Improving the Oriented Image class for use in the Registration Framework

Release 1.00

Rupert Brooks and Tal Arbel

October 11, 2007

McGill Centre for Intelligent Machines
McGill University, Montreal, Canada
rupert.brooks@mcgill.ca, arbel@cim.mcgill.ca

Abstract

The original design of the ITK registration framework was based around the `itk::Image` class, which assumed that the pixel axes were aligned with the coordinate system axes. The `itk::OrientedImage` was added later as a subclass, but problems remain with its gradient calculations. Furthermore, general code that uses the `itk::OrientedImage` will suffer an unnecessary penalty when the image is oriented parallel to the image axes. We propose a new `itk::FastOrientedImage` class that alleviates these performance problems, and a change to the design of `itk::ImageToImageMetric` that resolves the gradient issue, and adds a number of additional capacities to the image metrics.

Contents

1 Introduction

The registration framework in ITK was originally designed around the class `itk::Image`, which defines an image as a lattice of pixels oriented parallel to the coordinate system axes. A more general definition of image, however, is to allow the directions of each pixel axis to be arbitrary orthogonal directions in space. This is achieved by storing a transformation matrix, or direction cosines, for each axis. In ITK, this was implemented in the `itk::OrientedImage` class.

However, there are two problems with this approach. One is that, inevitably, computing the coordinate system to pixel and pixel to coordinate system conversions takes more time in the more general, oriented, case. Since this calculation must be performed thousands or millions of times in the registration framework, the `itk::OrientedImage` incurs a significant performance penalty.

Secondly, and more seriously, this leads to an ambiguity in the definition of derivative. The registration framework requires the image derivative relative to the world coordinate system. With the `itk::Image` class the world coordinate system was aligned with the image coordinate system. Thus what was required, and was computed, were image derivatives relative to the pixel axes, scaled by the pixel spacing. In the case of an oriented image, the derivative with respect to the pixel directions must be transformed to get the derivative with respect to the coordinate system axes. However, the ITK design keeps the image gradient separate from the image definition itself. Thus the gradient is computed by a number of image filters, including the `itk::GradientImageFilter`, `itk::GradientRecursiveGaussianImageFilter`, and `itk::CentralDifferencingImageFilter` as well as others. Changing the definition of image gradient to support the registration framework will be time consuming and difficult to correctly implement. It is also possible that existing work would be broken by making major changes to how derivatives are calculated on `itk::OrientedImage`. [1]

2 Background

2.1 Pixel coordinates to world coordinates

An image has a natural pixel coordinate system. This coordinate system is anchored in a world coordinate system by three components. The origin,O, is the location of the pixel $\bar{0}$ in world coordinates. The spacing, S, is a vector of distances between pixel centers. Finally the directions are an orthogonal transformation matrix, D.

The world coordinates, X_w, of a particular pixel, X_p, are given by

$$X_w = O + D \cdot diag(S) \cdot X_p \tag{1}$$

where $diag(S)$ is matrix whose diagonal entries are the elements of S, and whose off-diagonal entries are zero. Similarly, the pixel coordinates of a particular world pixel are given by

$$X_p = diag(S)^{-1} \cdot D^{-1} \cdot (X_w - O) \tag{2}$$

These conversions are handled by four methods of the `itk::Image` class and subclasses, specifically `TransformPhysicalPointToIndex`, `TransformIndexToPhysicalPoint`,

[1] As this document was being written, the ITK working group came to a decision that the image gradient issue should be resolved by adding a method to the `itk::Image` base class and correcting all the filters which calculate derivatives [1]. We think it would still be of interest to generalize the method for calculating derivatives in the image metrics, as this would add considerable versatility to the system.

TransformPhysicalPointToContinuousIndex, and, TransformContinuousIndexToPhysicalPoint. We will refer to these four methods in this document as the Pixel-World Relationship methods (PWR methods).

2.2 Image and spatial gradients

Since these coordinate conversions are one-to-one and differentiable, the derivative of any function defined in the space can be taken with respect to either set of coordinates. In particular, considering an image as a function $f(X)$, the derivative of an image in world space can be related to the derivative of an image in pixel space by:

$$\frac{\partial f(X)}{\partial X_w} = \frac{\partial f(X)}{\partial X_p} \cdot \frac{\partial X_p}{\partial X_w}$$

From Equation 2 we obtain that,

$$\frac{\partial X_p}{\partial X_w} = diag(S)^{-1} \cdot D^{-1}$$

In the case of the current implementation, what is computed is

$$\frac{\partial f(X)}{\partial X_w} = \frac{\partial f(X)}{\partial X_p} \cdot diag(S)^{-1}$$

When the spatial coordinates are aligned with the pixel coordinate axes this is correct. However, in the general case handled by itk::OrientedImage, this is not the derivative that is needed.

