
MR Parameter Map Suite: ITK Classes for Calculating Magnetic Resonance T_2 and T_1 Parameter Maps

Release 1.00

Don C. Bigler¹, Mark Meadowcroft¹, Xiaoyu Sun¹, Jeffrey J. Vesek¹, M. Alex Dresner², Michael B. Smith³, and Qing X. Yang¹

June 2, 2008

¹Penn State Center for NMR Research, Hershey PA

²Philips Healthcare, Philadelphia PA

³Novartis Institutes for Biomedical Research, Inc., Cambridge MA

Abstract

This document describes a suite of new multi-threaded classes for calculating magnetic resonance (MR) T_2 and T_1 parameter maps implemented using the Insight Toolkit ITK (www.itk.org). Similar to MR diffusion tensor imaging (DTI), MR T_2 and T_1 parameter maps provide a non-invasive means for quantitatively measuring disease or pathology *in-vivo*. Included in the suite are classes for reading proprietary Bruker 2dseq and Philips PAR/REC images and example programs and data for validating the new classes.

Contents

1	Background	2
1.1	Measuring T_2 Relaxation	2
1.2	Measuring T_1 Relaxation	3
2	Implementation	3
2.1	itk::MRT2ParameterMap3DImageFilter	3
2.2	itk::MRT1ParameterMap3DImageFilter	5
2.3	itk::Bruker2DSEQImageIO	7
2.4	itk::PhilipsRECImageIO	9
3	Examples	10
3.1	Bruker 2dseq T_2 Parameter Map	11
3.2	Bruker 2dseq T_1 Parameter Map	16
3.3	Philips REC T_2 Parameter Map	20
3.4	Philips REC T_1 Parameter Map	25
4	Conclusion	31

1 Background

Magnetic resonance imaging (MRI) is a powerful, non-invasive, medical imaging modality capable of producing a wide variety of image contrasts. In addition to standard T_1 - and T_2 -weighted contrast images, MRI is also capable of producing quantitative measurements of magnetic resonance (MR) specific or derived parameters. Typically these quantitative measurements are not acquired directly from the MR scanner, but are calculated with multiple image volumes using a specific set of experimental parameters. For example, the ITK `itk::DiffusionTensor3DReconstructionImageFilter` takes as input multiple MR image volumes acquired with varying gradient directions and strengths. The `itk::TensorFractionalAnisotropyImageFilter` converts its tensor output into the fractional anisotropy scalar quantity, which is a measure of the shape of the diffusion of water (derived parameter).

Similar to diffusion tensor imaging (DTI), the T_2 and T_1 parameter maps or their inverses, R_2 and R_1 , are calculated using multiple image acquisitions with varying parameters. Whereas DTI indirectly measures the directional distribution of the diffusion of water, T_2 and T_1 parameter maps are exponential relaxation times specific to the tissue or sample that is being imaged. Increasingly whole brain MR parameter maps are used to perform statistical parametric mapping studies to characterize and monitor neurodegenerative disease or pathology. For example, this technique, termed voxel-based relaxometry (VBR), was applied to studies of epilepsy to show increases in T_2 within the hippocampus, parahippocampal gyrus, and anterior temporal lobe white matter, when compared with normal controls [13, 12]. A similar study was performed for patients suffering from multiple-system atrophy of cerebellar type [15]. In this study, the inverse of T_2 , R_2 , was shown to decrease in the cerebellum and brainstem and increase in the putamen. Also, preliminary work in amyotrophic lateral sclerosis (ALS) demonstrated increases in T_2 within the subthalamic region when compared with age-matched normal controls [1]. Statistical parametric maps require the brain images to be registered to a reference brain atlas - a task well suited for ITK. However, currently ITK does not provide classes for calculating T_2 and T_1 parameter maps.

1.1 Measuring T_2 Relaxation

T_2 is conventionally measured by acquiring multiple spin-echo images. The following equation describes the signal for a voxel acquired using a spin-echo sequence:

$$S \propto \rho e^{\left(\frac{-TE}{T_2}\right)} \left[1 - e^{\left(\frac{-TR}{T_1}\right)} \right] \quad (1)$$

where ρ is the proton density, TE is the time to echo, and TR is the time to repetition. To calculate T_2 , TR is set to a long value, $\geq 5T_1$, and TE is varied over the multiple spin-echo image acquisitions. Since ρ is assumed constant Equation 1 reduces to the following:

$$S_i(t) = S_0 e^{\frac{-TE_i}{T_2}} \quad (2)$$

where $S_0 = \rho \left[1 - e^{\left(\frac{-TR}{T_1}\right)} \right]$ and i refers to the i th image with echo time TE_i . Because $S_i(t)$ and TE_i are known, S_0 and T_2 can be estimated using a linear least squares or nonlinear fit. Although not described in

detail in this paper, the estimation of T_2^* is the same as T_2 . The difference is that the images are acquired using a gradient echo sequence instead of a spin-echo sequence.

1.2 Measuring T_1 Relaxation

Many image acquisition methods exist for measuring T_1 [9]. The signal obtained when using the steady-state or saturation recovery approach is the same as Equation 1. However to measure T_1 , TE is set as small as possible and TR is varied. The nonlinear fitting in this case becomes:

$$\mathcal{S}_i(t) = S_0 \left[1 - e^{\left(\frac{-TR_i}{T_1}\right)} \right] \quad (3)$$

where $S_0 = \rho e^{\frac{-TE}{T_2}}$ and i refers to the i th image with repetition time TR_i . T_1 may also be measured using the inversion recovery sequence. The inversion recovery sequence is a spin-echo sequence with an additional 180 degree radiofrequency (RF) pulse at the beginning of the sequence. Again, the fit is nonlinear and changes to the following:

$$\mathcal{S}_i(t) = S_0 \left[1 - 2e^{\left(\frac{-TI_i}{T_1}\right)} \right] \quad (4)$$

where $S_0 = \rho e^{\frac{-TE}{T_2}}$ and i refers to the i th image with inversion time TI_i . Both the steady-state and inversion recovery methods require long TR or TI times in order to obtain good estimates of T_1 and therefore require long scan times. Since long scan times are not suitable for human studies, fast T_1 -mapping protocols have been developed to overcome this problem [10, 2, 5, 11, 17, 19, 3, 6, 7, 14, 16, 4, 18, 8]. Many fast methods are variants of the Look-Locker or TOMROP technique [10, 5]. The `itk::MRT1ParameterMap3DImageFilter` class described in the next section (2.2) provides an implementation of the Look-Locker method by Deichmann and Haase [2]. T_1 is not measured directly, but a slightly lower value T_1^* . T_1 is estimated after performing a fit for T_1^* (Equation 13).

2 Implementation

This section describes the multi-threaded ITK classes for calculating the parameter maps. Since the classes are derived from the `itk::ImageToImageFilter`, they can easily be integrated with a registration pipeline. In addition to the classes for calculating T_2 and T_1 , classes for reading proprietary Bruker 2dseq (2.3) and Philips PAR/REC (2.4) MR image files are described. The examples outlined in the next section (3.1, 3.2, 3.3, and 3.4) use the new readers to calculate T_2 and T_1 parameter maps from the original proprietary image data formats.

2.1 `itk::MRT2ParameterMap3DImageFilter`

The interface of `itk::MRT2ParameterMap3DImageFilter` was designed to be similar with `itk::DiffusionTensor3DReconstructionImageFilter`. As such, this filter is templated over the input pixel type and the output pixel type. The input T_2 -weighted MR images must all have the same size and dimensions. The 3D `itk::VectorImage` output will always have four components. The first component of the output will be the T_2 time in seconds (R_2 in Hz if `PerformR2MappingOn()` is called),

the second component will be the constant A as shown in the equations below, and the fourth component will be the R-squared value from the curve fitting. The third component will vary depending on the type of T_2 fitting selected using `SetAlgorithm()`. For `LINEAR` and `NON_LINEAR` the third component will be zero. For `NON_LINEAR_WITH_CONSTANT` the third component will be the value C as shown below.

- `LINEAR` (Linear least squares):

$$S_i(t) = Ae^{\left(\frac{-TE_i}{T_2}\right)} \quad (5)$$

- `NON_LINEAR` (Non-linear least squares using Levenberg-Marquardt): Same as Equation 5.
- `NON_LINEAR_WITH_CONSTANT` (Non-linear least squares using Levenberg-Marquardt):

$$S_i(t) = Ae^{\left(\frac{-TE_i}{T_2}\right)} + C \quad (6)$$

There are two ways to use this class. When multiple T_2 or T_2^* -weighted images are available the images are added as follows:

```
filter->AddMREchoImage( echoTime1, image1 );
filter->AddMREchoImage( echoTime2, image2 );
...
```

When the 'n' T_2 or T_2^* -weighted images are stored in a single multi-component image (`itk::VectorImage`), use `SetMREchoImage()` like this:

```
filter->SetMREchoImage( echoTimeContainer, vectorImage );
...
```

A number of Get/Set methods are provided for controlling the output of the filter:

- `Algorithm` - Set/Get the T_2 fitting algorithm used (`LINEAR`, `NON_LINEAR`, and `NON_LINEAR_WITH_CONSTANT`).
- `MaxT2Time` - Set the maximum T_2 time (T_2 times greater than or equal to this value will be set to this value).
- `PerformR2Mapping` - If On R_2 (in Hz) will be calculated instead of T_2 .

Multi-threaded fitting is performed over every voxel in the output image region across the entire image time series using the overridden `void ThreadedGenerateData()` function. Linear least squares fitting provides starting parameters for the non-linear algorithms. However, the input time series image data must be linearized first before a linear least squares fit is possible. `void FitLinearExponential()` handles this task:

```
template< class TMREchoImagePixelType, class TMRParameterMapImagePixelType >
void
MRT2ParameterMap3DImageFilter<TMREchoImagePixelType,TMRParameterMapImagePixelType>
::FitLinearExponential(ExponentialFitType X, ExponentialFitType Y, unsigned int num,
```

```

    MRParameterMapPixelType &output)
{
    EchoTimeType Sumxy=0, Sumx=0, Sumy=0, Sumx2=0, Sumy2=0, b=0, denom=0;

    for(unsigned int i=0; i<num; i++)
    {
        Sumxy += X[i]*log(Y[i]);
        Sumx += X[i];
        Sumy += log(Y[i]);
        Sumy2 += log(Y[i])*log(Y[i]);
        Sumx2 += X[i]*X[i];
    }
    denom = Sumx2-(Sumx*Sumx/static_cast<EchoTimeType>(num));
    if( denom == 0 )
    {
        b = NumericTraits< EchoTimeType >::max() *
            ((Sumxy-(Sumx*Sumy/static_cast<EchoTimeType>(num))) < 0)?-1.0f:1.0f;
    }
    else
    {
        b = (Sumxy-(Sumx*Sumy/static_cast<EchoTimeType>(num)))/denom;
    }
    if( b == 0 )
    {
        b = NumericTraits< EchoTimeType >::max();
    }
    output[0] = static_cast<typename MRParameterMapPixelType::ValueType>
        (-b); // T2
    output[1] = static_cast<typename MRParameterMapPixelType::ValueType>
        (exp((Sumy-b*Sumx)/static_cast<EchoTimeType>(num))); // Constant
    output[3] = static_cast<typename MRParameterMapPixelType::ValueType>
        ((Sumxy*Sumxy)/(Sumy2*Sumx2)); // R-squared
}

```

Non-linear fitting is accomplished using the `vn1 levenberg_marquardt` class. See the included source code for more details.

