
s = 1
-- operators:
B = A/s                 -- matrix divided by scalar
A:Div(s)              -- inplace division by a scalar
B = tmath.Inverse(A)    -- returns inverse matrix using LU decomposition
X = tmath.Solve(A,B)    -- solves A*X=B using LU decomposition
```

```
-- define objects:
A = tmath.Identity(4,4)
B = tmath.Identity(4,4)
s = 1
-- operators:
C = A:CW()/B          -- componentwise scalar division of 2 matrices
A:CW():Div(B)       -- componentwise scalar inplace division A_ij = A_ij/B_ij
B = A:CW():Inverse() -- componentwise reciprocal of matrices
```

Sums and differences

```
-- define objects:
A = tmath.Identity(4,4)
s = 1
-- operators:
B = -A              -- unary minus
C = A+B             -- matrix sum
C = A-B
A:Add(B)            -- inplace matrix sum A=A+B
A:Sub(B)
B = A:CW()+s        -- adds a scalar to all coefficients of A
B = A:CW()-s
A:CW():Add(s)       -- inplace componentwise sum with a scalar
A:CW():Sub(s)
```

Binary operators Binary operators usually return scalar values being either true or false. Applied to real numbers, any false value is represented by the number zero. Any true value is not zero. On return, true will be represented by "1".

```
-- define objects:
A = tmath.Identity(4,4)
B = tmath.Identity(4,4)
s = 1
-- operators:
--ERROR! C = (A:CW() < B)  -- componentwise "<" (less) of 2 matrices or a scalar
--ERROR! C = A:CW() < s
--ERROR! C = A:CW() <= B -- componentwise "<=" (less or equal) of 2 matrices or a
    scalar
--ERROR! C = A:CW() <= s
C = A:CW() == B -- componentwise "==" (equal) of 2 matrices or a scalar
C = A:CW() == s
```

1.2.7 Properties

Matrix and MatrixBlock provide a few methods to test the type of the matrix, i.e.

```
-- define objects:
A = tmath.Identity(4,4)
B = tmath.Identity(4,4)
s = 1
-- operators:
A:IsApprox(B, 1e-7)            -- is A approximately equal to B (tolerance 1e-7)?
A:IsApproxToConstant(s,1e-10)
A:IsDiagonal(1e-10)
A:IsIdentity(1e-10)
A:IsLowerTriangular(1e-10)
A:IsUnitary(1e-10)             -- is it unitary (orthonormal basis)?
A:IsUpperTriangular(1e-10)
A:IsVector()
A:IsScalar()
A:IsZero(1e-10)
```

All methods return a boolean value.

1.2.8 LU decomposition

Let A be a square matrix. An LU decomposition is a decomposition of the form

$$\mathbf{A} = \mathbf{LU}$$

where L and U are a lower and an upper triangular matrix. For example, the LU decomposition of a 3×3 matrix writes

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

`tmath` provides the data type MatrixLU which performs LU decomposition. Actually, the methods tmath.Solve and tmath.Inverse use this data type internally. On creation, the input matrix will be factorized. After that, the LU object provides methods to compute the inverse (not recommended), solve a system of linear equations or compute rank and determinant.

```
-- define objects:
A = tmath.Identity(4,4)
b = tmath.ZeroVector(4)
-- operators:
solver = tmath.MatrixLU(A)
print("rank(A) " .. solver:Rank())
print("det(A)  " .. solver:Determinant())
x = solver:Solve(b)
inv = solver:Inverse()
```

1.2.9 Cholesky decomposition

If A is a real symmetric (Hermitian) and positive definite matrix, then A can be uniquely decomposed as

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T$$

where L is a lower triangular matrix with strictly positive diagonal entries, and L^T denotes the conjugate transpose of L. This is the standard Cholesky decomposition.

The standard Cholesky decomposition is error-prone if the the matrix A is ill-conditioned or not positive definite. The reason is the application of square roots which can be avoided using the (slower, but) stable Cholesky decomposition

$$\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} D_1 & 0 & 0 \\ 0 & D_2 & 0 \\ 0 & 0 & D_3 \end{bmatrix} \begin{bmatrix} 1 & L_{21} & L_{31} \\ 0 & 1 & L_{32} \\ 0 & 0 & 1 \end{bmatrix}$$

When A is positive definite the elements of the diagonal matrix D are all positive. The factorization can be applied to any square symmetric matrix (though the inversion can not be applied).

`tmath` provides the data type MatrixLDLt. It can be used as