3 Proposed Method

3.1 Pixel coordinates to image coordinates

The original itk::Image case is a special case of the itk::OrientedImage calculation. Thus, a itk::OrientedImage can be used in place of an itk::Image as is. Unfortunately, there are significantly more operations to perform the general calculation, than there are to perform the simpler calculation used in itk::Image. Thus general code suffers a significant performance deterioration.

It is clear that the authors of the itk::OrientedImage class have considered the performance issues. The spacing and direction matrices are premultiplied and inverted, and the multiplication loops are unrolled using a template metaprogramming technique. Nevertheless, the PWR methods run much slower in the itk::OrientedImage class. Ideally, when the image axes are aligned with the world coordinate axes, it would be nice to be able to use the more efficient calculation used in itk::Image. In fact, there are a number of special cases, which can be calculated more efficiently (Table 1). Which case applies can be determined once the image coordinate system is known, meaning it must be selected at run time. It does not have to be determined inside the PWR method themselves, however, but can be determined when the coordinate system is set.

The method that we propose to do this is to use function pointers to control which function is used for the evaluation of the PWR methods. When the coordinate system permits, the efficient functions will be used, and when it does not, the generic calculation will be used. When the coordinate system is updated, the PWR function pointers will be changed accordingly. It is worth noting that the calculation TransformPhysicalPointToIndex is identical to the calculation for

Condition	Calculation
$O = 0, S = 1, D = I$	$X_w = X_p$
$O \neq 0, S = 1, D = I$	$X_w = X_p + O$
$O = 0, S \neq 1, D = I$	$X_w = S^T X_p$
$O = 0$, any S, D not diagonal	$X_w = [D \cdot diag(S)] \cdot X_p$
$O \neq 0, S \neq 1, D = I$	$X_w = S^T X_p + O$
$O \neq 0, S = 1, D$ not diagonal	$X_w = [D \cdot diag(S)] \cdot X_p + O$

Table 1: Special cases of the PWR functions that can be calculated efficiently

Condition	Calculation
$O = 0$, any $S, D = I$	$G_w = G_i$
$O = 0$, any S, D not diagonal	$G_w = G_i \cdot D^{-1}$

Table 2: Special cases of the gradient correction functions that can be calculated efficiently. Here G_w is the gradient with respect to world coordinates, and G_i is the gradient as currently calculated.

`TransformPhysicalPointToContinuousIndex` followed by a cast to the index type. Thus there is no point implementing a set of functions for `TransformPhysicalPointToIndex`. The converse is not true however, in the case of `TransformIndexToPhysicalPoint`.

Function pointers can lead to somewhat obscure code, however, this was judged to be the best approach available. Alternatives would be to use a naive approach of checking for the special cases on each evaluation, but this would add almost as many checks as operations saved, and save very little time in the end. A more reasonable alternative method would be to create subclasses of the `itk::OrientedImage` for each special case. However, if these are chosen at compile time, then the resulting code is not general. If these are chosen at runtime, then a separate filter chain would have to be declared for every possible case. A filter template instantiated using the general class as its type would accept one of the specific efficient classes as input but would always produce the generic, inefficient, class as output.

3.2 Image Gradient

The image derivative problem inevitably requires a deeper refactoring of the ITK system. To correct the definition of derivative throughout the code, any derivative or Hessian based filter would have to be modified to reflect the new interpretation of derivative. This could affect a lot of legacy code, to fix a problem which seems to be specific to one element of ITK. As it is the image registration framework which requires this change, we argue the change should be implemented in that framework.

In principle, the change is simple. A method can be added to the `itk::OrientedImage` class to correct the image gradient from pixel space to world space. Like the PWR methods, this method can be implemented very efficiently in special cases. By locating this method in the `itk::OrientedImage`, efficient versions of this function can implemented, and the choice controlled using function pointers (Table 2).

However, this may seem to just push the burden of managing the derivative problem to the `itk::ImageToImageMetric` framework. Each metric would have to be modified to perform this adjustment when computing the derivative of the moving image. However, there are other issues relating to derivatives in the image registration framework which could make this worthwhile.

