Block operations

Dense matrix and array manipulation

This page explains the essentials of block operations. A block is a rectangular part of a matrix or array. Blocks expressions can be used both as rvalues and as lvalues. As usual with **Eigen** expressions, this abstraction has zero runtime cost provided that you let your compiler optimize.

Using block operations

The most general block operation in **Eigen** is called **.block()** . There are two versions, whose syntax is as follows:

Block operation	Version constructing a dynamic-size block expression	Version constructing a fixed-size block expression
Block of size `(p,q)`, starting at `(i,j)`	`matrix.block(i,j,p,q);`	`matrix.block<p,q>(i,j);`

As always in **Eigen**, indices start at 0.

Both versions can be used on fixed-size and dynamic-size matrices and arrays. These two expressions are semantically equivalent. The only difference is that the fixed-size version will typically give you faster code if the block size is small, but requires this size to be known at compile time.

The following program uses the dynamic-size and fixed-size versions to print the values of several blocks inside a matrix.

Example:

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

using namespace std;

int main()
{
  Eigen::MatrixXf m(4,4);
  m <<  1, 2, 3, 4,
        5, 6, 7, 8,
        9,10,11,12,
       13,14,15,16;
  cout << "Block in the middle" << endl;
  cout << m.block<2,2>(1,1) << endl << endl;
  for (int i = 1; i <= 3; ++i)
  {
    cout << "Block of size " << i << "x" << i << endl;
    cout << m.block(0,0,i,i) << endl << endl;
  }
}
```

Output:

```
Block in the middle
 6  7
10 11

Block of size 1x1
1

Block of size 2x2
1 2
5 6

Block of size 3x3
 1  2  3
 5  6  7
 9 10 11
```

In the above example the **.block()** function was employed as a *rvalue*, i.e. it was only read from. However, blocks can also be used as *lvalues*, meaning that you can assign to a block.

This is illustrated in the following example. This example also demonstrates blocks in arrays, which works exactly like the above-demonstrated blocks in matrices.

Example:

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

using namespace std;
using namespace Eigen;

int main()
{
  Array22f m;
```

Output:

```
Here is the array a:
0.6 0.6 0.6 0.6
0.6 0.6 0.6 0.6
0.6 0.6 0.6 0.6
0.6 0.6 0.6 0.6
```

```
  m << 1,2,
       3,4;
  Array44f a = Array44f::Constant(0.6);
  cout << "Here is the array a:"
        << endl << a << endl << endl;
  a.block<2,2>(1,1) = m;
  cout << "Here is now a with m
        copied into its central
        2x2 block:" << endl << a
        << endl << endl;
  a.block(0,0,2,3) = a.block(2,1,2,3);
  cout << "Here is now a with
        bottom-right 2x3 block
        copied into top-left 2x2
        block:" << endl << a <<
        endl << endl;
}
```

```
Here is now a with m copied into its central 2x2 block:
0.6 0.6 0.6 0.6
0.6   1   2 0.6
0.6   3   4 0.6
0.6 0.6 0.6 0.6

Here is now a with bottom-right 2x3 block copied into top-left 2x2 block:
  3   4 0.6 0.6
0.6 0.6 0.6 0.6
0.6   3   4 0.6
0.6 0.6 0.6 0.6
```

While the **.block()** method can be used for any block operation, there are other methods for special cases, providing more specialized API and/or better performance. On the topic of performance, all what matters is that you give **Eigen** as much information as possible at compile time. For example, if your block is a single whole column in a matrix, using the specialized **.col()** function described below lets **Eigen** know that, which can give it optimization opportunities.

The rest of this page describes these specialized methods.

Columns and rows

Individual columns and rows are special cases of blocks. **Eigen** provides methods to easily address them: **.col()** and **.row()**.

Block operation	Method
i^{th} row *	`matrix.row(i);`
j^{th} column *	`matrix.col(j);`

The argument for `col()` and `row()` is the index of the column or row to be accessed. As always in **Eigen**, indices start at 0.

