=1

Combining Transforms in ITK

Release 0.00

Stefan Klein[1] and Marius Staring[1]

May 3, 2006

[1]Image Sciences Institute, University Medical Center Utrecht,
Q0S.459, P.O. Box 85500, 3508 GA Utrecht, The Netherlands
stefan@isi.uu.nl and marius@isi.uu.nl

Abstract

The current ITK release 2.6 does not contain a way to combine transforms in a structural way. This document describes a contribution to the Insight Toolkit ITK www.itk.org, which implements what we call an itk::CombinationTransform. Two ways to combine transforms are implemented, by addition and by composition. Depending on this selection, different implementations of the TransformPoint() are provided. Also the expressions for the Jacobian and the inverse transform depend on the way transforms are combined. By using function pointers expensive if-statements are avoided. This paper is accompanied with the source code.

Contents

1 Introduction

The current ITK release 2.6 does not contain a way to combine transforms in a structural way. Some means to do so are implemented, more or less ad hoc. The itk::BSplineDeformableTransform has for example a bulk transform, which is added to the B-spline transform, and all classes inheriting from the itk::MatrixOffsetTransformBase have a smart Compose() function. We present a more generic way to implement the combination of different transforms by defining a itk::CombinationTransform.

Generally, there are two ways to combine a set of transforms $\{T_1, \ldots, T_N\}$ to get a combined transform $T(\boldsymbol{x})$, where $\boldsymbol{x}$ are the voxels. The first way is by addition:

$$T(\boldsymbol{x}) = T_1(\boldsymbol{x}) + \cdots + T_N(\boldsymbol{x}), \tag{1}$$

as is done for the BulkTransform in the itk::BSplineDeformableTransform. The second way is by composition:

$$T(\boldsymbol{x}) = (T_N \circ \cdots \circ T_1)(\boldsymbol{x}), \tag{2}$$

as in the itk::MatrixOffsetTransformBase.

For image registration often the Jacobian of the transform has to be calculated. We assume that only one of the transforms in the set $\{T_1, \ldots, T_N\}$ has to be optimised over. Say this transform is the last one and has parameters $\boldsymbol{\mu}$: $T_N(\boldsymbol{x}, \boldsymbol{\mu})$. All other transforms $\{T_1, \ldots, T_{N-1}\}$ serve as an initialisation. Now, for the case of addition, the derivative of the combined transform $T(\boldsymbol{x}, \boldsymbol{\mu})$ to its parameters $\boldsymbol{\mu}$ at some point $\boldsymbol{p}$ is

$$\frac{\partial T}{\partial \boldsymbol{\mu}}\Big|_{\boldsymbol{x}=\boldsymbol{p}} = \frac{\partial T_N}{\partial \boldsymbol{\mu}}\Big|_{\boldsymbol{x}=\boldsymbol{p}}, \tag{3}$$

which is exactly the Jacobian of T_N at point $\boldsymbol{p}$. For the case of composition however, we get

$$\frac{\partial T}{\partial \boldsymbol{\mu}}\Big|_{\boldsymbol{x}=\boldsymbol{p}} = \frac{\partial}{\partial \boldsymbol{\mu}} T_N\left((T_{N-1} \circ \cdots \circ T_1)(\boldsymbol{p}), \boldsymbol{\mu}\right) = \frac{\partial T_N}{\partial \boldsymbol{\mu}}\Big|_{\boldsymbol{x}=(T_{N-1} \circ \cdots \circ T_1)(\boldsymbol{p})}, \tag{4}$$

which is the Jacobian of T_N at the transformed point $(T_{N-1} \circ \cdots \circ T_1)(\boldsymbol{p})$.

2 Implementation

We define and implement a itk::CombinationTransform that combines two transforms, which we call the InitialTransform and the CurrentTransform. Combining more transforms can be done recursively, by setting the InitialTransform to a itk::CombinationTransform type again.

2.1 CombinationTransform

The CombinationTransform is defined in the accompanying source code in the files itkCombinationTransform.h and itkCombinationTransform.txx. The CombinationTransform is templated over the CoordinateRepresentationType and the SpaceDimension: it is assumed that the transformations that are combined all map from nD to nD.