A number of different methods can be used for computing the image derivative in these metrics, each of

which has different advantages and disadvantages. We propose to add a versatile method for handling derivatives to the ImageToImageMetric base class, which can then be inherited by all subclasses. In this paper, we have implemented a subclass of ImageToImageMetric which contains this general approach, and implemented three of the most commonly used image metrics in this framework.

There are at least three methods in current use in the registration framework for computing derivatives. The first, default method, used by at least the `itk::MeanSquaresImageToImageMetric` and the `itk::NormalizedCorrelationImageToImageMetric` is to compute and store the gradient image of the moving image, and then to compute the required gradient by nearest neighbor interpolation during the registration process. In contrast, the `itk::MattesMutualInformationImageToImageMetric` uses two different methods of computing the gradient. If a B–Spline image interpolator is being used, then it is exploited to compute the gradient. In the general case, a finite differencing approach is used to compute the gradient.

Each method has its advantages and disadvantages. The first method is fastest when the gradient is required on all pixels. However, its memory requirements can be burdensome. The second and third methods trade speed for memory efficiency, and arguably, slightly more accuracy. When not all the pixels are being utilized - something proposed as a general approach in [7] and [2], this can be more efficient.

Furthermore, there are alternative image gradient computation methods in the literature, for example the inverse compositional method [3, 4], which it may be of interest to incorporate in ITK. If the this calculation is in the base class, adding new methods of computing the image gradient to all metrics becomes greatly simplified.

4 Implementation

This submission contains 5 classes:

- `itk::FastOrientedImage`
- `itk::ModImageToImageMetric`
- `itk::MeanSquaresModImageToImageMetric`
- `itk::NormalizedCorrelationModImageToImageMetric`
- `itk::MattesMutualInformationModImageToImageMetric`

The implementation of `itk::FastOrientedImage` is straightforward. A function pointer variable is added to the class for each of the four PWR functions, and the gradient correction function. The callable part of these methods is implemented as a simple wrapper function which calls the function pointed to by the corresponding variable, and contains any code common to all methods. Upon instantiation, the constructor sets these to point to the most generic function, and the `SetOrigin()`, `SetDirection)(` and `SetSpacing()` methods are modified to call a method, `SetIndexFunctions()`, that chooses and sets the appropriate function pointers.

`itk::OrientedImage` used the `itk::ImageTransformHelper` to inline and unroll loops. As our approach has created several specialized loops numerous helper classes would be needed. Loops have been manually unrolled for the two and three dimensional case, which seems to help the efficiency significantly. This unrolling comes at the expense of some code clarity. However, as these functions are used extremely heavily by the registration framework, this is considered worthwhile.

The itk::ModImageToImageMetric class adds a ComputeImageGradient() method, and a m_GradientMethod variable with corresponding methods for setting it. The ComputeImageGradient() method is based on the one in itk::MattesMutualInformationMetric. When an image gradient is needed in the metric calculation, this method should be used to get it. This approach is implemented for three of the most commonly used image metrics.

5 Testing

Testing can be performed by comparing against the results from the original implementations of the classes in question. For the itk::FastOrientedImage class, this is implemented in itkFastOrientedImageTest which tests both performance and accuracy for image dimensions 2,3 and 4. Results are identical, up to numerical accuracy. Timing results for optimized code on two compilers are shown in Table 3. For the continuous index methods, the calculation in the new class is more efficient than itk::OrientedImage, even when the full calculation must be done. In many cases this implementation cannot achieve the efficiency of the itk::Image, even when doing the same calculation. This is due to the inlining of these function calls in itk::Image, something that the function pointer approach does not allow. However, the TransformPhysicalPointToContinuousIndex() method is faster than either existing image class. Overall, these methods minimize the performance penalty incurred by writing general code using itk::OrientedImage.

The TransformPhysicalPointToIndex methods are startlingly slow in all implementations. This is due to the notoriously slow conversion between floating point and integer on Intel processors [5]. It would probably be necessary to write an assembly optimized version to overcome this problem, which would be inherently platform and compiler dependent. As this particular function is not critical to registration speed, we have ignored its slowness.

Testing of the image metric classes is performed by comparing to the original implementation for the particular derivative case that is used in the original implementation. Results should be numerically identical in those cases. This test is performed by the ModImageToImageMetricTest test program. Performance is difficult to evaluate but can be roughly examined by performing registrations in both cases and comparing the time required. When the image is rotated 90 degrees the registration takes the same steps as when rotated zero degrees, thus these runs may be compared. Selected results are shown in Table 4. It is difficult to draw conclusions from these limited examples, but it can be concluded that the use of this derivative correction imposes at worst a small performance penalty when $D = I$. There is however a time increase when the gradient correction must be performed. This time difference is more noticeable using Visual C++, and is likely due to differences in compiler optimizations. This underscores the importance of switching to an efficient evaluation when the data permits rather than using the general method at all times.

