Advanced initialization

Dense matrix and array manipulation

This page discusses several advanced methods for initializing matrices. It gives more details on the comma-initializer, which was introduced before. It also explains how to get special matrices such as the identity matrix and the zero matrix.

The comma initializer

Eigen offers a comma initializer syntax which allows the user to easily set all the coefficients of a matrix, vector or array. Simply list the coefficients, starting at the top-left corner and moving from left to right and from the top to the bottom. The size of the object needs to be specified beforehand. If you list too few or too many coefficients, **Eigen** will complain.

Example:

```
Matrix3f m;
m << 1, 2, 3,
     4, 5, 6,
     7, 8, 9;
std::cout << m;
```

Output:

```
1 2 3
4 5 6
7 8 9
```

Moreover, the elements of the initialization list may themselves be vectors or matrices. A common use is to join vectors or matrices together. For example, here is how to join two row vectors together. Remember that you have to set the size before you can use the comma initializer.

Example:

```
RowVectorXd vec1(3);
vec1 << 1, 2, 3;
std::cout << "vec1 = " << vec1 << std::endl;

RowVectorXd vec2(4);
vec2 << 1, 4, 9, 16;
std::cout << "vec2 = " << vec2 << std::endl;

RowVectorXd joined(7);
joined << vec1, vec2;
std::cout << "joined = " << joined << std::endl;
```

Output:

```
vec1 = 1 2 3
vec2 =  1  4  9 16
joined =  1  2  3  1  4  9 16
```

We can use the same technique to initialize matrices with a block structure.

Example:

```
MatrixXf matA(2, 2);
matA << 1, 2, 3, 4;
MatrixXf matB(4, 4);
matB << matA, matA/10, matA/10, matA;
std::cout << matB << std::endl;
```

Output:

```
  1   2 0.1 0.2
  3   4 0.3 0.4
0.1 0.2   1   2
0.3 0.4   3   4
```

The comma initializer can also be used to fill block expressions such as `m.row(i)`. Here is a more complicated way to get the same result as in the first example above:

Example: **Output:**

```
Matrix3f m;
m.row(0) << 1, 2, 3;
m.block(1,0,2,2) << 4, 5, 7, 8;
m.col(2).tail(2) << 6, 9;
std::cout << m;
```

```
1 2 3
4 5 6
7 8 9
```

Special matrices and arrays

The **Matrix** and **Array** classes have static methods like **Zero()**, which can be used to initialize all coefficients to zero. There are three variants. The first variant takes no arguments and can only be used for fixed-size objects. If you want to initialize a dynamic-size object to zero, you need to specify the size. Thus, the second variant requires one argument and can be used for one-dimensional dynamic-size objects, while the third variant requires two arguments and can be used for two-dimensional objects. All three variants are illustrated in the following example:

Example:

```
std::cout << "A fixed-size array:\n";
Array33f a1 = Array33f::Zero();
std::cout << a1 << "\n\n";


std::cout << "A one-dimensional
       dynamic-size array:\n";
ArrayXf a2 = ArrayXf::Zero(3);
std::cout << a2 << "\n\n";


std::cout << "A two-dimensional
       dynamic-size array:\n";
ArrayXXf a3 = ArrayXXf::Zero(3, 4);
std::cout << a3 << "\n";
```

Output:

```
A fixed-size array:
0 0 0
0 0 0
0 0 0

A one-dimensional dynamic-size array:
0
0
0

A two-dimensional dynamic-size array:
0 0 0 0
0 0 0 0
0 0 0 0
```

Similarly, the static method **Constant**(value) sets all coefficients to `value`. If the size of the object needs to be specified, the additional arguments go before the `value` argument, as in `MatrixXd::Constant(rows, cols, value)`. The method **Random()** fills the matrix or array with random coefficients. The identity matrix can be obtained by calling **Identity()**; this method is only available for **Matrix**, not for **Array**, because "identity matrix" is a linear algebra concept. The method **LinSpaced**(size, low, high) is only available for vectors and one-dimensional arrays; it yields a vector of the specified size whose coefficients are equally spaced between `low` and `high`. The method `LinSpaced()` is illustrated in the following example, which prints a table with angles in degrees, the corresponding angle in radians, and their sine and cosine.