2.2 itk::MRT1ParameterMap3DImageFilter

The interface to `itk::MRT1ParameterMap3DImageFilter` is basically the same as `itk::MRT2ParameterMap3DImageFilter` (2.1). Like `itk::MRT2ParameterMap3DImageFilter` the 3D `itk::VectorImage` output will always have four components. The first component of the output will be the T_1 time in seconds (R_1 in Hz if `PerformR1MappingOn()` is called), the second component will be the constant A as shown in the functions below, and the fourth component will be the R-squared value from the curve fitting. The third component will vary depending on the type of T_1 fitting selected. For all of the 2 parameter models below the third component will be zero. For the remaining models the third component will be the value B as shown below.

- IDEAL_STEADY_STATE (Non-linear least squares using Levenberg-Marquardt):

$$S_i(t) = A \left[1 - e^{\left(\frac{-TR_i}{T_1} \right)} \right] \quad (7)$$

- HYBRID_STEADY_STATE_3PARAM (Non-linear least squares using Levenberg-Marquardt):

$$S_i(t) = A \left[B - e^{\left(\frac{-TR_i}{T_1}\right)} \right] \quad (8)$$

- INVERSION_RECOVERY (Non-linear least squares using Levenberg-Marquardt):

$$S_i(t) = A \left[1 - 2e^{\left(\frac{-TR_i}{T_1}\right)} \right] \quad (9)$$

- INVERSION_RECOVERY_3PARAM (Non-linear least squares using Levenberg-Marquardt):

$$S_i(t) = A \left[1 - Be^{\left(\frac{-TR_i}{T_1}\right)} \right] \quad (10)$$

- ABSOLUTE_INVERSION_RECOVERY (Non-linear least squares using Levenberg-Marquardt):

$$S_i(t) = \left| A \left[1 - 2e^{\left(\frac{-TR_i}{T_1}\right)} \right] \right| \quad (11)$$

- ABSOLUTE_INVERSION_RECOVERY_3PARAM (Non-linear least squares using Levenberg-Marquardt):

$$S_i(t) = \left| A \left[1 - Be^{\left(\frac{-TR_i}{T_1}\right)} \right] \right| \quad (12)$$

- LOOK_LOCKER (Non-linear least squares using Levenberg-Marquardt):

$$S_i(t) = A \left[1 - Be^{\left(\frac{-TR_i}{T_1}\right)} \right], T_1 = (T_1^*)(B - 1) \quad (13)$$

- ABSOLUTE_LOOK_LOCKER (Non-linear least squares using Levenberg-Marquardt):

$$S_i(t) = \left| A \left[1 - Be^{\left(\frac{-TR_i}{T_1}\right)} \right] \right|, T_1 = (T_1^*)(B - 1) \quad (14)$$

The INVERSION_RECOVERY, INVERSION_RECOVERY_3PARAM, and LOOK_LOCKER algorithms require the input data to be the real component of the complex reconstructed MRI. Usually, the image provided by the MR scanner is the magnitude of the complex reconstructed image, which forces the image data to be all positive. If the real image data is not available, the absolute versions of these algorithms may be used instead. The absolute versions attempt to find the data point where the measured relaxation curve goes from negative to positive. This should be the point where the gradient switches from negative to positive as shown in the code below.

```
// Make function non absolute again. This is done by making sure the
// Y values are always increasing.
Yfixed = Y;
for(unsigned int i=0; i<num-1; i++)
{
    // Negative slope
    if( Y[i+1]-Y[i] < 0 )
```

```

{
    Yfixed[i] = -Yfixed[i];
}
// Positive slope
else
{
    break;
}
}

```

This code may fail if the image data is very noisy. In this case, smoothing of the input data will reduce the noise and should eliminate the problem. However, in most instances it should correctly locate the point where the measured relaxation curve goes from negative to positive without requiring smoothing.

Similar to `itk::MRT2ParameterMap3DImageFilter` (2.1), there are two ways to use `itk::MRT1ParameterMap3DImageFilter`. When multiple images are available the images are added as follows:

```

filter->AddMRImage( time1, image1 );
filter->AddMRImage( time2, image2 );
...

```

When the 'n' images are stored in a single multi-component image (`itk::VectorImage`), use `SetMRImage()` like this:

```

filter->SetMRImage( timeContainer, vectorImage );
...

```

Get/Set methods are provided for controlling the output of the filter:

- Algorithm - Set/Get the T_1 fitting algorithm used.
- MaxT1Time - Set the maximum T_1 time (T_1 times greater than or equal to this value will be set to this value).
- PerformR1Mapping - If On R_1 (in Hz) will be calculated instead of T_1 .

Multi-threaded non-linear fitting using the `vnl_levenberg_marquardt` class is performed over every voxel in the output image region across the entire image time series using the overridden `void ThreadedGenerateData()` function. Unlike `itk::MRT2ParameterMap3DImageFilter`, the starting parameters are obtained by making a guess using the input data. The guess depends on the particular algorithm chosen using the call to `SetAlgorithm()` (see the included source code).

2.3 itk::Bruker2DSEQImageIO

The Bruker image format is a very flexible and complex image format. A brief description is found at <http://www.mrc-cbu.cam.ac.uk/Imaging/Common/brukerformat.shtml>.

itk::Bruker2DSEQImageIO provides read-only access to most Bruker 2dseq binary image files via virtual void ReadImageInformation() and virtual void Read(), which are pure virtual methods declared in the parent class `itk::ImageIOBase`. A known limitation is the reading of 2dseq files that contain slices with varying orientations, like a reference scan with three orthogonal slices. For these images itk::Bruker2DSEQImageIO will only read the first 'n' slices that have the same orientation. The other known limitation is that the class cannot handle 4D images. This is in spite of the fact that Bruker 2dseq images can be 4D. The user must determine if the 3D volume read is 4D by examining the parameters in the `itk::MetaDataDictionary`. virtual bool CanReadFile() is also defined and may be used to determine if the Bruker 2dseq is readable. The implementation included with the source code was designed using example images acquired at the Penn State Center for NMR Research (<http://www.hmc.psu.edu/nmr/lab/>), but has not undergone exhaustive testing using all possible Bruker 2dseq image types.

itk::Bruker2DSEQImageIO provides access to important acquisition parameters via the `itk::MetaDataDictionary`. The list of parameters is by no means complete, but other parameters may be added as needed. The following code defines the names of the parameters stored in the `itk::MetaDataDictionary`:

```
extern const char *const RECO_BYTE_ORDER;
extern const char *const RECO_FOV;
extern const char *const RECO_SIZE;
extern const char *const RECO_WORDTYPE;
extern const char *const RECO_IMAGE_TYPE;
extern const char *const RECO_TRANSPOSITION;
extern const char *const ACQ_DIM;
extern const char *const NI/*IMND_N_SLICES*/;
extern const char *const NR;
extern const char *const ACQ_SLICE_THICK/*IMND_SLICE_THICK*/;
extern const char *const NECHOES/*IMND_N_ECHO_IMAGES*/;
extern const char *const ACQ_SLICE_SEPN/*IMND_SLICE_SEPN*/;
extern const char *const ACQ_SLICE_SEPN_MODE;
extern const char *const ACQ_ECHO_TIME;
extern const char *const ACQ_REPETITION_TIME;
extern const char *const ACQ_INVERSION_TIME;
```

Types are also defined for vector quantities stored in the `itk::MetaDataDictionary`:

```
/** Special types used for Bruker meta data. */
typedef VectorContainer< unsigned int, double > RECOFOVContainerType;
typedef VectorContainer< unsigned int, int > RECOTranspositionContainerType;
typedef VectorContainer< unsigned int, double > ACQEchoTimeContainerType;
typedef VectorContainer< unsigned int, double > ACQRepetitionTimeContainerType;
typedef VectorContainer< unsigned int, double > ACQInversionTimeContainerType;
typedef VectorContainer< unsigned int, double > ACQSliceSepnContainerType;
```

The examples in the next section (3.1 and 3.2) show how to use the types with the parameter names to retrieve the values from the `itk::MetaDataDictionary`. Finally, itk::Bruker2DSEQImageIOFactory is also included with the source code for users who wish to use the object factory mechanisms of ITK.

2.4 itk::PhilipsRECImageIO

Like `itk::Bruker2DSEQImageIO` (2.3), `itk::PhilipsRECImageIO` is derived from `itk::ImageIOBase` and implements read-only image access to Philips PAR/REC image data. The *.PAR file describes in human readable text the binary image data stored in the *.REC file. The actual parsing of the PAR parameters is done using the functions defined in `itkPhilipsPAR.h` and `itkPhilipsPAR.cxx`. Currently `itk::PhilipsRECImageIO` supports reading PAR file versions 3 through 4.1, based on sample data acquired at the Penn State Center for NMR Research (<http://www.hmc.psu.edu/nmr/lab/>). Unlike `itk::Bruker2DSEQImageIO`, `itk::PhilipsRECImageIO` supports 4D images.

