N4ITK: Nick's N3 ITK Implementation For MRI Bias Field Correction

Nicholas J. Tustison and James C. Gee

April 30, 2009

Penn Image Computing and Science Laboratory
University of Pennsylvania

Abstract

Several algorithms exist for correcting the nonuniform intensity in magnetic resonance images caused by field inhomogeneities.[1] These algorithms constitute important preprocessing steps for subsequent image analysis tasks. One such algorithm, known as parametric bias field correction (PABIC) [2], has already been implemented in ITK. Another popular algorithm is the non-uniform intensity normalization (N3) approach [1]. A particularly advantageous aspect of this algorithm is that it does not require a prior tissue model for its application. In addition, the source code for N3 is publicly available at the McConnell Brain Imaging Centre (Montreal Nuerological Institute, McGill University) which includes source code and the coordinating set of perl scripts.[2] This submission describes an implementation of the N3 algorithm for the Insight Toolkit which is implemented as a single class, viz. `itk::N3MRIBiasFieldCorrectionImageFilter`. We tried to maintain minimal difference between the publicly available MNI N3 implementation and our ITK implementation. The only intentional variation is the substitution of an earlier contribution, i.e. the class `itk::BSplineScatteredDataPointSetToImageFilter`, for the originally proposed least-squares approach for B-spline fitting used to model the bias field. This avoids the potential for ill-conditioned fitting matrices for higher B-spline mesh resolutions.

1 Introduction

We forego theoretical discussions of MRI bias field correction and defer to those references already given. Instead, we discuss our implementation and how it relates to both Sled's paper [1] and the original N3 public offering. For notational purposes in this article only, we denote the MNI N3 implementation as N3MNI and the ITK implementation we offer as N4ITK.

[1] A listing of several relevant algorithms compiled by Finn A. Nielsen at the Technical University of Denmark is provided at http://neuro.imm.dtu.dk/staff/fnielsen/bib/Nielsen2001BibSegmentation/Nielsen2001BibSegmentation.html.

[2] http://www.bic.mni.mcgill.ca/software/N3/

2 Implementation

As mentioned in the abstract, the N4ITK implementation is given as a single class `itk::N3MRIBiasFieldCorrectionImageFilter`. It is derived from the `itk::ImageToImageFilter` class (as is the related class `itk::MRIBiasFieldCorrectionFilter`) since its operation takes as input the MR image (with an associated mask) corrupted by a bias field and outputs the corrected image. For the user that wants to reconstruct the bias field once the algorithm terminates, we demonstrate how that can be accomplished with the additional class `itk::BSplineControlPointImageFilter` which we included with this submission. Note that it is only needed if the bias field is to be reconstructed after the N3 algorithm terminates.

2.1 Algorithmic Overview

The steps for the N3 algorithm are illustrated in Fig. 4 of [1]. Initially, the intensities of the input image are transformed into the log space and an initial log bias field of all zeros is instantiated. In N3MNI, an option is given whereby the user can provide an initial bias field estimate but, to keep the options to a minimum, we decided to omit that possibility. However, given the open-source nature of the code, the ITK user can modify the code according to preference.

```
63  /**
64   * Calculate the log of the input image.
65   */
66  typename RealImageType::Pointer logInputImage = RealImageType::New();
67
68  typedef LogImageFilter<InputImageType, RealImageType> LogFilterType;
69  typename LogFilterType::Pointer logFilter = LogFilterType::New();
70  logFilter->SetInput( this->GetInput() );
71  logFilter->Update();
72  logInputImage = logFilter->GetOutput();
```

```
97  /**
98   * Provide an initial log bias field of zeros
99   */
100 typename RealImageType::Pointer logBiasField = RealImageType::New();
101 logBiasField->SetOrigin( this->GetInput()->GetOrigin() );
102 logBiasField->SetRegions( this->GetInput()->GetRequestedRegion() );
103 logBiasField->SetSpacing( this->GetInput()->GetSpacing() );
104 logBiasField->SetDirection( this->GetInput()->GetDirection() );
105 logBiasField->Allocate();
106 logBiasField->FillBuffer( 0.0 );
```