The derivative calculation must be tested by registering images oriented at various angles. The test program tests the registration process with the images rotated by 0, 90, ± 30, and ± 120 degrees. In the original implementation the rotated case usually fails. However, with the new derivative calculation, it succeeds in all orientations.

6 Conclusion and future work

In this paper, we have addressed two issues with the use of the itk::OrientedImage in the registration framework. We suggest creating a general gradient calculation method in the itk::ImageToImageMetric

Case	Method	Time VC++ 2003			Time GCC 4.0.3		
		FOI	OI	I	FOI	OI	I
$O=0,S=1,D=I$	TCITPP	0.593	4.438	0.516	0.46	3.79	0.63
$O=0,S=1,D=I$	TITPP	0.625	1.64	0.781	0.48	1.12	0.79
$O=0,S=1,D=I$	TPPTI	4.079	5	5.062	1.73	1.95	2.77
$O=0,S=1,D=I$	TPPTCI	0.688	6.062	1.797	0.9	4.59	1.95
$O=0,S\neq 1,D=I$	TCITPP	0.641	4.734	0.656	0.62	3.8	0.64
$O=0,S\neq 1,D=I$	TITPP	0.641	1.265	0.547	0.66	1.14	0.79
$O=0,S\neq 1,D=I$	TPPTI	4.5	5	5.063	1.79	1.95	2.78
$O=0,S\neq 1,D=I$	TPPTCI	0.719	6.031	1.781	0.99	4.6	1.95
$O\neq 0,S\neq 1,D=I$	TCITPP	0.656	4.86	0.531	0.74	3.8	0.64
$O\neq 0,S\neq 1,D=I$	TITPP	0.687	1.266	0.547	0.8	1.12	0.81
$O\neq 0,S\neq 1,D=I$	TPPTI	5.484	6.063	5.422	1.86	1.94	2.97
$O\neq 0,S\neq 1,D=I$	TPPTCI	0.812	6.391	1.906	1.16	4.56	2.13
$O\neq 0,S=1,D=I$	TCITPP	0.656	4.422	0.531	0.62	3.8	0.63
$O\neq 0,S=1,D=I$	TITPP	0.641	1.266	0.547	0.64	1.13	0.78
$O\neq 0,S=1,D=I$	TPPTI	5.921	5.563	5.891	1.81	1.94	2.99
$O\neq 0,S=1,D=I$	TPPTCI	0.703	6.062	1.906	0.99	4.58	2.14
$O\neq 0,S=1,D\neq I$	TCITPP	1.563	4.859	0.532*	1.8	3.8	0.64*
$O\neq 0,S=1,D\neq I$	TITPP	1.625	1.265	0.547*	1.68	1.13	0.8*
$O\neq 0,S=1,D\neq I$	TPPTI	10.078	5.5	5.438*	2.43	1.95	2.97*
$O\neq 0,S=1,D\neq I$	TPPTCI	3.203	6.062	1.891*	1.89	4.58	2.14*
$O=0,S=1,D\neq I$	TCITPP	1.344	4.859	0.531*	1.21	3.82	0.63*
$O=0,S=1,D\neq I$	TITPP	1.375	1.266	0.547*	1.19	1.13	0.78*
$O=0,S=1,D\neq I$	TPPTI	8.906	6.063	4.578*	2.44	1.94	2.78*
$O=0,S=1,D\neq I$	TPPTCI	1.234	6.406	1.797*	1.82	4.58	1.95*
$O=0,S\neq 1,D\neq I$	TCITPP	1.328	4.438	0.531*	1.22	3.8	0.64*
$O=0,S\neq 1,D\neq I$	TITPP	1.359	1.438	0.922*	1.18	1.13	0.79*
$O=0,S\neq 1,D\neq I$	TPPTI	8.969	6.062	4.594*	2.43	1.94	2.78*
$O=0,S\neq 1,D\neq I$	TPPTCI	1.234	6.047	1.891*	1.82	4.58	1.95*
$O\neq 0,S\neq 1,D\neq I$	TCITPP	1.562	4.438	0.531*	1.79	3.81	0.63*
$O\neq 0,S\neq 1,D\neq I$	TITPP	1.625	1.266	0.546*	1.69	1.12	0.81*
$O\neq 0,S\neq 1,D\neq I$	TPPTI	10.016	5.484	5.594*	2.42	1.95	2.96*
$O\neq 0,S\neq 1,D\neq I$	TPPTCI	2.797	6.047	1.922*	1.9	4.6	2.13*