`itk::PhilipsRECImageIO` also provides access to important acquisition parameters via the `itk::MetaDataDictionary`. The following code defines the names of the parameters stored in the `itk::MetaDataDictionary`:

```
extern const char *const PAR_Version;
extern const char *const PAR_SliceOrientation;
extern const char *const PAR_ExaminationName;
extern const char *const PAR_ProtocolName;
extern const char *const PAR_SeriesType;
extern const char *const PAR_AcquisitionNr;
extern const char *const PAR_ReconstructionNr;
extern const char *const PAR_ScanDuration;
extern const char *const PAR_MaxNumberOfCardiacPhases;
extern const char *const PAR_TriggerTimes;
extern const char *const PAR_MaxNumberOfEchoes;
extern const char *const PAR_EchoTimes;
extern const char *const PAR_MaxNumberOfDynamics;
extern const char *const PAR_MaxNumberOfMixes;
extern const char *const PAR_PatientPosition;
extern const char *const PAR_PreparationDirection;
extern const char *const PAR_Technique;
extern const char *const PAR_ScanMode;
extern const char *const PAR_NumberOfAverages;
extern const char *const PAR_ScanResolution;
extern const char *const PAR_RepetitionTimes;
extern const char *const PAR_ScanPercentage;
extern const char *const PAR_FOV;
extern const char *const PAR_WaterFatShiftPixels;
extern const char *const PAR_AngulationMidSlice;
extern const char *const PAR_OffCentreMidSlice;
extern const char *const PAR_FlowCompensation;
extern const char *const PAR_Presaturation;
extern const char *const PAR_CardiacFrequency;
extern const char *const PAR_MinRRInterval;
extern const char *const PAR_MaxRRInterval;
extern const char *const PAR_PhaseEncodingVelocity;
extern const char *const PAR_MTC;
extern const char *const PAR_SPIR;
extern const char *const PAR_EPIFactor;
extern const char *const PAR_TurboFactor;
extern const char *const PAR_DynamicScan;
extern const char *const PAR_Diffusion;
extern const char *const PAR_DiffusionEchoTime;
```

```
extern const char *const PAR_MaxNumberOfDiffusionValues;
extern const char *const PAR_GradientBValues;
extern const char *const PAR_MaxNumberOfGradientOrients;
extern const char *const PAR_GradientDirectionValues;
extern const char *const PAR_InversionDelay;
extern const char *const PAR_NumberOfImageTypes;
extern const char *const PAR_ImageTypes;
extern const char *const PAR_NumberOfScanningSequences;
extern const char *const PAR_ScanningSequences;
extern const char *const PAR_ScanningSequenceImageTypeRescaleValues;
```

Types are also defined for vector quantities stored in the [itk::MetaDataDictionary](#):

```
/** Special types used for Philips PAR meta data. */
typedef VectorContainer< unsigned int, double > EchoTimesContainerType;
typedef VectorContainer< unsigned int, double > TriggerTimesContainerType;
typedef VectorContainer< unsigned int, double > RepetitionTimesContainerType;
typedef vnl_vector_fixed< int, 2 > ScanResolutionType;
typedef vnl_vector_fixed< float, 3 > FOVType;
typedef vnl_vector_fixed< double, 3 > AngulationMidSliceType;
typedef vnl_vector_fixed< double, 3 > OffCentreMidSliceType;
typedef vnl_vector_fixed< float, 3 > PhaseEncodingVelocityType;
/** Image types: 0=Magnitude, 1=Real, 2=Imaginary, 3=Phase, & 4=Special/Processed. */
typedef vnl_vector_fixed< int, 8 > ImageTypesType;
typedef vnl_vector_fixed< int, 8 > ScanningSequencesType;
typedef std::vector< int > SliceIndexType;
typedef vnl_vector_fixed< double, 3 > ImageTypeRescaleValuesType;
typedef VectorContainer< unsigned int, ImageTypeRescaleValuesType >
ImageTypeRescaleValuesContainerType;
typedef VectorContainer< unsigned int, ImageTypeRescaleValuesContainerType::Pointer >
ScanningSequenceImageTypeRescaleValuesContainerType;
typedef double GradientBValueType;
typedef VectorContainer< unsigned int, GradientBValueType > GradientBValueContainerType;
typedef vnl_vector_fixed< double, 3 > GradientDirectionType;
typedef VectorContainer< unsigned int, GradientDirectionType >
GradientDirectionContainerType;
```

The examples in the next section (3.3 and 3.4) show how to use the types with the parameter names to retrieve the values from the [itk::MetaDataDictionary](#). Again, [itk::PhilipsRECImageIOFactory](#) is also included with the source code for users who wish to use the object factory mechanisms of ITK.

3 Examples

The examples that follow demonstrate how to use the classes described in the previous sections. The data and source for these examples are included with this article. The Bruker data directory contains both a T_2 data set and a T_1 saturation recovery data set. A phantom containing 14 test tubes with varied concentrations of agar and gadopentetate dimeglumine (Gd-DPTA) was used for both data sets. Table 1 shows the concentrations for each tube.

Two separate phantoms were used for the Philips T_2 and T_1 examples. The phantom for the T_2 example was designed to mimic brain white matter (WM) and gray matter (GM). A cylindrical container was filled with

Table 1: Gd-DTPA and agar concentrations for phantom used in Bruker examples.

Tube #	% Gd-DTPA	Agar (g/ml)
1	0.0229	0.054560
2	0.0229	0.038265
3	0.0229	0.029345
4	0	0.019065
5	0.0103	0.025325
6	0.0153	0.025615
7	0.0229	0.023750
8	0.0084	0.033580
9	0.0154	0.035165
10	0.0280	0.030315
11	0.0510	0.010755
12	0.0084	0.027200
13	0.0084	0.022890
14	0.0084	0.019765

2.6224×10^{-2} (g/ml) concentration agar doped with 0.024% Gd-DTPA (white matter) and embedded within the container were 17 plastic spheres filled with 1.8382×10^{-2} (g/ml) agar doped with 0.007% Gd-DTPA (gray matter). The T_1 phantom contains 5 tubes with varying amounts of Gd-DTPA. Table 2 shows the concentration of Gd-DTPA for each numbered tube.

Table 2: Gd-DTPA concentrations for phantom used in Philips T_1 example.

Tube #	% Gd-DTPA
1	0.043
2	0.024
3	0.014
4	0.008
5	0

3.1 Bruker 2dseq T_2 Parameter Map

In this example a T_2 parameter map is generated using a multi-echo spin-echo sequence acquired on a 3.0 T Bruker MedSpec MR system. The TR time was 1595.03 ms and 11 echo images were acquired with a starting TE of 7.919 ms and an echo spacing of 7.919 ms (final echo time was 87.107 ms). A single 3 mm slice was acquired with a 256 x 256 matrix size and 20 cm x 20 cm field of view (FOV). The data for this example is located in the DCB021304.oj1/3/pdata/1/ folder. The source code is located in the BrukerT2Map.cxx file.

BrukerT2Map.cxx takes up to 9 arguments. The first 5 arguments are required and are used to specify the full path of the input Bruker 2dseq, the output T_2 parameter map filename, the output exponential constant (S_0) map filename, the output constant map (C) filename, and the output R-squared map filename. The remaining 4 optional arguments are used to control the output, specifically whether or not to output T_2 or R_2 (default is T_2), the algorithm to use (default is LINEAR), the maximum T_2 time (default is 10 seconds), and a baseline threshold value for masking the input image (default is 0). The optional parameters used for this

example are 0 (output T_2), 0 (use LINEAR algorithm Equation 5), 2.0 (maximum T_2 time of 2.0 seconds), and 1000 (every voxel less than 1000 set to zero).

After checking for the correct number of arguments and reading the arguments from the command line, the program attempts to read the Bruker 2dseq and get the image information:

```
// Create 2DSEQ reader and check the file if it can be read.
Bruker2DSEQImageIOType::Pointer imageIO = Bruker2DSEQImageIOType::New();
if( !imageIO->CanReadFile(inputFilename) )
{
    std::cerr << "Could not read 2dseq file" << std::endl;
    return 1;
}

// Read the image information.
imageIO->SetFileName(inputFilename);
try
{
    imageIO->ReadImageInformation();
}
catch( itk::ExceptionObject &err )
{
    std::cerr << "ExceptionObject caught";
    std::cerr << " : " << err.GetDescription();
    return 1;
}
```

If the image is read without errors, the program will attempt to get the echo times from the [itk::MetaDataDictionary](#):

```
// Get the echo times in ms.
Bruker2DSEQImageIOType::ACQEchoTimeContainerType::Pointer ptrToEchoes = NULL;
if( !itk::ExposeMetaData<Bruker2DSEQImageIOType::ACQEchoTimeContainerType::Pointer>
    (imageIO->GetMetaDataDictionary(), itk::ACQ_ECHO_TIME, ptrToEchoes) )
{
    std::cerr << "Could not get the echo times" << std::endl;
    return 1;
}
if( !ptrToEchoes )
{
    std::cerr << "Received NULL echo times pointer from meta dictionary" << std::endl;
    return 1;
}
unsigned int numberOfEchoTimes = ptrToEchoes->Size();
```

Instead of acquiring multiple spin-echo images with different TE times, typically a multi-echo spin-echo image is acquired to measure T_2 . A spin-echo image is acquired using a 90 degree RF pulse followed by a 180 degree RF pulse. To acquire a multi-echo data set a series of 180 degree RF pulses follow the first 90 and 180 degree pulses. The application of a perfectly homogenous 180 RF pulse across a sample is difficult and imperfections in the flip-angle lead to what are called stimulated echoes for the echo images acquired after the first 180 degree RF pulse. Therefore in order to measure T_2 the first echo image without the stimulated

echo signal is usually thrown out. The following for loop below will do that as well as convert the echo times to seconds:

```
for(unsigned int i=1; i<(unsigned int)numberOfEchoTimes; i++)
{
    ptrToEchoes->SetElement(i-1,ptrToEchoes->ElementAt(i)/1000.0f); // convert to seconds
}
ptrToEchoes->CastToSTLContainer().pop_back();
--numberOfEchoTimes;
if( numberOfEchoTimes < 2 )
{
    std::cerr << "Must have at least 2 echo images to calculate T2" << std::endl;
    return 1;
}
```

As already mentioned previously, `itk::Bruker2DSEQImageIO` is limited to reading Bruker 2dseq images in 3D only. For a multi-echo spin-echo image, the 2dseq is in reality a 4D image with the fourth dimension being the echo time. The code segment below will convert the 3D image to a `itk::VectorImage`, which will be supplied as an input to `itk::MRT2ParameterMap3DImageFilter`:

```
// Get real number of slices and create vector image.
// Also threshold the image at the same time.
int realSlices = dims[2]/(numberOfEchoTimes+1);
VectorImageType::RegionType region;
VectorImageType::SizeType size;
size[0] = dims[0];
size[1] = dims[1];
size[2] = realSlices;
VectorImageType::IndexType index;
index[0] = 0;
index[1] = 0;
index[2] = 0;
region.SetSize(size);
region.SetIndex(index);
VectorImageType::Pointer vectorImage = VectorImageType::New();
vectorImage->SetRegions(region);
vectorImage->SetVectorLength(numberOfEchoTimes);
VectorImageType::PointType origin = baselineReader->GetOutput()->GetOrigin();
origin[2] = -baselineReader->GetOutput()->GetSpacing()[2]*size[2]/2.0f;
vectorImage->SetOrigin(origin);
vectorImage->SetSpacing(baselineReader->GetOutput()->GetSpacing());
vectorImage->SetDirection(baselineReader->GetOutput()->GetDirection());
vectorImage->Allocate();
ImageType::IndexType echoIndex;
for(index[0]=0,echoIndex[0]=0;
    index[0]<(int)size[0];
    index[0]++,echoIndex[0]++)
{
    for(index[1]=0,echoIndex[1]=0;
        index[1]<(int)size[1];
        index[1]++,echoIndex[1]++)
    {
```

```

for( index[2]=0;
    index[2]<(int)size[2];
    index[2]++ )
{
    // Multi-echo Bruker images are stored as they are acquired. This
    // means that the images are not stored as volumes. We need to put
    // each echo from each slice into the vector as follows:
    echoIndex[2] = index[2]*(numberOfEchoTimes+1) + 1; // Skip to next slice ignoring the first echo.
    VectorImageType::PixelType echoVector(numberOfEchoTimes);
    for(unsigned int echo=0; echo<numberOfEchoTimes; echo++)
    {
        ImageType::PixelType pixelVal
            = baselineReader->GetOutput()->GetPixel(echoIndex);
        echoVector[echo] = (pixelVal < threshold)?0:pixelVal;
        ++echoIndex[2];
    }
    vectorImage->SetPixel(index, echoVector);
}
}
}