Example:

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

using namespace std;

int main()
{
  Eigen::MatrixXf m(3,3);
  m << 1,2,3,
       4,5,6,
       7,8,9;
  cout << "Here is the
        matrix m:" << endl
        << m << endl;
  cout << "2nd Row: " <<
        m.row(1) << endl;
  m.col(2) += 3 * m.col(0);
  cout << "After adding 3
        times the first
        column into the
        third column, the
        matrix m is:\n";
  cout << m << endl;
}
```

Output:

```
Here is the matrix m:
1 2 3
4 5 6
7 8 9
2nd Row: 4 5 6
After adding 3 times the first column into the third column, the matrix m is:
 1  2  6
 4  5 18
 7  8 30
```

That example also demonstrates that block expressions (here columns) can be used in arithmetic like any other expression.

Corner-related operations

Eigen also provides special methods for blocks that are flushed against one of the corners or sides of a matrix or array. For instance, **.topLeftCorner()** can be used to refer to a block in the top-left corner of a matrix.

The different possibilities are summarized in the following table:

Block operation	Version constructing a dynamic-size block expression	Version constructing a fixed-size block expression
Top-left p by q block *	`matrix.topLeftCorner(p,q);`	`matrix.topLeftCorner<p,q>();`
Bottom-left p by q block *	`matrix.bottomLeftCorner(p,q);`	`matrix.bottomLeftCorner<p,q>();`
Top-right p by q block *	`matrix.topRightCorner(p,q);`	`matrix.topRightCorner<p,q>();`
Bottom-right p by q block *	`matrix.bottomRightCorner(p,q);`	`matrix.bottomRightCorner<p,q>();`
Block containing the first q rows *	`matrix.topRows(q);`	`matrix.topRows<q>();`
Block containing the last q rows *	`matrix.bottomRows(q);`	`matrix.bottomRows<q>();`
Block containing the first p columns *	`matrix.leftCols(p);`	`matrix.leftCols<p>();`
Block containing the last q columns *	`matrix.rightCols(q);`	`matrix.rightCols<q>();`

Here is a simple example illustrating the use of the operations presented above:

Example:

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

using namespace std;

int main()
{
  Eigen::Matrix4f m;
  m << 1, 2, 3, 4,
       5, 6, 7, 8,
       9, 10,11,12,
       13,14,15,16;
  cout << "m.leftCols(2) =" << endl << m.leftCols(2) << endl << endl;
  cout << "m.bottomRows<2>() =" << endl << m.bottomRows<2>() << endl << endl;
  m.topLeftCorner(1,3) = m.bottomRightCorner(3,1).transpose();
  cout << "After assignment, m = " << endl << m << endl;
}
```

Output:

```
m.leftCols(2) =
 1  2
 5  6
 9 10
13 14

m.bottomRows<2>() =
 9 10 11 12
13 14 15 16

After assignment, m =
 8 12 16  4
 5  6  7  8
 9 10 11 12
13 14 15 16
```

Block operations for vectors

Eigen also provides a set of block operations designed specifically for the special case of vectors and one-dimensional arrays:

Block operation	Version constructing a dynamic-size block expression	Version constructing a fixed-size block expression
Block containing the first `n` elements *	`vector.head(n);`	`vector.head<n>();`
Block containing the last `n` elements *	`vector.tail(n);`	`vector.tail<n>();`
Block containing `n` elements, starting at position `i` *	`vector.segment(i,n);`	`vector.segment<n>(i);`

An example is presented below:

Example: **Output:**

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

using namespace std;

int main()
{
  Eigen::ArrayXf v(6);
  v << 1, 2, 3, 4, 5, 6;
  cout << "v.head(3) =" << endl << v.head(3) << endl << endl;
  cout << "v.tail<3>() = " << endl << v.tail<3>() << endl << endl;
  v.segment(1,4) *= 2;
  cout << "after 'v.segment(1,4) *= 2', v =" << endl << v << endl;
}
```

```
v.head(3) =
1
2
3

v.tail<3>() =
4
5
6

after 'v.segment(1,4) *= 2', v =
 1
 4
 6
 8
10
 6
```