The TransformPoint() for addition is defined as:

```
template <typename TScalarType, unsigned int NDimensions>
  typename CombinationTransform<TScalarType, NDimensions>::OutputPointType
  CombinationTransform<TScalarType, NDimensions>::
  TransformPointUseAddition( const InputPointType  & point ) const
{
  /** The Initial transform */
  OutputPointType out0 =
    this->m_InitialTransform->TransformPoint( point );

  /** The Current transform */
  OutputPointType out =
    this->m_CurrentTransform->TransformPoint( point );
```

```
  /** Both added together */
  for ( unsigned int i=0; i < SpaceDimension; i++ )
  {
    out[ i ] += ( out0[ i ] - point[ i ] );
  }

  return out;
} // end TransformPointUseAddition
```

For composition the code looks like:

```
template <typename TScalarType, unsigned int NDimensions>
  typename CombinationTransform<TScalarType, NDimensions>::OutputPointType
  CombinationTransform<TScalarType, NDimensions>::
  TransformPointUseComposition( const InputPointType  & point ) const
{
  return this->m_CurrentTransform->TransformPoint(
    this->m_InitialTransform->TransformPoint( point ) );
} // end TransformPointUseComposition
```

The `GetJacobian()` for addition is simply the `GetJacobian()` of the CurrentTransform. For composition it can be implemented by first calling `TransformPoint()` of the InitialTransform on a point $\boldsymbol{p}$ and then substituting the resulting transformed point into the `GetJacobian()` of the CurrentTransform. This way nothing has to be changed in existing implementations of all `itk::ImageToImageMetric`'s. The code is:

```
template <typename TScalarType, unsigned int NDimensions>
  const typename CombinationTransform<TScalarType, NDimensions>::JacobianType &
  CombinationTransform<TScalarType, NDimensions>::
  GetJacobianUseComposition( const InputPointType  & point ) const
{
  return this->m_CurrentTransform->GetJacobian(
    this->m_InitialTransform->TransformPoint( point ) );
} // end GetJacobianUseComposition
```

So, there are different implementations for different combination strategies. The `TransformPoint()` will have to select one of these options. We have chosen to implement that by using function pointers. When the `itk::CombinationTransform` is setup, a function pointer, called `m_SelectedTransformPointFunction`, is pointed to the correct `TransformPoint()`. The `TransformPoint()` implementation becomes:

```
template <typename TScalarType, unsigned int NDimensions>
  typename CombinationTransform<TScalarType, NDimensions>::OutputPointType
  CombinationTransform<TScalarType, NDimensions>::
  TransformPoint(const InputPointType  & point ) const
{
  /** Call the selected TransformPointFunction */
  return ((*this).*m_SelectedTransformPointFunction)(point);
} // end TransformPoint
```

Something similar is done for the `GetJacobian()` function.

For more details we refer to the well-documented source code.

2.2 BSplineCombinationTransform

Because the itk::BSplineDeformableTransform also has a TransformPoint() with five input arguments, we also have to define a BSplineCombinationTransform. The BSplineCombinationTransform inherits from the CombinationTransform and adds a TransformPoint() with five input arguments:

```
template <typename TScalarType, unsigned int NDimensions, unsigned int VSplineOrder>
  void BSplineCombinationTransform<TScalarType, NDimensions, VSplineOrder>::
  TransformPoint(
    const InputPointType &inputPoint,
    OutputPointType &outputPoint,
    WeightsType &weights,
    ParameterIndexArrayType &indices,
    bool &inside ) const
{
  /** Call the selected TransformPointBSplineFunction */
  ((*this).*m_SelectedTransformPointBSplineFunction)(
    inputPoint, outputPoint, weights, indices, inside);

} // end TransformPoint with extra arguments
```

Again, for more details we refer to the source code.

Note that in the itk::MattesMutualInformationImageToImageMetric a dynamic cast is used to check if the transform inherits from a itk::BSplineDeformableTransform, in order to achieve a speedup. A dynamic cast to a itk::BSplineCombinationTransform should be added if advantage is to be taken from this speedup for the case of the BSplineCombinationTransform.