```

The fitting options and `itk::VectorImage` input data are input into `itk::MRT2ParameterMap3DImageFilter` as follows:

```

// Create T2 mapping class.
MRT2ParameterMap3DImageFilterType::Pointer t2Map
    = MRT2ParameterMap3DImageFilterType::New();
// Select the fit type.
switch(algorithm)
{
    case MRT2ParameterMap3DImageFilterType::LINEAR:
        t2Map->SetAlgorithm(MRT2ParameterMap3DImageFilterType::LINEAR);
        break;
    case MRT2ParameterMap3DImageFilterType::NON_LINEAR:
        t2Map->SetAlgorithm(MRT2ParameterMap3DImageFilterType::NON_LINEAR);
        break;
    case MRT2ParameterMap3DImageFilterType::NON_LINEAR_WITH_CONSTANT:
        t2Map->SetAlgorithm(MRT2ParameterMap3DImageFilterType::NON_LINEAR_WITH_CONSTANT);
        break;
    default:
        std::cerr << "In valid algorithm = " << algorithm << std::endl;
        return 1;
}
t2Map->SetMaxT2Time(maxT2Time);
t2Map->SetMREchoImage(ptrToEchoes, vectorImage);
if( r2Mapping )
{
    t2Map->PerformR2MappingOn();
}

```

Finally, the `itk::VectorImage` output is connected to the input of `itk::VectorIndexSelectionCastImageFilter` so that the extracted components may be written to file with the file names supplied on the command line:

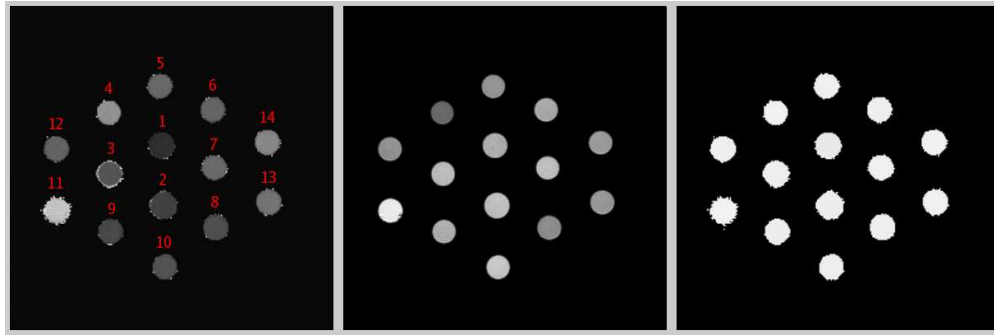


Figure 1: Output from Bruker T_2 parameter map example. The image on the left is the T_2 parameter map, the middle image is the constant S_0 , and the right image is the R-squared map. The tube numbers from Table 1 are labeled in red on the T_2 parameter map image. Note that some contrast exists in the constant image S_0 . This indicates that the TR of 1595.03 ms was not long enough to remove the contributions due to T_1 in Equation 1.

```
// Extract each output component and write to disk.
VectorIndexSelectionCastImageFilterType::Pointer extractComp
= VectorIndexSelectionCastImageFilterType::New();
extractComp->SetInput(t2Map->GetOutput());
WriterType::Pointer writer = WriterType::New();
writer->SetInput(extractComp->GetOutput());
```

Figure 1 shows the T_2 , exponent constant, and R-squared output images for this example and Table 3 contains the average T_2 time, exponent constant, and R-squared value for each tube. To verify the fitted values, the magnitude of the MR signal at the center of tube 1 was plotted for each echo image as a function of echo time and the fitted function was added as an overlay in Figure 2. As can be seen in the figure, the fitted values match the plotted MR signal values very well.

Table 3: Average T_2 , S_0 , and R-squared value for each tube in the Bruker T_2 phantom example.

Tube #	Average T_2 (ms)	Average S_0	Average R-squared
1	38.147 ± 0.982	23567 ± 594	0.783 ± 0.002
2	50.697 ± 1.088	25385 ± 690	0.799 ± 0.001
3	62.540 ± 1.344	25083 ± 459	0.807 ± 0.001
4	99.065 ± 2.733	16889 ± 395	0.818 ± 0.001
5	74.704 ± 1.548	21102 ± 520	0.812 ± 0.001
6	72.822 ± 1.442	23036 ± 454	0.811 ± 0.001
7	75.776 ± 2.633	24721 ± 2778	0.812 ± 0.002
8	59.594 ± 1.511	20244 ± 513	0.804 ± 0.001
9	55.987 ± 1.256	23424 ± 613	0.802 ± 0.001
10	61.708 ± 1.233	26303 ± 693	0.806 ± 0.001
11	129.920 ± 4.652	31113 ± 1307	0.823 ± 0.001
12	71.762 ± 1.579	20737 ± 475	0.810 ± 0.001
13	81.658 ± 2.013	21333 ± 532	0.814 ± 0.001
14	93.423 ± 2.056	21607 ± 471	0.817 ± 0.000

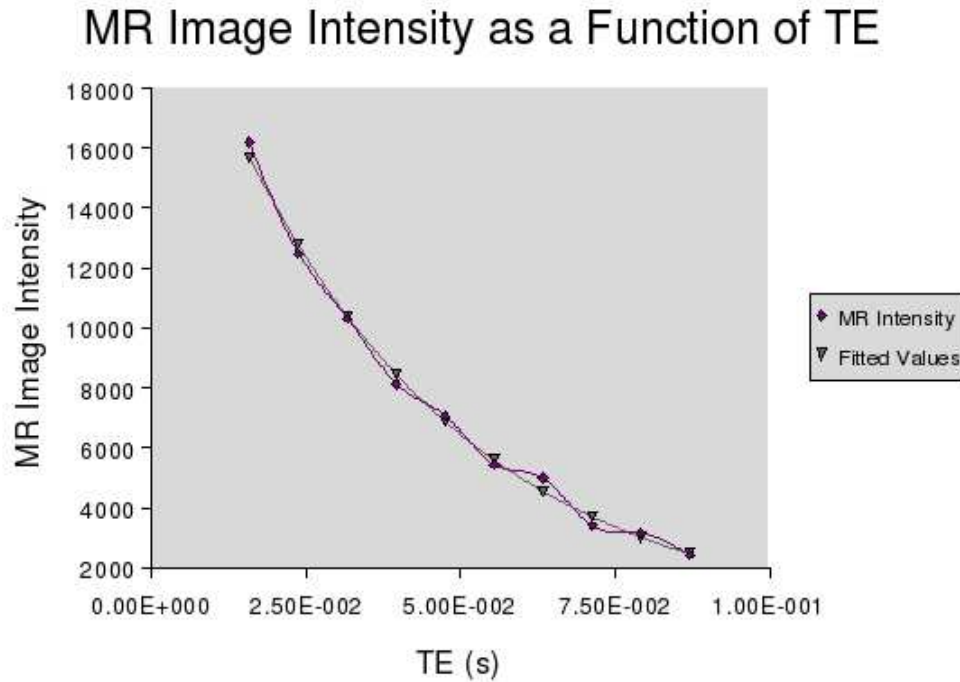


Figure 2: A plot of image intensity (S_i) as a function of TE for a pixel in the center of tube 1 with overlay of fitted curve. Visually, the fitted curve matches the experimental data very well.

3.2 Bruker 2dseq T_1 Parameter Map

This example calculates a T_1 parameter map using the saturation recovery method acquired on a 3.0 T Bruker MedSpec MR system. The TE time was 10 ms and 11 images were acquired with $TR = 30, 50, 100, 200, 500, 1000, 2000, 3000, 4000, 6000, \text{ and } 10000$ ms. The single slice acquisition thickness was 10 mm. The matrix size was 128 x 128 and the FOV 20 cm x 20 cm. The data for this example is located in the DCB021304.oj1/5/pdata/1/ folder. The source code is located in the BrukerT1Map.cxx file.

The arguments to BrukerT1Map.cxx are essentially the same as BrukerT2Map.cxx (3.1). The optional parameters used for this example are 0 (output T_1), 5 (use HYBRID_STEADY_STATE_3PARAM algorithm Equation 8), 5.0 (maximum T_1 time of 5.0 seconds), and 20546109 (every voxel less than 20546109 set to zero).

