编写以 Eigen 类型为参数的函数

Eigen 使用表达式模板可能导致每个表达式都属于不同类型。如果您将这样的表达式传递给采用Matrix类型参数的函数，您的表达式将被隐式计算为临时Matrix，然后将其传递给函数。这意味着您将失去表达式模板的好处。具体来说，这有两个缺点：

- 评估成临时可能是无用的和低效的；
- 这仅允许函数从表达式中读取，而不能写入。

幸运的是，所有这些无数的表达式类型都有一个共同点，它们都继承了一些常见的模板化基类。通过让您的函数采用这些基本类型的模板化参数，您可以让它们与 Eigen 的表达式模板完美配合。

一些最初的例子

本节将为 Eigen 提供的不同类型的对象提供简单的示例。在开始实际示例之前，我们需要概括一下我们可以使用哪些基本对象（另请参见类层次结构）。

- MatrixBase：所有密集矩阵表达式的公共基类（与数组表达式相对，与稀疏和特殊矩阵类相对）。在仅适用于密集矩阵的函数中使用它。
- ArrayBase：所有密集数组表达式的公共基类（与矩阵表达式等相反）。在仅适用于数组的函数中使用它。
- DenseBase：所有稠密矩阵表达的共同的基类，也就是，对于基类`MatrixBase`和`ArrayBase`。它可以用在旨在处理矩阵和数组的函数中。
- EigenBase：统一所有类型的对象的基类，这些对象可以被评估为密集矩阵或数组，例如特殊矩阵类，如对角矩阵、置换矩阵等。它可以用于旨在处理任何此类通用的函数类型。

EigenBase 示例

打印 Eigen 中存在的最通用对象的尺寸。它可以是任何矩阵表达式、任何稠密或稀疏矩阵以及任何数组。

例子：

```
#include <iostream>
#include <Eigen/Core>
using namespace Eigen;

template <typename Derived>
void print_size(const EigenBase<Derived>& b)
{
  std::cout << "size (rows, cols): " << b.size() << " (" << b.rows()
            << ", " << b.cols() << ")" << std::endl;
}

int main()
{
    Vector3f v;
    print_size(v);
    // v.asDiagonal() returns a 3x3 diagonal matrix pseudo-expression
    print_size(v.asDiagonal());
}
```

输出：

```
size (rows, cols): 3 (3, 1)
size (rows, cols): 9 (3, 3)
```

DenseBase 示例

打印密集表达式的子块。接受任何密集矩阵或数组表达式，但不接受稀疏对象和特殊矩阵类，例如 DiagonalMatrix。

```
template <typename Derived>
void print_block(const DenseBase<Derived>& b, int x, int y, int r, int c)
{
    std::cout << "block: " << b.block(x, y, r, c) << std::endl;
}
```

ArrayBase 示例

打印数组或数组表达式的最大系数。

```
template <typename Derived>
void print_max_coeff(const ArrayBase<Derived> &a)
{
    std::cout << "max: " << a.maxCoeff() << std::endl;
}
```

MatrixBase 示例

打印给定矩阵或矩阵表达式的逆条件数。

```
template <typename Derived>
void print_inv_cond(const MatrixBase<Derived>& a)
{
    const typename JacobiSVD<typename
Derived::PlainObject>::SingularValuesType&
        sing_vals = a.jacobiSvd().singularValues();
    std::cout << "inv cond: " << sing_vals(sing_vals.size()-1) / sing_vals(0)
<< std::endl;
}
```

多个模板化参数示例

计算两点之间的欧几里得距离。

```
template <typename DerivedA,typename DerivedB>
typename DerivedA::Scalar squaredist(const MatrixBase<DerivedA>& p1,const
MatrixBase<DerivedB>& p2)
{
    return (p1 - p2).squaredNorm();
}
```

请注意，我们使用了两个模板参数，每个参数一个。这允许函数处理不同类型的输入，例如，

```
squaredist(v1,2*v2)
```

其中第一个参数 `v1` 是一个向量，第二个参数 `2*v2` 是一个表达式。

这些示例只是为了让读者对如何编写带有简单常量Matrix或Array参数的函数有一个第一印象。它们还旨在让读者了解最常见的基类是函数的最佳候选者。在下一节中，我们将更详细地查看示例及其实现的不同方式，同时讨论每个实现的问题和优势。对于下面的讨论，Matrix和Array以及MatrixBase和ArrayBase可以交换并且所有参数仍然成立。

如何编写通用但非模板化的函数?