After initialization, we then iterate by alternating between estimating the unbiased log image and estimating the log of the bias field.

```
108 /**
109  * Iterate until convergence or iterative exhaustion.
110  */
111 bool isConverged = false;
112 unsigned int iteration = 0;
113 while( !isConverged && iteration++ < this->m_MaximumNumberOfIterations )
114   {
115   typedef SubtractImageFilter<RealImageType, RealImageType, RealImageType>
116     SubtracterType;
117
```

```
118     typename SubtracterType::Pointer subtracter1 = SubtracterType::New();
119     subtracter1->SetInput1( logInputImage );
120     subtracter1->SetInput2( logBiasField );
121     subtracter1->Update();
122
123     typename RealImageType::Pointer sharpenedImage
124       = this->SharpenImage( subtracter1->GetOutput() );
125
126     typename SubtracterType::Pointer subtracter2 = SubtracterType::New();
127     subtracter2->SetInput1( logInputImage );
128     subtracter2->SetInput2( sharpenedImage );
129     subtracter2->Update();
130
131     typename RealImageType::Pointer newLogBiasField
132       = this->SmoothField( subtracter2->GetOutput() );
133
134     RealType cv = this->CalculateConvergenceMeasurement(
135       logBiasField, newLogBiasField );
136     isConverged = ( cv < this->m_ConvergenceThreshold );
137
138     itkDebugMacro( "Iteration " << iteration << ": "
139       << " convergence criterion = " << cv );
140
141     logBiasField = newLogBiasField;
142     }
```

The two functions contained in the above iterative loop are `SharpenImage()` and `SmoothField()`. The former function essentially implements the discussion in Section II *C. Field Estimation* on page 89 of [1]. We use the vnl fft routines. The latter function gives a smooth estimate of the bias field using the class `itk::BSplineScatteredDataPointSetToImageFilter`. It should be noted the N3MNI gives the user the option of performing the image sharpening in intensity space (not log intensity space). However, since we achieved good results sharpening in log intensity space, to avoid additional calculations, and minimize user options, we avoided implementing this option. However, it can be easily included by the appropriate calls to a `itk::ExpImageFilter` and `itk::LogImageFilter`.

Following convergence or iterative exhaustion, the output image is produced by dividing the intensities of the input image (not in log intensity space) by the smooth bias field estimate.

```
144   typedef ExpImageFilter<RealImageType, RealImageType> ExpImageFilterType;
145   typename ExpImageFilterType::Pointer expFilter = ExpImageFilterType::New();
146   expFilter->SetInput( logBiasField );
147   expFilter->Update();
148
149   /**
150    * Divide the input image by the bias field to get the final image.
151    */
152   typedef DivideImageFilter<InputImageType, RealImageType, OutputImageType>
153     DividerType;
154   typename DividerType::Pointer divider = DividerType::New();
155   divider->SetInput1( this->GetInput() );
156   divider->SetInput2( expFilter->GetOutput() );
157   divider->Update();
158
159   this->SetNthOutput( 0, divider->GetOutput() );
```

2.2 Parameters

One of the attractive aspects of the N3 algorithm is the minimal number of parameters available to tune and the relatively good performance achieved with the default parameters which we tried to maintain, where we

could, for both N3MNI and [1]. The available parameters are:

- `m_MaskLabel` (default = 1): The algorithm requires a mask be supplied by the user with the corresponding mask label. According to Sled, mask generation is not crucial and good results can be achieved with a simple scheme like Otsu thresholding.
- `m_NumberOfHistogramBins` (default = 200): One of the steps of N3 requires histogram construction from the intensities of the uncorrected input image. The default value is the same as in N3MNI.
- `m_WeinerFilterNoise` (default = 0.1): Field estimation is performed by deconvolution using a Wiener filter which has an additive noise term to prevent division by zero (see Equation (12) of [1]). This is identical to the `noise` variable in N3MNI and equal to Z^2 in [1].
- `m_BiasFieldFullWidthAtHalfMaximum` (default = 0.15): A key contribution to N3 is the usage of a simple Gaussian to model the bias field. This variable characterizes that Gaussian and is the same as the `FWHM` variable in both N3MNI and [1].
- `m_MaximumNumberOfIterations` (default = 50): Optimization occurs iteratively until the number of iterations exceeds the maximum specified by this variable.
- `m_ConvergenceThreshold` (default = 0.001): In [1], the authors propose the coefficient of variation between the ratio of subsequent field estimates as the convergence criterion. However, in both N3MNI and N4ITK, the standard deviation of the ratio between subsequent field estimates is used.
- `m_SplineOrder` (default = 3): A smooth field estimate is produced after each iterative correction using B-splines. In both N3MNI and [1], cubic splines are used. Although any feasible order of spline is available, the default in N4ITK is also cubic.
- `m_NumberOfFittingLevels` (default = 4): The B-spline fitting algorithm [4] is different from what is used in N3MNI and proposed in [1]. The version we use was already available in ITK as one of our earlier contributions [3] and is not susceptible to ill-conditioned fitting matrices. One of the parameters for that fitting is the number of hierarchical levels to fit where each successive level doubles the B-spline mesh resolution.
- `m_NumberOfControlPoints` (default = $\underbrace{[4,\ldots]}_{ImageDimension}$): Since the field is usually low frequency, by default we set the number of control points to the minimum `m_SplineOrder`$+1$.

