使用自定义标量类型

默认情况下，Eigen目前支持标准浮点类型（`float`，`double`，`std::complex<float>`，`std::complex<double>`，`long double`），以及所有天然整数类型（例如，`int`，`unsigned int`，`short`，等等），和`bool`。在 x86-64 系统上，`long double`允许本地强制使用具有扩展精度的 x87 寄存器（与 SSE 相比）。

为了添加对自定义类型`T`的支持，您需要：

1. 确保类型支持通用运算符（+、-、*、/等）`T`
2. 添加 struct Eigen::NumTraits<T> 的特化（参见NumTraits）
3. 定义对您的类型有意义的数学函数。这包括标准的，如 sqrt、pow、sin、tan、conj、real、imag等，以及Eigen特定的abs2 。（参见文件Eigen/src/Core/MathFunctions.h）

数学函数应该在与相同的命名空间中定义`T`，或者在`std`命名空间中定义，尽管不推荐第二种方法。

这是一个添加对 Adolc `adouble`类型支持的具体示例。Adolc是一个自动微分库。该类型`adouble`基本上是一个真实值，跟踪任意数量的偏导数的值。

```
#ifndef ADOLCSUPPORT_H
#define ADOLCSUPPORT_H

#define ADOLC_TAPELESS
#include <adolc/adouble.h>
#include <Eigen/Core>

namespace Eigen {
template<> struct NumTraits<adtl::adouble>
 : NumTraits<double> // permits to get the epsilon, dummy_precision, lowest, highest functions
{
    typedef adtl::adouble Real;
    typedef adtl::adouble NonInteger;
    typedef adtl::adouble Nested;

    enum {
        IsComplex = 0,
        IsInteger = 0,
        IsSigned = 1,
        RequireInitialization = 1,
        ReadCost = 1,
        AddCost = 3,
        MulCost = 3
    };
};
}

namespace adtl {
inline const adouble& conj(const adouble& x)  { return x; }
inline const adouble& real(const adouble& x)  { return x; }
inline adouble imag(const adouble&)    { return 0.; }
inline adouble abs(const adouble&  x)  { return fabs(x); }
inline adouble abs2(const adouble& x)  { return x*x; }
}
```

```
35
36 #endif // ADOLCSUPPORT_H
```

这个其他示例添加了对GMP`mpq_class`类型的支持。它特别展示了如何在 LU 分解过程中改变Eigen选择最佳支点的方式。它选择得分最高的系数，其中得分默认为数字的绝对值，但我们可以定义不同的得分，例如更喜欢具有更紧凑表示的枢轴（这是一个示例，而不是推荐）。请注意，分数应始终为非负数，并且只允许零分数为零。此外，这可能会与不精确标量类型的阈值发生严重交互。

```
1  #include <gmpxx.h>
2  #include <Eigen/Core>
3  #include <boost/operators.hpp>
4
5  namespace Eigen {
6      template<> struct NumTraits<mpq_class> : GenericNumTraits<mpq_class>
7      {
8          typedef mpq_class Real;
9          typedef mpq_class NonInteger;
10         typedef mpq_class Nested;
11
12         static inline Real epsilon() { return 0; }
13         static inline Real dummy_precision() { return 0; }
14         static inline int digits10() { return 0; }
15
16         enum {
17             IsInteger = 0,
18             IsSigned = 1,
19             IsComplex = 0,
20             RequireInitialization = 1,
21             ReadCost = 6,
22             AddCost = 150,
23             MulCost = 100
24         };
25     };
26
27     namespace internal {
28     template<> struct scalar_score_coeff_op<mpq_class> {
29         struct result_type : boost::totally_ordered1<result_type> {
30             std::size_t len;
31             result_type(int i = 0) : len(i) {} // Eigen uses Score(0) and
   Score()
32             result_type(mpq_class const& q) :
   len(mpz_size(q.get_num_mpz_t()) +
   mpz_size(q.get_den_mpz_t())-1) {}
33             friend bool operator<(result_type x, result_type y) {
34                 // 0 is the worst possible pivot
35                 if (x.len == 0) return y.len > 0;
36                 if (y.len == 0) return false;
37                 // Prefer a pivot with a small representation
38                 return x.len > y.len;
39             }
40             friend bool operator==(result_type x, result_type y) {
41                 // Only used to test if the score is 0
42                 return x.len == y.len;
43             }
44         };
45         result_type operator()(mpq_class const& x) const { return x; }
46     };
```

```
47      }
48  }
```