```
-- define objects:
A = tmath.Identity(4,4)
b = tmath.ZeroVector(4)
-- operators:
solver = tmath.MatrixLDLt(A)
if ( solver:IsPositive() ) then
  print("A is positive (semi)definite")
end
if ( solver:IsNegative() ) then
  print("A is negative (semi)definite")
end
x = solver:Solve(b) -- solves A*x=b
solver:SolveInPlace(b) -- solves A*x=b and sets b=x
L = solver:MatrixL() -- returns matrix L
D = solver:VectorD() -- returns matrix D as a vector
```

1.2.10 Eigenvalue problems

Given a linear transformation A, a non-zero vector x is defined to be an eigenvector of the transformation if it satisfies the eigenvalue equation

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

for some scalar λ. Herein, the scalar λ is called an eigenvalue of A corresponding to the eigenvector x.

A generalized eigenvalue problem is given by the equation

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x}$$

with positive definite matrix B.

The spectral theorem for matrices can be stated as follows. Let A be a square $n \times n$ matrix. Let $q_1 \dots q_k$ be an eigenvector basis, i.e. an indexed set of k linearly independent eigenvectors, where k is the dimension of the space spanned by the eigenvectors of A. If $k = n$, then A can be written

$$\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$$

where Q is the square $n \times n$ matrix whose i-th column is the basis eigenvector i of A and Λ is the diagonal matrix containing the corresponding eigenvalues.

- If the matrix A is real and symmetric, then all eigenvalues are real.
- If the matrix A is positive definite, then all eigenvalues are positive.
- For the standard symmetric eigenvalue problem, i.e. $B = I$, the eigenvectors are orthogonal, i.e. $\mathbf{Q}^T = \mathbf{Q}^{-1}$
- For the generalized symmetric eigenvalue problem the eigenvectors can be normalized such that, i.e. $\mathbf{Q}\mathbf{B}\mathbf{Q}^T = \mathbf{I}$

Nonsymmetric eigenvalue problem

```
-- define objects:
A = tmath.Identity(4,4)
-- operators:
solver = tmath.MatrixEigenUnsym(A) -- create an eigenvalue solver object and factorize
U = solver:PseudoEigenvalueMatrix() -- returns vector of real block diagonal eigenvalues D
V = solver:PseudoEigenvectors() -- returns the matrix of the pseudo eigenvectors V ( such that A*V=V*D)
```

Symmetric eigenvalue problem

```
-- define objects:
A = tmath.Identity(4,4)
B = tmath.Identity(4,4)*2
s = 1
-- operators:
solver = tmath.MatrixEigenSym(A) -- create an eigenvalue solver object and factorize, also compute eigenvectors
solver = tmath.MatrixEigenSym(A,false) -- create an eigenvalue solver object and factorize, but do not compute eigenvectors
solver = tmath.MatrixEigenSym(A,B,false) -- create a generalized eigenvalue solver object and factorize, but do not compute eigenvectors
solver = tmath.MatrixEigenSym(A,B) -- create a generalized eigenvalue solver object and factorize, also compute eigenvectors
if (solver:EigenvectorsAvailable()) then
  print("Eigenvector have been computed.")
end
if (solver:IsGeneralEigenproblem()) then
  print("It is a generalized problem.")
end
--[[
v = solver:Eigenvalues()
Q = solver:Eigenvectors()
C = solver:OperatorInverseSqrt()  -- returns the positive inverse square root of the matrix (if positive definite)
C = solver:OperatorSqrt() -- returns the positive square root of the matrix (if positive definite)
C = solver^s -- returns the matrix power of A (by scalar exponent), C=A^{s}
C = solver:Pow(s)
C = solver:Exp() -- returns the matrix exponential of A, C=e^{A}
C = solver:Log() -- returns the matrix logarithm of A, C=ln(A)
--]]
```

1.2.11 SVD

Suppose A is an m-by-n matrix. Then there exists a factorization of the form

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T,$$

where U is an m-by-m unitary matrix, the matrix Σ is m-by-n diagonal matrix with nonnegative real numbers on the diagonal, and V^T denotes the conjugate transpose of V, an n-by-n unitary matrix. This factorization is called a singular-value decomposition of A.

It is common to sort the diagonal entries $\Sigma_{i,i}$ in non-increasing order. In this case, the diagonal matrix Σ is uniquely determined by A (though the matrices U and V are not). The diagonal entries of Σ are known as the singular values of A.

- The columns of V form a set of orthonormal "input" vectors for A. (eigenvectors of $A^T A$.)
- The columns of U form a set of orthonormal "output" vectors for A. (eigenvectors of AA^T.)
- The diagonal values in matrix Σ are the singular values, by which each corresponding input is multiplied to give a corresponding output. (square roots of the eigenvalues of A times A.)