2.3 Bias Field Generation

Oftentimes, the user would like to see the calculated bias field. One of the more obvious reasons for this would be when the bias field is calculated on a downsampled image (suggested in [1] and given as an option in N3MNI and included in the testing code). One would then like to reconstruct the bias field to estimate the corrected image in full resolution. Since the B-spline bias field is a continuous object defined by the control point values and spline order, we can reconstruct the bias field at the image full resolution without loss of accuracy. We demonstrate how this is to be done in the test code `itkN3MRIBiasFieldCorrectionImageFilterTest.cxx`. Note that the control points describe a B-spline scalar field in log space so the `itk::ExpImageFilter` has to be used after reconstruction.

```
105   typedef itk::BSplineControlPointImageFilter<typename
106     CorrecterType::BiasFieldControlPointLatticeType, typename
107     CorrecterType::ScalarImageType> BSplinerType;
108   typename BSplinerType::Pointer bspliner = BSplinerType::New();
109   bspliner->SetInput( correcter->GetBiasFieldControlPointLattice() );
110   bspliner->SetSplineOrder( correcter->GetSplineOrder() );
111   bspliner->SetSize(
112     reader->GetOutput()->GetLargestPossibleRegion().GetSize() );
113   bspliner->SetOrigin( reader->GetOutput()->GetOrigin() );
114   bspliner->SetDirection( reader->GetOutput()->GetDirection() );
115   bspliner->SetSpacing( reader->GetOutput()->GetSpacing() );
116   bspliner->Update();
117
118   typename ImageType::Pointer logField = ImageType::New();
119   logField->SetOrigin( bspliner->GetOutput()->GetOrigin() );
120   logField->SetSpacing( bspliner->GetOutput()->GetSpacing() );
121   logField->SetRegions(
122     bspliner->GetOutput()->GetLargestPossibleRegion().GetSize() );
123   logField->SetDirection( bspliner->GetOutput()->GetDirection() );
124   logField->Allocate();
125
126   itk::ImageRegionIterator<typename CorrecterType::ScalarImageType> ItB(
127     bspliner->GetOutput(),
128     bspliner->GetOutput()->GetLargestPossibleRegion() );
129   itk::ImageRegionIterator<ImageType> ItF( logField,
130     logField->GetLargestPossibleRegion() );
131   for( ItB.GoToBegin(), ItF.GoToBegin(); !ItB.IsAtEnd(); ++ItB, ++ItF )
132     {
133     ItF.Set( ItB.Get()[0] );
134     }
135
136   typedef itk::ExpImageFilter<ImageType, ImageType> ExpFilterType;
137   typename ExpFilterType::Pointer expFilter = ExpFilterType::New();
138   expFilter->SetInput( logField );
139   expFilter->Update();
```

2.4 Test Code

The test code included with this submission, `itkN3MRIBiasFieldCorrectionImageFilterTest.cxx`, is designed to allow the user to immediately apply the N4ITK classes to their own images. Usage is given as follows:

```
itkN3MRIBiasFieldCorrectionImageFilterTest imageDimension inputImage outputImage [shrinkFactor] [maskImage] [
numberOfIterations] [numberOfFittingLevels] [outputBiasField]
```

This class takes the input image, subsamples it according to the optional `shrinkFactor` option, and creates the bias field corrected output image. Other optional parameters are the `maskImage` (if not available, one is created using the `itk::OtsuThresholdImageFilter`), the number of iterations (default = 50), the number of fitting levels (default = 4), and a file name for writing out the resulting bias field.

3 Sample Results

We demonstrate usage with two MR images—a 2-D brain slice and a volume from a hyperpolarized helium-3 image. We use a previous contribution [5] and ITK-SNAP to visualize the results.