After checking for the correct number of arguments and reading the arguments from the command line, the program attempts to read the Bruker 2dseq and get the image information - the same as in BrukerT2Map.cxx (3.1). After successfully reading the image information the program will extract the repetition times or the inversion times depending on the fit algorithm specified on the command line and convert the times to seconds:

```
// Extract repetition/inversion times depending on the algorithm.
Bruker2DSEQImageIOType::ACQRepetitionTimeContainerType::Pointer ptrToTimePoints = NULL;
if( (algorithm == MRT1ParameterMap3DImageFilterType::IDEAL_STEADY_STATE) ||
    (algorithm == MRT1ParameterMap3DImageFilterType::HYBRID_STEADY_STATE_3PARAM) )
{
    // Repetition times for the saturation recovery fit type.
```

```

    if( !itk::ExposeMetaData<Bruker2DSEQImageIOType::ACQRepetitionTimeContainerType::Pointer>
        (imageIO->GetMetaDataDictionary(), itk::ACQ_REPETITION_TIME, ptrToTimePoints) )
    {
        std::cerr << "Could not get the repetition times" << std::endl;
        return 1;
    }
}
else
{
    // Inversion times for all others.
    if( !itk::ExposeMetaData<Bruker2DSEQImageIOType::ACQRepetitionTimeContainerType::Pointer>
        (imageIO->GetMetaDataDictionary(), itk::ACQ_INVERSION_TIME, ptrToTimePoints) )
    {
        std::cerr << "Could not get the inversion times" << std::endl;
        return 1;
    }
}
if( !ptrToTimePoints )
{
    std::cerr << "Received NULL repetition/inversion times pointer";
    std::cerr << " from meta dictionary" << std::endl;
    return 1;
}
unsigned int numberOfTimePoints = ptrToTimePoints->Size();
if( numberOfTimePoints < 2 )
{
    std::cerr << "Must have at least 2 images to calculate T1" << std::endl;
    return 1;
}

// Convert the times to seconds.
for(unsigned int i=0; i<(unsigned int)numberOfTimePoints; i++)
{
    ptrToTimePoints->SetElement(i, ptrToTimePoints->ElementAt(i)/1000.0f); // convert to seconds
}

```

If this task completes without errors, the image is read and then a [itk::VectorImage](#) is created to store the data series. Instead of thresholding the entire data series using the threshold value, just the first image volume in the series is thresholded and used as a mask when the data is copied to the [itk::VectorImage](#):

```

// Get real number of slices and create vector image.
// Also threshold the image at the same time.
int realSlices = dims[2]/numberOfTimePoints;
VectorImageType::RegionType region;
VectorImageType::SizeType size;
size[0] = dims[0];
size[1] = dims[1];
size[2] = realSlices;
VectorImageType::IndexType index;
index[0] = 0;
index[1] = 0;
index[2] = 0;
region.SetSize(size);

```

```

region.SetIndex(index);
VectorImageType::Pointer vectorImage = VectorImageType::New();
vectorImage->SetRegions(region);
vectorImage->SetVectorLength(numberOfTimePoints);
VectorImageType::PointType origin = baselineReader->GetOutput()->GetOrigin();
origin[2] = -baselineReader->GetOutput()->GetSpacing()[2]*size[2]/2.0f;
vectorImage->SetOrigin(origin);
vectorImage->SetSpacing(baselineReader->GetOutput()->GetSpacing());
vectorImage->SetDirection(baselineReader->GetOutput()->GetDirection());
vectorImage->Allocate();
ImageType::IndexType imageTimePointIndex;
for(index[0]=0, imageTimePointIndex[0]=0;
    index[0]<(int)size[0];
    index[0]++, imageTimePointIndex[0]++)
{
    for(index[1]=0, imageTimePointIndex[1]=0;
        index[1]<(int)size[1];
        index[1]++, imageTimePointIndex[1]++)
    {
        for( index[2]=0;
            index[2]<(int)size[2];
            index[2]++ )
        {
            // Multi-echo inversion recovery Bruker images are stored as they are acquired.
            // This means that the images are not stored as volumes. We need to put
            // each echo from each slice into the vector as follows:
            if( (algorithm != MRT1ParameterMap3DImageFilterType::IDEAL_STEADY_STATE) &&
                (algorithm != MRT1ParameterMap3DImageFilterType::HYBRID_STEADY_STATE_3PARAM) )
            {
                imageTimePointIndex[2] = index[2]*numberOfTimePoints; // Skip to next slice
            }
            VectorImageType::PixelType timePointVector(numberOfTimePoints);
            for(unsigned int timePoint=0; timePoint<numberOfTimePoints; timePoint++)
            {
                // Images are stored as volumes in the saturation recovery case, so we need
                // to skip to each new repetition while filling the vector.
                if( (algorithm == MRT1ParameterMap3DImageFilterType::IDEAL_STEADY_STATE) ||
                    (algorithm == MRT1ParameterMap3DImageFilterType::HYBRID_STEADY_STATE_3PARAM) )
                {
                    imageTimePointIndex[2] = index[2] + (realSlices*timePoint);
                }
                ImageType::PixelType pixelVal
                    = baselineReader->GetOutput()->GetPixel(imageTimePointIndex);
                ImageType::PixelType maskVal = 0;
                if( (algorithm == MRT1ParameterMap3DImageFilterType::IDEAL_STEADY_STATE) ||
                    (algorithm == MRT1ParameterMap3DImageFilterType::HYBRID_STEADY_STATE_3PARAM) )
                {
                    maskVal = t1Mask->GetOutput()->GetPixel(index);
                }
                else
                {
                    VectorImageType::IndexType tempIndex = index;
                    tempIndex[2] = tempIndex[2]*numberOfTimePoints; // Skip to next slice
                    maskVal = t1Mask->GetOutput()->GetPixel(tempIndex);
                }
            }
        }
    }
}

```

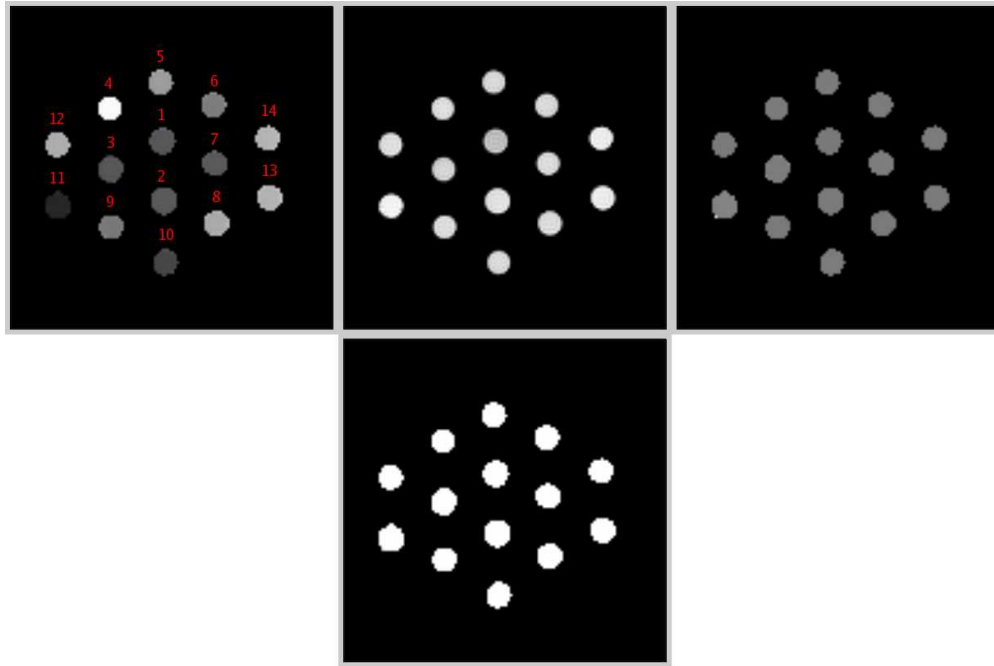


Figure 3: Output from Bruker T_1 parameter map example. The image on the top left is the T_1 parameter map, the top middle image is the constant S_0 , the top right image is the constant B , and the bottom middle image is the R-squared map. The tube numbers from Table 1 are labeled in red on the T_1 parameter map image. Unlike Figure 1, very little contrast exists in the constant image S_0 . This indicates that the TE of 10 ms was short enough to remove the contributions due to T_2 in Equation 1.

```

    }
    timePointVector[timePoint] = (maskVal==0)?0:pixelVal;
    if( (algorithm != MRT1ParameterMap3DImageFilterType::IDEAL_STEADY_STATE) &&
        (algorithm != MRT1ParameterMap3DImageFilterType::HYBRID_STEADY_STATE_3PARAM) )
    {
        ++imageTimePointIndex[2];
    }
}
vectorImage->SetPixel(index, timePointVector);
}
}
}

```

Finally, the T_1 map is calculated, the components extracted, and then written to file - the same as in the Bruker T_2 map example (3.1). Figure 3 shows the T_1 , exponent constant (S_0), constant (B), and R-squared output images for this example and Table 4 contains the average T_1 time, exponent constant, constant, and R-squared value for each tube. To verify the fitted values, the magnitude of the MR signal at the center of tube 1 was plotted for each image repetition as a function of repetition time and the fitted function was added as an overlay in Figure 4. Again, as can be seen in the figure, the fitted values match the plotted MR signal values very well.

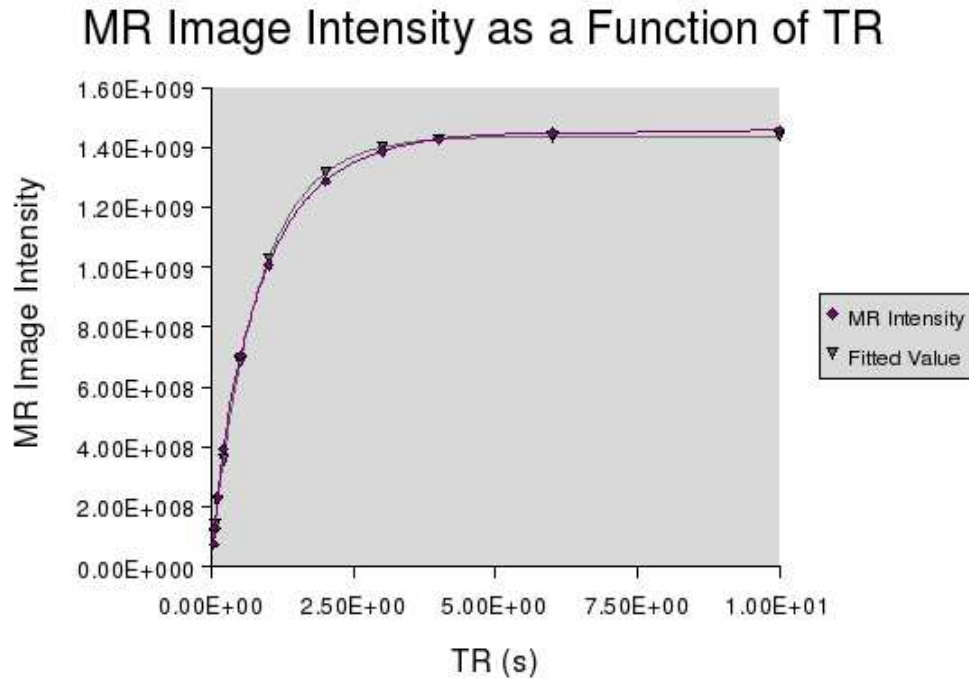


Figure 4: A plot of image intensity (S_i) as a function of TR for a pixel in the center of tube 1 with overlay of fitted curve. Again, this demonstrates that the predicted data is a good fit for the experimental data.