在前面的所有示例中，函数必须是模板函数。这种方法允许编写非常通用的代码，但通常需要编写非模板化函数并仍然保持一定程度的通用性以避免参数的愚蠢副本。典型的例子是编写接受 MatrixXf 或 MatrixXf 块的函数。这正是Ref类的目的。这是一个简单的例子：

例子：

```
#include <iostream>
#include <Eigen/SVD>
using namespace Eigen;
using namespace std;

float inv_cond(const Ref<const MatrixXf>& a)
{
    const VectorXf sing_vals = a.jacobiSvd().singularValues();
    return sing_vals(sing_vals.size()-1) / sing_vals(0);
}

int main()
{
    Matrix4f m = Matrix4f::Random();
    cout << "matrix m:" << endl << m << endl << endl;
    cout << "inv_cond(m):          " << inv_cond(m)                    <<
endl;
    cout << "inv_cond(m(1:3,1:3)): " << inv_cond(m.topLeftCorner(3, 3))   <<
endl;
    cout << "inv_cond(m+I):        " << inv_cond(m + Matrix4f::Identity())
<< endl;
}
```

输出：

```
matrix m:
   0.68   0.823  -0.444   -0.27
 -0.211  -0.605   0.108  0.0268
  0.566   -0.33 -0.0452   0.904
  0.597   0.536   0.258   0.832

inv_cond(m):          0.0562343
inv_cond(m(1:3,1:3)): 0.0836819
inv_cond(m+I):        0.160204
```

在对 inv_cond 的前两次调用中，没有发生复制，因为参数的内存布局与 `Ref<MatrixXf>` 接受的内存布局相匹配。然而，在最后一次调用中，我们有一个通用表达式，它将被 `Ref<>` 对象自动计算为一个临时的 `MatrixXf`。

参考对象也可以是可写的。以下是计算两个输入矩阵的协方差矩阵的函数示例，其中每一行是一个观察值：

```
void cov(const Ref<const MatrixXf> x, const Ref<const MatrixXf> y,
Ref<MatrixXf> C)
{
    const float num_observations = static_cast<float>(x.rows());
    const RowVectorXf x_mean = x.colwise().sum() / num_observations;
    const RowVectorXf y_mean = y.colwise().sum() / num_observations;
    C = (x.rowwise() - x_mean).transpose() * (y.rowwise() - y_mean) /
num_observations;
}
```

这里有两个例子调用 cov 没有任何副本：

```
MatrixXf m1, m2, m3
cov(m1, m2, m3);
cov(m1.leftCols<3>(), m2.leftCols<3>(), m3.topLeftCorner<3,3>());
```

Ref<> 类还有两个可选的模板参数，允许控制无需任何副本即可接受的内存布局类型。有关详细信息，请参阅类Ref文档。

采用普通矩阵或数组参数的函数在哪些情况下工作?

不使用模板函数和Ref类，前一个 cov 函数的简单实现可能如下所示

```
MatrixXf cov(const MatrixXf& x, const MatrixXf& y)
{
    const float num_observations = static_cast<float>(x.rows());
    const RowVectorXf x_mean = x.colwise().sum() / num_observations;
    const RowVectorXf y_mean = y.colwise().sum() / num_observations;
    return (x.rowwise() - x_mean).transpose() * (y.rowwise() - y_mean) /
num_observations;
}
```

与人们一开始可能会想到的相反，除非您需要一个也适用于双矩阵的通用实现，并且除非您不关心临时对象，否则此实现很好。为什么会这样？临时工在哪里？下面给出的代码如何编译？

```
MatrixXf x,y,z;
MatrixXf C = cov(x,y+z);
```

在这种特殊情况下，该示例很好并且可以工作，因为两个参数都声明为*const*引用。编译器创建一个临时对象并将表达式 x+z 计算为这个临时对象。一旦函数被处理，临时被释放并将结果分配给 C。

注意：使用对Matrix（或Array）的*const*引用的函数可以以临时性为代价处理表达式。

在哪些情况下，采用普通 Matrix 或 Array 参数的函数会失败?