3.1 2-D Brain Slice

Figure 1(a) is the uncorrected image used in our 2-D brain test. Close inspection seems to demonstrate a darkening in the white matter towards the upper right of the image. This darkening seems to be corrected in Figure 1(c).

```
>itkN3MRIBiasFieldCorrectionImageFilterTest 2 t81slice.nii.gz t81corrected.nii.gz 2 t81mask.nii.gz 50 4
t81biasfield.nii.gz
```

3.2 3-D Hyperpolarized Helium-3 Lung MRI

Figure 2(a) is the uncorrected image used in our 3-D helium-3 MR image volume. Close inspection seems to demonstrate a darkening in the white matter towards the upper portion of the given axial slice. This darkening seems to be corrected in Figure 2(c).

```
>itkN3MRIBiasFieldCorrectionImageFilterTest 3 he3volume.nii.gz he3corrected.nii.gz 2 he3mask.nii.gz 50 4
he3biasfield.nii.gz
```

References

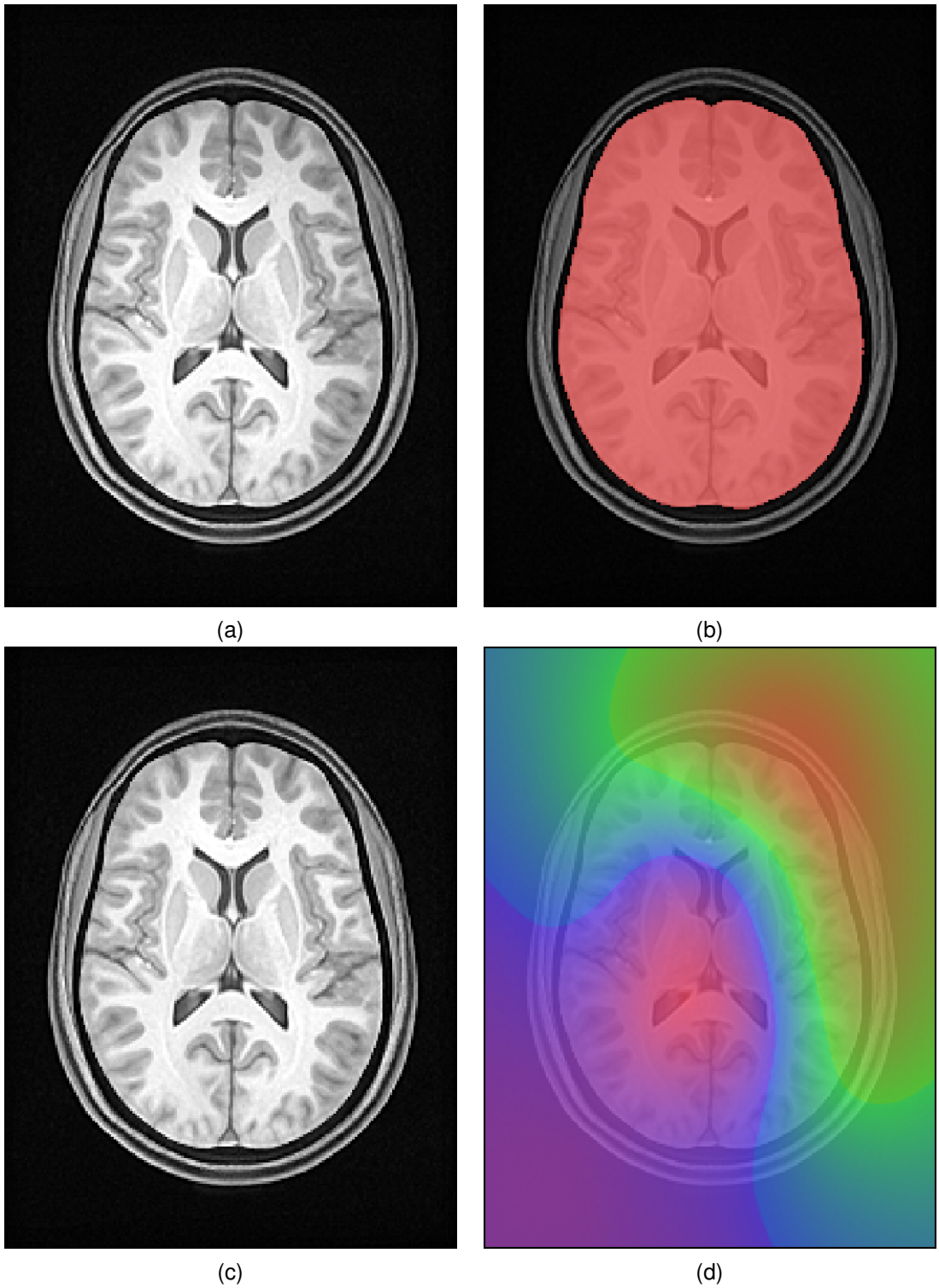
[1] J. G. Sled, A. P. Zijdenbos, and A. C. Evans. A nonparametric method for automatic correction of intensity nonuniformity in mri data. *IEEE Trans Med Imaging*, 17(1):87–97, Feb 1998. (document), 1, 2.1, 2.1, 2.2, 2.3