3.3 Philips REC T_2 Parameter Map

The previous examples (3.1 and 3.2) used the Bruker data to generate T_2 and T_1 parameter maps. In this example a Philips multi-echo spin-echo PAR/REC data set is used to calculate T_2 . The name of the file is T2_Map_4_3.PAR and the example code is saved in PhilipsT2Map.cxx. The image was acquired on a 3.0 T Philips Achieva system. A single 4 mm axial slice with SENSE factor 2.5, 256 x 256 matrix size, 230 mm x 230 mm FOV, TR of 252.901 ms, and 14 echoes (TE 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, and 112 ms) was acquired. The 252.901 ms TR time is not ideal for measuring T_2 , as it will include significant T_1 contributions and will reduce the overall signal to noise ratio (SNR) of the image. However, this is ideal for testing the non-linear fitting capabilities of `itk::MRT2ParameterMap3DImageFilter` used in this example.

PhilipsT2Map.cxx takes the same 9 arguments as BrukerT2Map.cxx (3.1). For this example the optional parameters are 0 (output T_2), 1 (use NON_LINEAR algorithm Equation ??), 2.0 (maximum T_2 time of 2.0 seconds), and 14 (every voxel less than 14 set to zero).

After successfully reading the image information using `itk::PhilipsRECImageIO`, PhilipsT2Map.cxx checks to make sure that the number of image volumes is greater than or equal to the number of echo times:

```
unsigned int numberOfEchoTimes = 0;
int tempNum = 0;
// Make sure that the number of images
// stored in the REC file matches the number of echoes.
if( !itk::ExposeMetaData<int>(imageIO->GetMetaDataDictionary(),
```

Table 4: Average T_1 , S_0 , constant B , and R-squared value for each tube in the Bruker T_1 phantom example.

Tube #	Average T_1 (ms)	Average S_0	Average B	Average R-squared
1	825.51 ± 9.00	$1.3027 \times 10^9 \pm 6.3788 \times 10^7$	1.0438 ± 0.0017	0.999 ± 0.000
2	838.20 ± 6.65	$1.5196 \times 10^9 \pm 4.5192 \times 10^7$	1.0487 ± 0.0014	0.998 ± 0.000
3	825.85 ± 15.32	$1.4088 \times 10^9 \pm 6.9022 \times 10^7$	1.0596 ± 0.0024	0.998 ± 0.000
4	1908.40 ± 22.33	$1.4908 \times 10^9 \pm 7.9244 \times 10^7$	1.0538 ± 0.0025	0.996 ± 0.000
5	1186.40 ± 14.60	$1.4531 \times 10^9 \pm 5.9687 \times 10^7$	1.0571 ± 0.0028	0.998 ± 0.000
6	1019.90 ± 9.41	$1.4596 \times 10^9 \pm 6.4035 \times 10^7$	1.0586 ± 0.0025	0.998 ± 0.000
7	838.67 ± 11.49	$1.4719 \times 10^9 \pm 5.0358 \times 10^7$	1.0653 ± 0.0026	0.998 ± 0.000
8	1260.90 ± 15.10	$1.4911 \times 10^9 \pm 6.9226 \times 10^7$	1.0481 ± 0.0023	0.998 ± 0.000
9	999.66 ± 8.06	$1.4871 \times 10^9 \pm 6.2972 \times 10^7$	1.0491 ± 0.0021	0.998 ± 0.000
10	734.15 ± 7.62	$1.4678 \times 10^9 \pm 6.6556 \times 10^7$	1.0601 ± 0.0028	0.998 ± 0.000
11	532.28 ± 12.63	$1.6377 \times 10^9 \pm 2.4449 \times 10^8$	1.1015 ± 0.0075	0.999 ± 0.000
12	1275.90 ± 13.15	$1.5154 \times 10^9 \pm 7.2595 \times 10^7$	1.0515 ± 0.0023	0.998 ± 0.000
13	1310.00 ± 10.64	$1.6134 \times 10^9 \pm 6.2195 \times 10^7$	1.0526 ± 0.0021	0.998 ± 0.000
14	1317.20 ± 13.43	$1.6257 \times 10^9 \pm 6.8480 \times 10^7$	1.0567 ± 0.0017	0.998 ± 0.000

```

itk::PAR_MaxNumberOfEchoes,tempNum) )
{
    std::cerr << "Could not determine the number of echoes" << std::endl;
    return 1;
}
numberOfEchoTimes = static_cast<unsigned int>(tempNum);

// Get the image dimensions and make sure that
// there exists at least numberOfEchoTimes image
// volumes. It's possible to have more if the
// REC file contains more than one image type
// (i.e. magnitude, phase, real, imaginary, etc.)
dims[0] = imageIO->GetDimensions(0);
dims[1] = imageIO->GetDimensions(1);
dims[2] = imageIO->GetDimensions(2);
dims[3] = imageIO->GetDimensions(3);
if( numberOfEchoTimes > dims[3] )
{
    std::cerr << "The number of echoes is larger than the number of image blocks" << std::endl;
    return 1;
}

```

As noted in the code comments it's possible that more than one image type is stored in the REC file. For example the image used in this example contains both the magnitude and phase images, so `dims[3] = 28` instead of 14. By default REC files store the slices unsorted. `itk::PhilipsRECImageIO` will automatically sort the slices by image type when the data is read. However, in order to calculate T_2 , the magnitude image is needed. The following code checks to see if the REC file contains the magnitude image using the `itk::MetaDataDictionary`:

```

// Check to see if the REC file has at least the magnitude image.
int haveMagnitude = 0;

```

```

int numberOfImageTypes = 0;
PhilipsRECImageIOType::ImageTypesType imageTypes;
if( !itk::ExposeMetaData<int>(imageIO->GetMetaDataDictionary(),
    itk::PAR_NumberOfImageTypes,numberOfImageTypes) )
{
    std::cerr << "Could not determine the number of image types" << std::endl;
    return 1;
}
if( !itk::ExposeMetaData<PhilipsRECImageIOType::ImageTypesType>
    (imageIO->GetMetaDataDictionary(), itk::PAR_ImageTypes,imageTypes) )
{
    std::cerr << "Could not get the image types vector" << std::endl;
    return 1;
}
for(int j=0; j<numberOfImageTypes; j++)
{
    if( imageTypes[j] == 0 )
    {
        haveMagnitude = 1;
    }
}
if( !haveMagnitude )
{
    std::cerr << "Magnitude image type not found in REC file" << std::endl;
    return 1;
}

```

An issue that must be dealt with when using Philips PAR/REC image data is the existence of multiple scanning sequences. The Philips T_1 example (3.4) makes use of this feature to extract the phase corrected real image for calculating T_1 via the inversion recovery method. However, for T_2 , if more than 1 scanning sequence exists in the REC file, processing is aborted with an error:

```

// Check to make sure that there is only one scanning sequence,
// otherwise the T2 map cannot be processed.
int numberOfScanningSequences = 0;
if( !itk::ExposeMetaData<int>(imageIO->GetMetaDataDictionary(),
    itk::PAR_NumberOfScanningSequences,numberOfScanningSequences) )
{
    std::cerr << "Could not determine the number of scanning sequences";
    std::cerr << std::endl;
    return 1;
}
if( numberOfScanningSequences > 1 )
{
    std::cerr << "Cannot process a T2 map when the number of scanning sequences";
    std::cerr << " is greater than 1" << std::endl;
    return 1;
}

```

Philips REC images are stored as 16 bit signed integers. The PAR file lists rescale intercept (RI), rescale slope (RS), and scale slope (SS) values for converting the integer data back to the original floating point

data. As described in the PAR file, the floating point value is obtained using the following formula:

$$FP = \frac{PV * RS + RI}{RS * SS} \quad (15)$$

where PV is the pixel value in the REC file and FP is the floating point value. The following source demonstrates how to get the scale values using the `itk::MetaDataDictionary` and convert the image to floating point using `itk::ShiftScaleImageFilter` and `itk::ShiftScaleInPlaceImageFilter`:

```
// Get rescale values for converting the 16 bit image to floating point.
PhilipsRECImageIOType::ScanningSequenceImageTypeRescaleValuesContainerType::Pointer
scanSequenceImageTypeRescaleValues = NULL;
if( !itk::ExposeMetaData
    <PhilipsRECImageIOType::ScanningSequenceImageTypeRescaleValuesContainerType::Pointer>
    (imageIO->GetMetaDataDictionary(),itk::PAR_ScanningSequenceImageTypeRescaleValues,
    scanSequenceImageTypeRescaleValues) )
{
    std::cerr << "Could not get the rescale values for each";
    std::cerr << " scanning sequence and image type" << std::endl;
    return 1;
}
if( !scanSequenceImageTypeRescaleValues )
{
    std::cerr << "Received NULL scanning sequence/image types vector";
    std::cerr << " pointer from meta dictionary" << std::endl;
    return 1;
}
PhilipsRECImageIOType::ImageTypeRescaleValuesContainerType::Pointer
rescaleValueVector
= scanSequenceImageTypeRescaleValues->ElementAt(0); // Only 1 scanning sequence.
if( !rescaleValueVector )
{
    std::cerr << "Received NULL rescale values vector pointer from";
    std::cerr << " meta dictionary" << std::endl;
    return 1;
}

// Change image to floating point value.
ShiftScaleImageFilterType::Pointer scaleOnly = NULL;
ShiftScaleInPlaceImageFilterType::Pointer shiftAndScale = NULL;
PhilipsRECImageIOType::ImageTypeRescaleValuesType rescaleValues
= rescaleValueVector->ElementAt(0); // Magnitude image will be the first element.
if( (rescaleValues[2] != 0) && // scale slope (SS)
    (rescaleValues[1] != 0) ) // rescale slope (RS)
{
    scaleOnly = ShiftScaleImageFilterType::New();
    scaleOnly->SetInput(t2Mask->GetOutput());
    scaleOnly->SetScale(rescaleValues[1]); // RS
    shiftAndScale = ShiftScaleInPlaceImageFilterType::New();
    shiftAndScale->SetInput(scaleOnly->GetOutput());
    shiftAndScale->SetShift(rescaleValues[0]); // rescale intercept (RI)
    shiftAndScale->SetScale(1.0/(rescaleValues[2]*rescaleValues[1])); // 1/(SS*RS)
}
```

```

else
{
    std::cerr << "Invalid rescale values" << std::endl;
    return 1;
}