```
-- define objects:
A = tmath.Identity(4,4)
w = tmath.ZeroVector(4)
-- operators:
svd = tmath.MatrixSVD(A)      -- create a SVD object and factorize matrix A
v = svd:Solve(w)              -- solves the system Av=w and returns v
U = svd:MatrixU()             -- returns matrix U
V = svd:MatrixV()             -- returns matrix V
sigma = svd:SingularValues() -- returns singular values as a vector
```

1.2.12 Functions

A great variety of auxiliary functions are provided within the namespace `tmath`. Please take a look at the API reference for details.

1.3 Sparse linear algebra

Sparse matrices are matrices which contain many zero elements when compared with the number of nonzero entries. Therefore, special memory layouts which do not save the zero elements explicitely may be advantageous regarding required memory and efficiency. TNG supports a few default types and algorithms for sparse matrices which are explained in this section.

The number of available methods is limited compared with dense algrithms. This is because sparse matrices are often used in limited use cases, i.e. for example solving large systems of equations. It is assumed, that they are usually created by some third-party module. Some arithmetic operators are implemented, though.

1.3.1 SparseMatrix

The type SparseMatrix is a sparse matrix with column-major memory layout. Sparse matrices are matrices where only the nonzero coefficients are stored, that is the storage format has to provide the value and the position (row, column) of a coefficient. SparseMatrix follows the CCS scheme (Compressed Column Storage), that is there will be 3 vectors:

- 'values': a vector of all nonzero coefficient values, they are ordered by columns
- 'inner indices': a vector of the row positions of all nonzero coefficients
- 'outer indices'. a vector that indicates at which position in the vector 'values' a column is starting.

Therefore, iterating through a column is fast (and extracting/appending a column), but iterating through a row may be slow. Also the filling of a matrix must be done in the predefined filling order according to the storage layout, that is ideally column after column. Within each column random and ordered fill (regarding the row index) is allowed.

1.3.2 Arithmetic operations

The following operators are defined:

```
-- define objects:
A = tmath.SparseIdentity(4,4)
B = tmath.SparseIdentity(4,4)
v = tmath.ZeroVector(4)
s = 1
-- operators:
u = A*v; -- returns the product of a sparse matrix and a dense matrix/vector
B = A*s; -- returns the product of a sparse matrix with a scalar
B = A/s; -- returns the quotient of a sparse matrix with a scalar
C = A+B; -- returns the sum of two sparse matrices
C = A-B; -- returns the difference of two sparse matrices
s = tmath.Dott(A,v) -- returns the weighted scalar product of a sparse matrix with a
    dense vector s = v'Av
```

Also check the list of matrix functions.

1.3.3 Properties

```
-- define objects:
A = tmath.SparseIdentity(4,4)
i,j = 2,2
s = 1e-7
-- operators:
A:Rows() -- returns the number of rows
A:Cols() -- returns the number of columns
A:NonZeros()      -- returns the number of nonzero coefficients
A:InnerNonZeros() -- returns the number of nonzero coefficients in the j-th columns
A:Resize(i,j)     -- resizes the dimensions
A:SetZero()       -- clears all nonzeros
A:Prune(s)        -- erases all coefficients below the given treshold
```

1.3.4 DynamicSparseMatrix

DynamicSparseMatrix has been introduced to allow random read and write access to all (nonzero) elements of a sparse matrix. The access is quite fast, but operations are slow. Therefore, it is only used as a transfer type which should be converted to SparseMatrix once the matrix has been filled.

1.3.5 SymSparseMatrix

SymSparseMatrix denotes a sparse matrix class which represents square symmetric matrices. Only one half of the nonzero coefficients is stored.

1.3.6 DynamicSymSparseMatrix

DynamicSymSparseMatrix has been introduced to allow random read and write access to all (nonzero) elements of a symmetric sparse matrix. Its recommended use is to fill a symmetric matrix in random order by DynamicSymSparseMatrix which should be converted to SymSparseMatrix once the matrix has been filled.

1.3.7 SparseSolver

SparseSolver is the base class for a variety of algorithms that solve linear systems involving sparse matrices. Therefore, the interface of these solvers is unified. Following data types are based on (derived from) SparseSolver:

- LL' solver for symmetric matrices: a basic implementation, backend Cholmod
- LU solver for quadratic matrices: a basic implementation, backend MUMPS(recommended), backend SuperLU, backend UmfPack

```
-- define objects:
A = tmath.SparseIdentity(4,4)
b = tmath.Vector(4)
b:SetConstant(2)
c = tmath.Matrix(b) -- copy b to c
-- operators:

-- create a solver object, here: MUMPS
solver = tmath.MUMPS()

-- compute the factorization of sparse matrix A
if (not solver:Compute(A)) then
  error("Error during factorization.")
end

-- solve Ax=b for x
x,succeeded = solver:Solve(b)
if (not succeeded) then
  error("Error during solution")
end

-- solve Ax=c for x and store the result in c
succeeded = solver:SolveInPlace(c)
```

1.3.8 SparseArpack

The data type SparseArpack encapsulates methods for computing eigenvalue problems using the backend ARPACK. At the moment, symmetric standard and generalized eigenproblems are supported.

Create an ARPACK object.

```
eigen = SparseArpack()
```

Change precision of the iterative eigen solver

```
eigen:SetPrecision(s)
```

Change the allowed maximum number of iterations

```
eigen:SetMaxIterations(n)
```

Create a sparse matrix, set its matrix type, create a SparseSolver object and set its options (precision, solver flags, etc.).

Call one of the algorithms of SparseArpack. For example, if one wants to compute the 20 smallest eigenvalues (being greater than 0) of a generalized symmetric eigenvalue problem using shift-invert transformation, one writes

```
succeeded = eigen:ShiftInvert(A,B,SparseSolver(),0,20)
```

The object stores the result:

```
print(eigen:Eigenvalues())
print(eigen:Eigenvectors())
```