[2] M. Styner, C. Brechbhler, G. Szkely, and G. Gerig. Parametric estimate of intensity inhomogeneities applied to mri. *IEEE Trans Med Imaging*, 19(3):153–165, Mar 2000. (document)

[3] N. J. Tustison and J. C. Gee. N-d C^k B-spline scattered data approximation. *The Insight Journal*, 2005. 2.2

[4] N. J. Tustison and J. C. Gee. Generalized n-D C^k B-spline scattered data approximation with confidence values. In *Proc. Third International Workshop Medical Imaging and Augmented Reality*, pages 76–83, 2006. 2.2

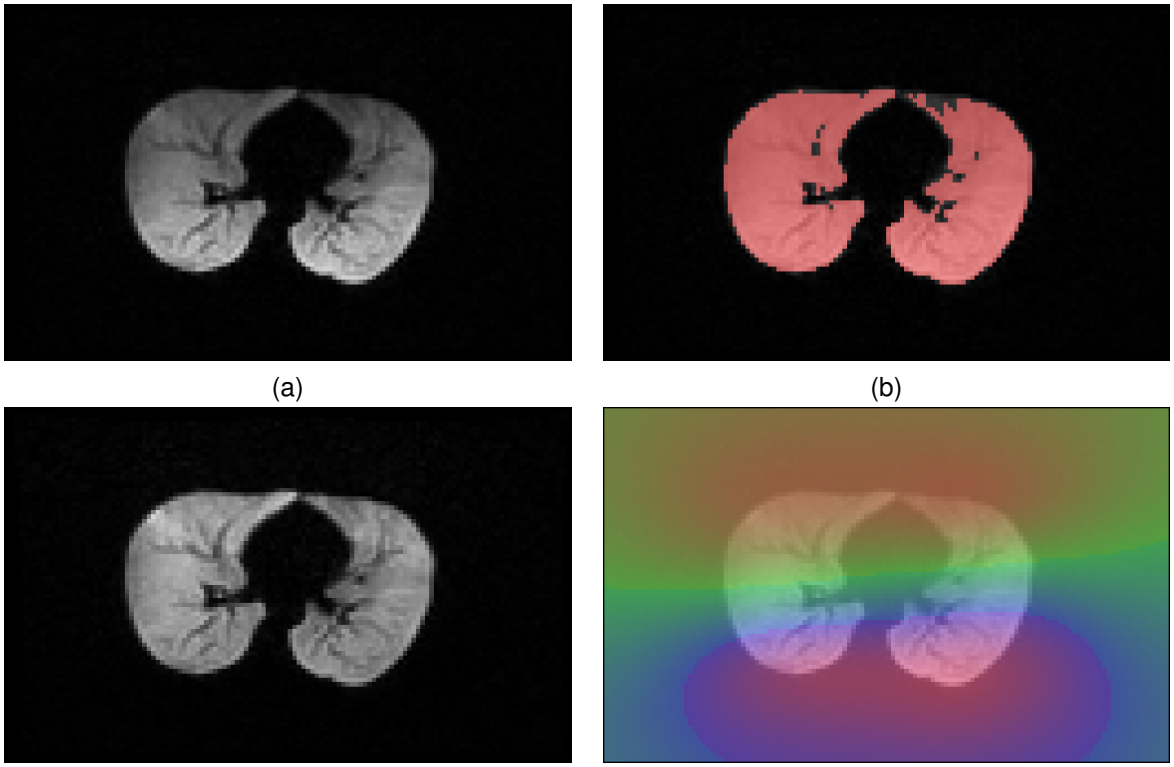
[5] N. J. Tustison, H. Zhang, G. Lehmann, P. Yushkevich, and J. C. Gee. Meeting andy warhol somewhere over the rainbow: Rgb colormapping and itk. *Insight Journal*, 2008. 3



(a) (b)

(c) (d)

Figure 1: (a) Uncorrected image. (b) Mask image. (c) Bias field corrected image. (d) Uncorrected image with the calculated bias field superimposed.



(a) (b)

(c) (d)

Figure 2: (a) Uncorrected image. (b) Mask image. (c) Bias field corrected image. (d) Uncorrected image with the calculated bias field superimposed.