```

Because the Philips data is 4D, it is easier in this instance to use `AddMREchoImage()` to add the input images to `itk::MRT2ParameterMap3DImageFilter` instead of `SetMREchoImage()`. This is done using a `itk::VectorContainer` of `itk::ExtractImageFilter` smart pointers, one for each echo image, to extract the images from the 4D volume as follows:

```

ExtractImageFilterContainerType::Pointer extractVOI =
ExtractImageFilterContainerType::New();
extractVOI->resize(numberOfEchoTimes-1);
ExtractImageFilterType::InputImageRegionType extractionRegion;
ExtractImageFilterType::InputImageSizeType extractionSize;
extractionSize[0] = dims[0];
extractionSize[1] = dims[1];
extractionSize[2] = dims[2];
extractionSize[3] = 0;
ExtractImageFilterType::InputImageIndexType extractionIndex;
extractionIndex[0] = 0;
extractionIndex[1] = 0;
extractionIndex[2] = 0;
extractionRegion.SetSize(extractionSize);
PhilipsRECImageIOType::EchoTimesContainerType::Pointer
    ptrToEchoes = NULL;
if( !itk::ExposeMetaData<PhilipsRECImageIOType::EchoTimesContainerType::Pointer>
    (imageIO->GetMetaDataDictionary(), itk::PAR_EchoTimes, ptrToEchoes) )
{
    std::cerr << "Could not get the echo times" << std::endl;
    return 1;
}
if( !ptrToEchoes )
{
    std::cerr << "Received NULL echo times pointer from";
    std::cerr << " meta dictionary" << std::endl;
    return 1;
}
if( ptrToEchoes->size() != numberOfEchoTimes )
{
    std::cerr << "The size of the echo times vector does";
    std::cerr << " not match the number of echoes listed in";
    std::cerr << " the PAR file" << std::endl;
    return 1;
}
for(unsigned int i=1; i<numberOfEchoTimes; i++)
{
    extractVOI->SetElement(i-1, ExtractImageFilterType::New());
    extractVOI->ElementAt(i-1)->SetInput(shiftAndScale->GetOutput());
    extractionIndex[3] = i;
    extractionRegion.SetIndex(extractionIndex);
    extractVOI->ElementAt(i-1)->SetExtractionRegion(extractionRegion);
}

```

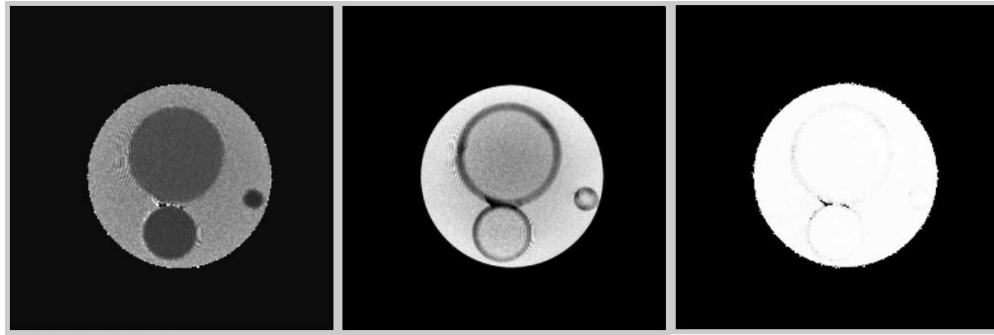


Figure 5: Output from Philips T_2 parameter map example. The image on the left is the T_2 parameter map, the middle image is the constant S_0 , and the right image is the R-squared map. Like Figure 1 significant T_1 contrast exists in the constant image S_0 due to the extremely short TR of 252.901 ms.

```
t2Map->AddMREchoImage(ptrToEchoes->ElementAt(i)/1000.0f, // convert to seconds
    extractVOI->ElementAt(i-1)->GetOutput());
}
```

Like the Bruker T_2 example (3.1), the first echo image is not used and the echo times are converted to seconds. Figure 5 shows the output images from this example. Figure 6 plots the fitted data against the experimentally acquired data points for a location at the center of a gray matter region in the phantom. As seen in the figure, the nonlinear fitting routine found a reasonable fit in spite of the noisy input data.

3.4 Philips REC T_1 Parameter Map

The final example demonstrates an inversion recovery T_1 parameter map measurement using a fast inversion recovery Look-Locker sequence acquired on a 3.0 T Philips Achieva system. The Philips T_1 phantom specified at the beginning of this section was used (3). The name of the file is T1_LL_10_1.PAR and the example code is saved in PhilipsT1Map.cxx. A single 2 mm axial slice with SENSE factor 2.5, 160 x 160 matrix size, 225 mm x 225 mm FOV, TR of 25.0 ms, TE of 4.45 ms, and 8 inversion times (TI 83, 532, 980, 1429, 1877, 2325, 2774, and 3222 ms) was acquired.

PhilipsT1Map.cxx takes the same 9 parameters as BrukerT1Map.cxx (3.2). For this example the optional parameters are 0 (output T_1), 1 (use INVERSION_RECOVERY algorithm Equation 9), 5.0 (maximum T_1 time of 5.0 seconds), and 50 (every voxel less than 50 set to zero). The IDEAL_STEADY_STATE, Equation 7, and HYBRID_STEADY_STATE_3PARAM, Equation 8, algorithms are not supported.

Similar to the Philips T_2 example (3.3), PhilipsT1Map.cxx checks to make sure that the number of image volumes is greater than or equal to the number of time points. However, the number of time points in this case is equal to the number of cardiac phases, instead of the number of echo times:

```
unsigned int numberOfTimePoints = 0;
int tempNum = 0;
// Find the maximum number of cardiac phases.
if( !itk::ExposeMetaData<int>(imageIO->GetMetaDataDictionary(),
    itk::PAR_MaxNumberOfCardiacPhases,tempNum) )
{
    std::cerr << "Could not determine the number of cardiac";
    std::cerr << " phases" << std::endl;
}
```

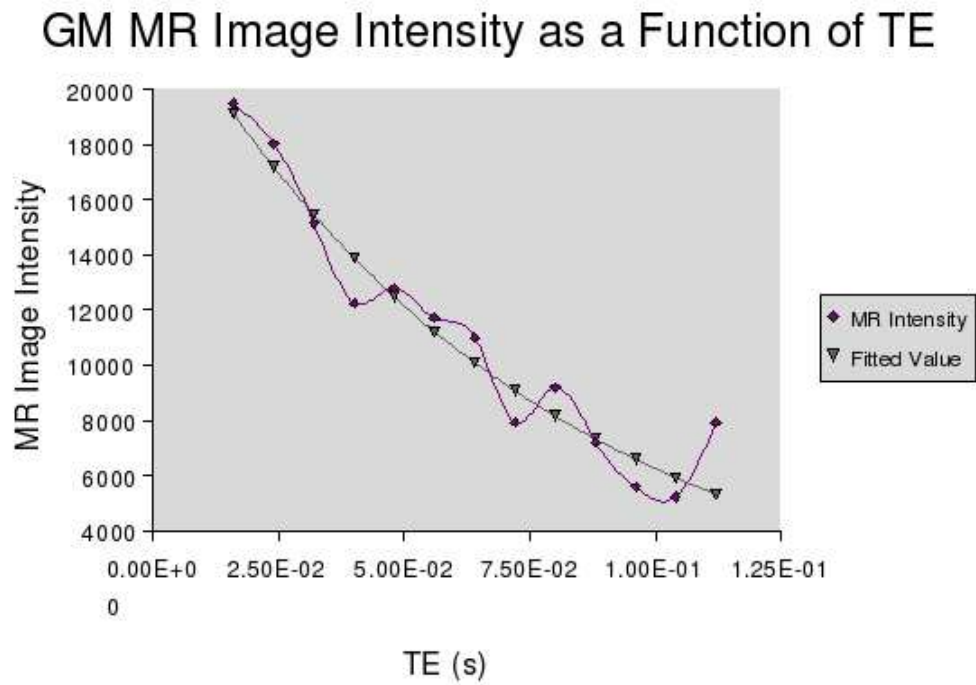


Figure 6: A plot of MR image intensity (S_i) as a function of TE for a pixel in the center of the gray matter sphere used in the Philips T_2 parameter map example with overlay of fitted curve. The experimental image intensity points are noisy due to the short TR 252.901 ms and sense factor 2.5. A reasonable fit was obtained in spite of the expected variation in the experimental data points.

```

    return 1;
}
numberOfTimePoints = static_cast<unsigned int>(tempNum);
if( numberOfTimePoints < 2 )
{
    std::cerr << "Must have at least 2 cardiac phase images to";
    std::cerr << " calculate T1" << std::endl;
    return 1;
}

// Get the image dimensions and make sure that
// there exists at least numberOfTimePoints image
// volumes. It's possible to have more if the
// REC file contains more than one image type
// (i.e. magnitude, phase, real, imaginary, etc.)
dims[0] = imageIO->GetDimensions(0);
dims[1] = imageIO->GetDimensions(1);
dims[2] = imageIO->GetDimensions(2);
dims[3] = imageIO->GetDimensions(3);
if( numberOfTimePoints > dims[3] )
{
    std::cerr << "The number of time points is larger than the number";
    std::cerr << " of image blocks" << std::endl;
    return 1;
}

```

Unlike PhilipsT2Map.cxx, PhilipsT1Map.cxx requires either the magnitude image or both the magnitude image and the phase corrected real image depending on the fit type. The magnitude image is used to generate a binary mask using the optional threshold value supplied on the command line. The phase corrected real image is listed as image type 4 in the PAR file and is required for the non-absolute value inversion recovery fitting options. The code below will check for the existence of the proper image types stored in the [itk::MetaDataDictionary](#) and return with an error if the required image type does not exist in the REC file:

```

// Check to see if we have the correct image types.
int haveCorrectImage = 0;
int numberOfImageTypes = 0;
PhilipsRECImageIOType::ImageTypesType imageTypes;
if( !itk::ExposeMetaData<int>(imageIO->GetMetaDataDictionary(),
    itk::PAR_NumberOfImageTypes,numberOfImageTypes) )
{
    std::cerr << "Could not determine the number of image types" << std::endl;
    return 1;
}
if( !itk::ExposeMetaData<PhilipsRECImageIOType::ImageTypesType>
    (imageIO->GetMetaDataDictionary(),itk::PAR_ImageTypes,imageTypes) )
{
    std::cerr << "Could not get the image types vector" << std::endl;
    return 1;
}
// Need image 4 (corrected real image) for inversion recovery/Look-Locker
// fit type.
if( (algorithm == MRT1ParameterMap3DImageFilterType::INVERSION_RECOVERY) ||

```

```

    (algorithm == MRT1ParameterMap3DImageFilterType::INVERSION_RECOVERY_3PARAM) ||
    (algorithm == MRT1ParameterMap3DImageFilterType::LOOK_LOCKER) )
{
    for(int j=0; j<numberOfImageTypes; j++)
    {
        if( imageTypes[j] == 4 )
        {
            haveCorrectImage = 1;
        }
    }
    if( !haveCorrectImage )
    {
        std::cerr << "No corrected real image type detected in PAR";
        std::cerr << " file" << std::endl;
        return 1;
    }
}
// Now check for existence of the magnitude image.
haveCorrectImage = 0;
for(int j=0; j<numberOfImageTypes; j++)
{
    if( imageTypes[j] == 0 )
    {
        haveCorrectImage = 1;
    }
}
if( !haveCorrectImage )
{
    std::cerr << "No magnitude image type detected in PAR file" << std::endl;
    return 1;
}