Example:

```
ArrayXXf table(10, 4);
table.col(0) = ArrayXf::LinSpaced(10,
       0, 90);
table.col(1) = M_PI / 180 * table.col(0);
table.col(2) = table.col(1).sin();
table.col(3) = table.col(1).cos();
std::cout << "  Degrees   Radians
       Sine    Cosine\n";
std::cout << table << std::endl;
```

Output:

```
  Degrees   Radians      Sine    Cosine
        0         0         0         1
       10     0.175     0.174     0.985
       20     0.349     0.342      0.94
       30     0.524       0.5     0.866
       40     0.698     0.643     0.766
       50     0.873     0.766     0.643
       60      1.05     0.866       0.5
       70      1.22      0.94     0.342
```

```
        80         1.4       0.985       0.174
        90        1.57           1 -4.37e-08
```

This example shows that objects like the ones returned by LinSpaced() can be assigned to variables (and expressions). **Eigen** defines utility functions like **setZero()**, **MatrixBase::setIdentity()** and **DenseBase::setLinSpaced()** to do this conveniently. The following example contrasts three ways to construct the matrix $J = \left[\begin{smallmatrix} O & I \\ I & O \end{smallmatrix} \right]$: using static methods and assignment, using static methods and the comma-initializer, or using the setXxx() methods.

Example:

```
const int size = 6;
MatrixXd mat1(size, size);
mat1.topLeftCorner(size/2, size/2)     = MatrixXd::Zero(size/2, size/2);
mat1.topRightCorner(size/2, size/2)    = MatrixXd::Identity(size/2, size/2);
mat1.bottomLeftCorner(size/2, size/2)  = MatrixXd::Identity(size/2, size/2);
mat1.bottomRightCorner(size/2, size/2) = MatrixXd::Zero(size/2, size/2);
std::cout << mat1 << std::endl << std::endl;

MatrixXd mat2(size, size);
mat2.topLeftCorner(size/2, size/2).setZero();
mat2.topRightCorner(size/2, size/2).setIdentity();
mat2.bottomLeftCorner(size/2, size/2).setIdentity();
mat2.bottomRightCorner(size/2, size/2).setZero();
std::cout << mat2 << std::endl << std::endl;

MatrixXd mat3(size, size);
mat3 << MatrixXd::Zero(size/2, size/2), MatrixXd::Identity(size/2, size/2),
        MatrixXd::Identity(size/2, size/2), MatrixXd::Zero(size/2, size/2);
std::cout << mat3 << std::endl;
```

Output:

```
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0

0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0

0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
```

A summary of all pre-defined matrix, vector and array objects can be found in the **Quick reference guide**.

Usage as temporary objects

As shown above, static methods as Zero() and Constant() can be used to initialize variables at the time of declaration or at the right-hand side of an assignment operator. You can think of these methods as returning a matrix or array; in fact, they return so-called **expression objects** which evaluate to a matrix or array when needed, so that this syntax does not incur any overhead.

These expressions can also be used as a temporary object. The second example in the **Getting started** guide, which we reproduce here, already illustrates this.

Example:

```
#include <iostream>
#include <Eigen/Dense>

using namespace Eigen;
using namespace std;

int main()
{
  MatrixXd m = MatrixXd::Random(3,3);
```

Output:

```
m =
  94 89.8 43.5
49.4  101 86.8
88.3 29.8 37.8
m * v =
404
```

```
  m = (m + MatrixXd::Constant(3,3,1.2)) * 50;
  cout << "m =" << endl << m << endl;
  VectorXd v(3);
  v << 1, 2, 3;
  cout << "m * v =" << endl << m * v << endl;
}
```

```
512
261
```

The expression `m + MatrixXf::Constant(3,3,1.2)` constructs the 3-by-3 matrix expression with all its coefficients equal to 1.2 plus the corresponding coefficient of *m*.

The comma-initializer, too, can also be used to construct temporary objects. The following example constructs a random matrix of size 2-by-3, and then multiplies this matrix on the left with $\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$.

Example:

```
MatrixXf mat = MatrixXf::Random(2, 3);
std::cout << mat << std::endl << std::endl;
mat = (MatrixXf(2,2) << 0, 1, 1, 0).finished() * mat;
std::cout << mat << std::endl;
```

Output:

```
  0.68   0.566  0.823
-0.211   0.597 -0.605

-0.211   0.597 -0.605
  0.68   0.566  0.823
```

The **finished()** method is necessary here to get the actual matrix object once the comma initialization of our temporary submatrix is done.