BERNARD BARON, ARTUR PASIERBEK, MARCIN POŁOMSKI, RADOSŁAW SOKÓŁ
SILESIAN UNIVERSITY OF TECHNOLOGY, POLAND

OBJECT-ORIENTED SPARSE MATRIX IMPLEMENTATION IN C++

INTRODUCTION

Many computational algorithms store their intermediate or final results in a form of a matrix. While matrices are well known among programmers, they tend to consume a lot of memory. At the same time, many algorithms require or generate matrices that are very sparsely populated (Fig. 1). This is where sparse matrix concept comes to help.

Sparse matrices are well documented in various publications. However, most such publications concentrate on internal structures used by various sparse matrices implementations, and not on the code itself. This work presents a working C++ implementation of sparse matrix programming concept, built as a self-contained C++ class ready to be integrated into any software package.

AIMS

Ease of use

The sparse matrix C++ class should be straightforward to use.

Math friendliness

While computer languages such as C++ tend to use zero-based indexes, world of science is used to one-based indexes.

Error resistance

The class being presented protects from wrong indexing.

Performance

Special attention was paid to making inserting new matrix cells as efficient as possible.

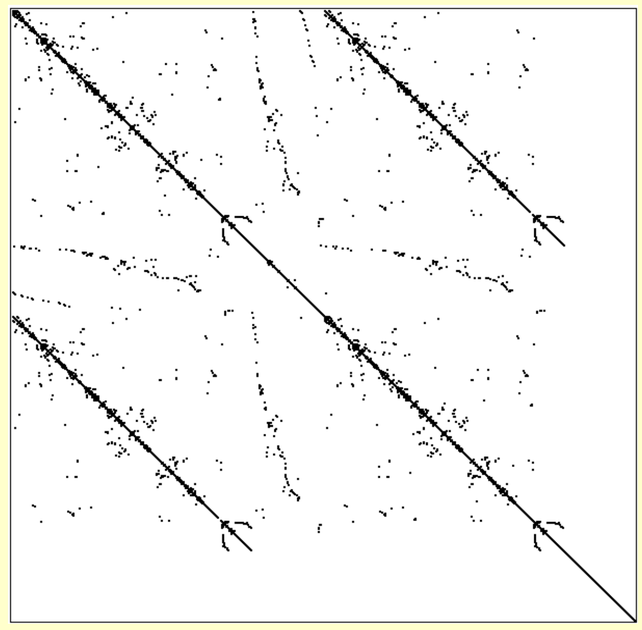

Fig. 1. A typical sparse matrix; here, a Jacobi matrix of constraints of a power system

EXAMPLES

To create a new sparse matrix object, one has only to define a new variable, specifying data type and number of rows and columns:

```
sparse_matrix<double> Matrix(10, 20); // 10 rows, 20 columns, 'double' type
```

Filling the matrix with data is as easy as accessing individual cells and assigning new values to them:

```
for (unsigned int r = 1; r <= Matrix.Rows(); ++r) {
   for (unsigned int item = 0; item < 5; ++item {
      unsigned int c = rand() % (Matrix.Columns() + 1);
      Matrix(r, c) = (double) rand() / 10.0;
   }
}
```

Reading data from the matrix may be performed as above, accessing individual cells given their coordinates and checking return value (on read-only access, the sparse matrix class does not create new cells). However, for most applications, the preferred way of accessing sparse matrix data is to parse nonempty cells only, retrieving their contents and coordinates:

```
sparse_matrix_element<double> c;
for (unsigned int r = 1; r <= Matrix.Rows(); ++r) {
   unsigned int NonZero = Matrix.NumNonZeroElements(r);
   for (unsigned int i = 0; i < NonZero; ++i) {
      c = Matrix.ReadNonZeroElement(r, i);
      printf("There is a non-empty cell in row %u col %u "
         "with a value of %f.\n", r, c.Coordinate, c.Value);
   }
}
```

REAL-WORLD EXAMPLE: Creating a sparse admittance matrix for a power system

```
sparse_matrix<double> SystemAdmittanceMatrix()
{
   complex<double> Yij;
   unsigned int StartNodeIdx, EndNodeIdx;
   sparse_matrix<double> Y = sparse_matrix<double> (Coord.Nw(), Coord.Nw());
   unsigned int i;
   const Node *N;
   for (i = 1; i <= Coord.Nw(); ++i) {
      N = (const Node*) &(Coord.Nodes.ByGunn(i));
      Y(i,i) = complex<double>(N->Gs, N->Bs);
   }
   for (i = 1; i <= Coord.Lines.Size(); ++i) {
      const Line &L = Coord.Lines.ByGuln(i);
      StartNodeIdx = L.StartNode;
      EndNodeIdx = L.EndNode;
      Yij = 1.0 / complex<double>(L.R, L.X);
      if (L.IsTransformer()) {
         const complex<double> t = L.theta * complex<double>(cos(L.psi), sin(L.psi));
         Y(StartNodeIdx,EndNodeIdx) -= Yij / t;
         Y(EndNodeIdx,StartNodeIdx) -= Yij / conj(t);
         Y(StartNodeIdx,StartNodeIdx) += Yij / sqr(L.theta);
         Y(EndNodeIdx,EndNodeIdx) += Yij;
      } else {
         Y(EndNodeIdx,StartNodeIdx) -= Yij;
         Y(StartNodeIdx,EndNodeIdx) -= Yij;
         const complex<double> gb = Yij + complex<double>(L.G, L.B);
         Y(StartNodeIdx,StartNodeIdx) += gb;
         Y(EndNodeIdx,EndNodeIdx) += gb;
      }
    }
   return Y;
}
```

PERFORMANCE

A lot of effort has been put to ensure that the sparse matrix C++ class performs as efficiently as possible. Using it incurs only a slight performance penalty, while ensuring that no indexing errors occur.