Table 3: Timing results for itk::FastOrientedImage. Times reported are for 50 million evaluations of each function in the 3 dimensional case. The tests were performed on two platforms: (1) with code compiled on Visual C++ 2003 and run on a 3.2 GHz Pentium 4 under Windows XP Pro SP2q, and (2) compiled with GCC 4.0.3 and run on a 3.2 GHz Pentium 4 under Ubuntu Linux. All code was compiled using the "Release" settings for the ITK distribution. Timing results must be considered approximate as they can vary from run to run. The cases marked with a * give erroneous results using itk::Image.

Case	Metric	Time VC++ 2003		Time GCC 4.0.3	
		ModMetric	Metric	ModMetric	Metric
No Rotation	MSD	.328	.297	0.38	0.34
No Rotation	NCC	.266	.265	0.3	0.28
No Rotation	MI	.782	.781	0.98	0.96
90 degrees	MSD	0.515 (157% of no rotation)	Fail	0.38 (100% of no rotation)	Fail
90 degrees	NCC	0.375 (141% of no rotation)	Fail	0.32 (107% of no rotation)	Fail
90 degrees	MI	0.812 (103% of no rotation)	Fail	1 (102% of no rotation)	Fail

Table 4: Timing results for itk::FastOrientedImage. Compilation and machine details as in Table 3. All Metrics used the FastOrientedImage class, differences are due solely to the gradient calculation. Timing results must be considered approximate as they can vary from run to run.

class, which can be reused in its subclasses. This will make the subclasses more flexible, and reduce the amount of duplicated code. We propose to correct the image gradient calculation problem by putting a gradient correction method in `itk::OrientedImage`, but placing the responsibility for applying it in the image metrics. This is not overly different from the solution recently proposed in the issue tracking system [1].

We have addressed the performance issues with the use of `itk::OrientedImage` by using function pointers to efficiently switch between efficient versions of the pixel to world functions when they are appropriate, and the general versions when they must be used. The original version of `itk::OrientedImage` used a metaprogramming technique to unroll the loops for efficiency. We have manually unrolled the loops for the critical 2 and 3 dimensional cases.

We conclude that with these modifications, it is feasible to create general image registration code using `itk::OrientedImage` without sacrificing much performance on images aligned with the coordinate axes. Indeed, we achieve superior performance to both the `itk::OrientedImage` and the `itk::Image` classes with the `TransformPhysicalPointToContinuousIndex()` method.

7 Acknowledgment

The idea for using function pointers in this context was inspired by the implementation of the `itk::CombinationTransform` [6].

References

[1] Issue 0005081: Derivative of imagetoimagemetrics in orientedimage. Kitware Bug Tracking system http://public.kitware.com/Bug/view.php?id=5081. Accessed October 5, 2007. 1, 6

[2] Stephen Aylward, Julien Jomier, Sebastien Barre, Brad Davis, and Luis Ibanez. Optimizing ITKs registration methods for multi-processor, shared-memory systems. *Insight Journal*, 2007. ISC/NA-MIC Workshop on Open Science at MICCAI 2007 Issue http://hdl.handle.net/1926/566. 3.2

[3] Simon Baker and Iain Matthews. Lucas-Kanade 20 years on: A unified framework. *International Journal of Computer Vision*, 56(3):221–255, 2004. 3.2

[4] Rupert Brooks and Tal Arbel. Generalizing inverse compositional image alignment. In *Proceedings of the 18th International Conference on Pattern Recognition (ICPR2006)*, volume 2, pages 1200–1203, Hong Kong, August 2006. 3.2

[5] Erik de Castro Lopo. Faster floating point to integer conversions. Web article http://www.mega-nerd.com/FPcast/, 2001. Accessed October 5, 2007. 5

[6] Stefan Klein and Marius Staring. itk::combinationtransform. *Insight Journal*, (1), January-June 2006. http://hdl.handle.net/1926/197. 7

[7] Marius Staring. Contributions to the normalized correlation and the mean squares metric. *Insight Journal*, January-June 2006. January-June 2006 Issue http://hdl.handle.net/1926/190. 3.2