在这里，我们考虑对上面给出的函数稍加修改的版本。这一次，我们不想返回结果，而是传递一个额外的非常量参数，它允许我们存储结果。第一个简单的实现可能如下所示。

```
// 注意：这段代码有缺陷！
void cov(const MatrixXf& x, const MatrixXf& y, MatrixXf& C)
{
    const float num_observations = static_cast<float>(x.rows());
    const RowVectorXf x_mean = x.colwise().sum() / num_observations;
    const RowVectorXf y_mean = y.colwise().sum() / num_observations;
    C = (x.rowwise() - x_mean).transpose() * (y.rowwise() - y_mean) /
num_observations;
}
```

尝试执行以下代码时

```
MatrixXf C = MatrixXf::Zero(3,6);
cov(x,y, C.block(0,0,3,3));
```

编译器将失败，因为不可能将 返回的表达式`MatrixXf::block()`转换为非常量`MatrixXf&`。之所以如此，是因为编译器想要保护您免于将结果写入临时对象。在这种特殊情况下，这种保护不是有意的——我们想写入一个临时对象。那么我们怎样才能克服这个问题呢？

目前首选的解决方案是基于一个小*技巧*。需要传递对矩阵的常量引用，并且在内部需要丢弃常量。C98兼容编译器的正确实现是

```
template <typename Derived, typename OtherDerived>
void cov(const MatrixBase<Derived>& x, const MatrixBase<Derived>& y,
MatrixBase<OtherDerived> const & C)
{
    typedef typename Derived::Scalar Scalar;
    typedef typename internal::plain_row_type<Derived>::type RowVectorType;

    const Scalar num_observations = static_cast<Scalar>(x.rows());

    const RowVectorType x_mean = x.colwise().sum() / num_observations;
    const RowVectorType y_mean = y.colwise().sum() / num_observations;

    const_cast< MatrixBase<OtherDerived>& >(C) =
        (x.rowwise() - x_mean).transpose() * (y.rowwise() - y_mean) /
num_observations;
}
```

上面的实现现在不仅适用于临时表达式，而且还允许将该函数与任意浮点标量类型的矩阵一起使用。

注意： const cast hack 仅适用于模板化函数。它不适用于 MatrixXf 实现，因为不可能将Block表达式转换为Matrix引用！

如何在通用实现中调整矩阵的大小？

有人可能认为我们现在已经完成了，对吧？这并不完全正确，因为为了让我们的协方差函数普遍适用，我们希望以下代码能够工作

```
MatrixXf x = MatrixXf::Random(100,3);
MatrixXf y = MatrixXf::Random(100,3);
MatrixXf C;
cov(x, y, C);
```

当我们使用以MatrixBase作为参数的实现时，情况不再如此。一般来说，Eigen 支持自动调整大小，但不能在表达式上这样做。为什么应该允许调整矩阵块的大小？它是对子矩阵的引用，我们绝对不想调整它的大小。那么如果我们不能在MatrixBase上调整大小，我们如何合并调整大小？解决方案是在此实现中调整派生对象的大小。

```
template <typename Derived, typename OtherDerived>
void cov(const MatrixBase<Derived>& x, const MatrixBase<Derived>& y,
MatrixBase<OtherDerived> const & C_)
{
    typedef typename Derived::Scalar Scalar;
    typedef typename internal::plain_row_type<Derived>::type RowVectorType;

    const Scalar num_observations = static_cast<Scalar>(x.rows());

    const RowVectorType x_mean = x.colwise().sum() / num_observations;
    const RowVectorType y_mean = y.colwise().sum() / num_observations;

    MatrixBase<OtherDerived>& C = const_cast< MatrixBase<OtherDerived>& >
(C_);

    C.derived().resize(x.cols(),x.cols()); // resize the derived object
    C = (x.rowwise() - x_mean).transpose() * (y.rowwise() - y_mean) /
num_observations;
}
```

此实现现在适用于作为表达式的参数和作为矩阵且大小错误的参数。在这种情况下，调整表达式的大小不会造成任何伤害，除非它们确实需要调整大小。这意味着，传递具有错误维度的表达式将导致运行时错误（仅在调试模式下），而传递正确大小的表达式将正常工作。

注意：在上面的讨论中，Matrix和Array以及MatrixBase和ArrayBase可以互换，并且所有参数仍然有效。

总结

- 总而言之，采用不可写（const 引用）对象的函数的实现不是一个大问题，并且不会导致在编译和运行程序方面出现问题。但是，简单的实现可能会在您的代码中引入不必要的临时对象。为了避免将参数评估为临时变量，请将它们作为（常量）引用传递给MatrixBase或ArrayBase（因此模板化您的函数）。
- 采用可写（非常量）参数的函数必须采用 const 引用并在函数体内抛弃常量性。
- 将MatrixBase（或ArrayBase）对象作为参数并可能需要调整它们大小（在它们可调整大小的情况下）的函数，必须在派生类上调用 resize()，如由derived() 返回的。