Tab. 1 and Fig. 2 present performance results (in seconds) for sparse linear equation solve problems for equations set size from 50 000 to 1 000 000 equations and variables.

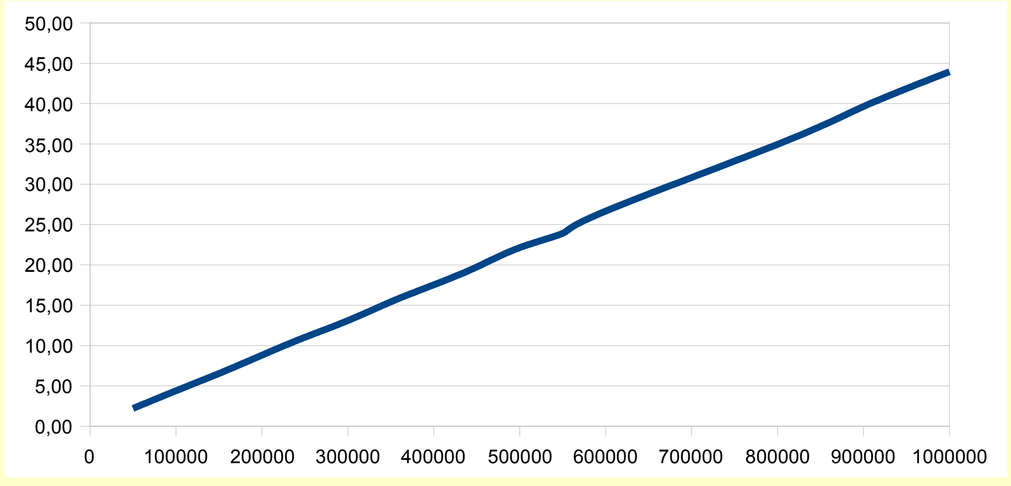

Fig. 2. Linear equations solution times for 10 000 - 1 000 000 equations/variables

Eqs	Time
50000	2,20
100000	4,39
150000	6,53
200000	8,80
250000	11,02
300000	13,09
350000	15,42
400000	17,53
450000	19,74
500000	22,14
550000	23,92
600000	26,67
800000	34,95
900000	39,64
1000000	43,95

Tab. 1. Times (seconds) table for Fig. 2

CLASS INTERFACE

```
typedef unsigned int uint;

template <class T> struct sparse_matrix_element
{
   uint Coordinate;
   T Value;
   sparse_matrix_element();
   sparse_matrix_element(const uint, const T);
};

template <class T> class sparse_matrix
{
   public:

   sparse_matrix(const uint rows, const uint columns);
   sparse_matrix(const sparse_matrix &m);
   ~sparse_matrix() throw();
   sparse_matrix<T> & operator = (const sparse_matrix<T>&);

   void SetAllocationDelta(uint) const throw();
   uint GetAllocationDelta() const throw();

   uint Columns() const throw();
   uint Rows() const throw();
   uint SizeOf() const throw();

   void Clear() throw();
   const T operator () (uint Row, uint Column) const;
   T & operator () (uint Row, uint Column);
   bool operator == (const sparse_matrix<T> &m) const throw();
   bool operator != (const sparse_matrix<T> &m) const throw();
   uint NumNonZeroElements() const;
   uint NumNonZeroElements(uint row) const;
   sparse_matrix_element<T> ReadNonZeroElement
                                    (uint row, const uint i) const;
   void DeleteNonZeroElement(uint row, const uint i);
   T DiagonalElement(uint row) const;
   uint DiagonalElementIndex(uint row) const;
   sparse_matrix<T> Transpose() const;
   void Optimize(const T Threshold);

   void InsertRows(uint Row, uint NumRows);
   void InsezrtRow(uint Row);
   void AddRows(uint NumRows);
   void AddRow();
   void InsertColumns(uint Column, uint NumColumns);
   void InsertColumn(uint Column);
   void AddColumns(uint NumColumns);
   void AddColumn();
   void DeleteRows(uint Row, uint NumRows);
   void DeleteRow(uint Row);
   void DeleteColumns(uint Column, uint NumColumns);
   void DeleteColumn(uint Column);

   private:

   uint w, h;
   struct rowdesc {
      uint num_nonzero;
      uint capacity;
      uint diagindex;
      sparse_matrix_element<T> el[];
      uint column_search(const uint zero_based_column) throw();
      uint insertion_pos(const uint zero_based_column) throw();
   } **rows;

   static struct rowdesc * new_row() throw();
   void upsize_row(const uint rindex);

};
```

CONCLUSION

The sparse_matrix class has been implemented and developed as a part of a large power system optimization grant project. It contributed to tremendously reduced development time and error rate, eliminating the need to open-code sparse matrix accesses in every subroutine. It also helped in error finding, thanks to its range checking code. Paired with sparse sub-matrix helper code (not discussed here) it made it possible to divide large sparse matrix generation process into small, independent sub-matrices, eventually merged together with no indexing errors.


Prof. Bernard BARON
Bernard.Baron@polsl.pl


Artur PASIERBEK, PhD
Artur.Pasierbek@polsl.pl


Tomasz KRASZEWSKI, MSc
Tomasz.Kraszewski@polsl.pl


Marcin POŁOMSKI, MSc
Marcin.Polomski@polsl.pl


Radosław SOKÓŁ, MSc
Radoslaw.Sokol@polsl.pl

Silesian University of Technology
http://www.polsl.pl/

Faculty of Electrical Engineering
http://www.elektr.polsl.pl/

Institute of Industrial Electrical Engineering and Informatics
http://ietip.elektr.polsl.pl/