```

If the correct image types exist in the REC file, as reported in the PAR file, the program will attempt to read the inversion times from the [itk::MetaDataDictionary](#). The actual inversion times are listed as trigger times in the PAR file and are extracted as follows:

```

// Get trigger times.
PhilipsRECImageIOType::TriggerTimesContainerType::Pointer
ptrToTimePoints = NULL;
if( !itk::ExposeMetaData<PhilipsRECImageIOType::TriggerTimesContainerType::Pointer>
    (imageIO->GetMetaDataDictionary(),itk::PAR_TriggerTimes,ptrToTimePoints) )
{
    std::cerr << "Could not get the trigger times" << std::endl;
    return 1;
}
if( !ptrToTimePoints )
{
    std::cerr << "Received NULL trigger times pointer from meta";
    std::cerr << " dictionary" << std::endl;
    return 1;
}
if( ptrToTimePoints->size() != numberOfTimePoints )
{

```

```

std::cerr << "The size of the time points vector does not match the";
std::cerr << " number of cardiac phases listed in the PAR file" << std::endl;
return 1;
}

```

After scaling the 16 bit data read using `itk::PhilipsRECImageIO` to floating point data as detailed in the previous example (3.3), each 3D volume for each time point is extracted, masked, and assigned as inputs to `itk::MRT1ParameterMap3DImageFilter` using `AddMRImage()`:

```

// Extract volumes according to algorithm type.
if( (algorithm == MRT1ParameterMap3DImageFilterType::INVERSION_RECOVERY) ||
    (algorithm == MRT1ParameterMap3DImageFilterType::INVERSION_RECOVERY_3PARAM) ||
    (algorithm == MRT1ParameterMap3DImageFilterType::LOOK_LOCKER) )
{
    // Corrected real image is after the magnitude and real images in first scanning sequence.
    for(unsigned int i=0; i<numberOfTimePoints; i++)
    {
        extractVOI->SetElement(i, ExtractImageFilterType::New());
        extractVOI->ElementAt(i)->SetInput(shiftAndScale->GetOutput());
        extractionIndex[3] = 2*numberOfTimePoints + i;
        extractionRegion.SetIndex(extractionIndex);
        extractVOI->ElementAt(i)->SetExtractionRegion(extractionRegion);
        maskFilterContainer->SetElement(i, MaskImageFilterType::New());
        maskFilterContainer->ElementAt(i)->SetInput1(extractVOI->ElementAt(i)->GetOutput());
        maskFilterContainer->ElementAt(i)->SetInput2(magnitudeMask->GetOutput());
        t1Map->AddMRImage(ptrToTimePoints->ElementAt(i)/1000.0f, // convert to seconds
            maskFilterContainer->ElementAt(i)->GetOutput());
    }
}
else
{
    // Magnitude image is at front.
    for(unsigned int i=0; i<numberOfTimePoints; i++)
    {
        extractVOI->SetElement(i, ExtractImageFilterType::New());
        extractVOI->ElementAt(i)->SetInput(shiftAndScale->GetOutput());
        maskFilterContainer->SetElement(i, MaskImageFilterType::New());
        maskFilterContainer->ElementAt(i)->SetInput1(extractVOI->ElementAt(i)->GetOutput());
        maskFilterContainer->ElementAt(i)->SetInput2(magnitudeMask->GetOutput());
        extractionIndex[3] = i;
        extractionRegion.SetIndex(extractionIndex);
        extractVOI->ElementAt(i)->SetExtractionRegion(extractionRegion);
        t1Map->AddMRImage(ptrToTimePoints->ElementAt(i)/1000.0f, // convert to seconds
            maskFilterContainer->ElementAt(i)->GetOutput());
    }
}
}

```

The final output is then extracted and written to disk as shown previously (3.1). Figure 7 shows the T_1 , exponent constant, and R-squared output images for this example and Table 5 contains the average T_1 time, exponent constant, and R-squared value for each tube in the phantom. For verification the magnitude of the MR signal at the center of tube 5 (pure water) was plotted for each image time point as a function of inversion time and the fitted function was added as an overlay in Figure 8. Similar to the previous examples, the predicted inversion recovery curve fits the experimental data well.

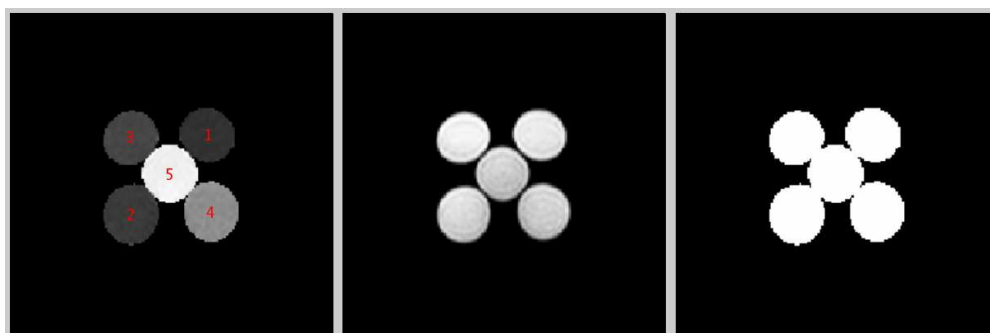


Figure 7: Output from Philips T_1 parameter map example. The image on the left is the T_2 parameter map, the middle image is the constant S_0 , and the right image is the R-squared map. The tube numbers from Table 2 are labeled in red on the T_1 parameter map image.

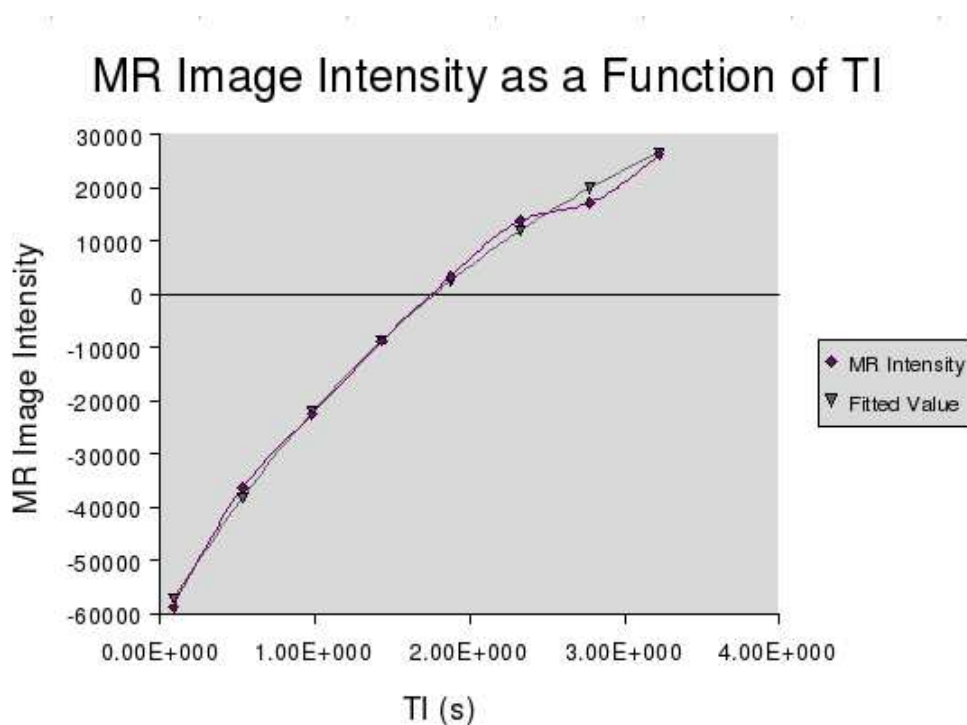


Figure 8: A plot of MR image intensity as a function of TI for a pixel in the center of tube 5 with overlay of fitted curve. Like the previous examples, this demonstrates that the predicted data is a good fit for the experimental data.

Table 5: Average T_1 , S_0 , and R-squared value for each tube in the Philips T_1 phantom example.

Tube #	Average T_1 (ms)	Average S_0	Average R-squared
1	895.48 ± 17.72	68353 ± 3511	0.998 ± 0.001
2	976.82 ± 23.56	62715 ± 4718	0.998 ± 0.001
3	1074.00 ± 18.94	72041 ± 3235	0.998 ± 0.001
4	1639.40 ± 28.21	60428 ± 3412	0.998 ± 0.001
5	2601.60 ± 59.02	60281 ± 3991	0.998 ± 0.001

4 Conclusion

This paper outlines the background, implementation, and use of a suite of classes and programs for generating MR T_2 and T_1 parameter maps using Philips PAR/REC and Bruker 2dseq image files. The supplied examples based on image phantoms demonstrate that these classes generate valid parameter maps. Included with the MR parameter mapping classes are new image readers for Philips PAR/REC and Bruker 2dseq image files, which are useful not only for MR parameter mapping, but also for integration with DTI, segmentation, and registration pipelines within ITK. Feel free to contact the author for suggestions or bugs (<http://www.hmc.psu.edu/nmrlab/groups/students/dbigler.htm>).

References

- [1] Don C. Bigler, Byeong-Yeul Lee, Helen E. Stephens, Collin M. Torok, J. Ryan Olson, Jianli Wang, Kevin R. Scott, Qing X. Yang, Zachary Simmons, and Michael B. Smith. Multimodality voxel-based mri study - possible tool for disease diagnosis and monitoring in als. In *Joint Annual Meeting ISMRM-ESMRMB*, Berlin, Germany, 2007. 1
- [2] R. Deichmann and A. Haase. Quantification of t1 values by snapshot-flash nmr imaging. *Journal of Magnetic Resonance*, 96(3):608–612, 1992. 1.2
- [3] R. Deichmann, D. Hahn, and A. Haase. Fast t1 mapping on a whole-body scanner. *Magnetic Resonance in Medicine*, 42(1):206–209, 1999. 10.1002/(SICI)1522-2594(199907)42:1;206::AID-MRM28;3.0.CO;2-Q. 1.2
- [4] Ralf Deichmann. Fast high-resolution t1 mapping of the human brain. *Magnetic Resonance in Medicine*, 54(1):20–27, 2005. 1.2
- [5] P. A. Gowland and M. O. Leach. Fast and accurate measurements of t-1 using a multi-readout single inversion-recovery sequence. *Magnetic Resonance in Medicine*, 26(1):79–88, 1992. 1.2
- [6] Elizabeth Henderson, Graeme McKinnon, Ting-Yim Lee, and Brian K. Rutt. A fast 3d look-locker method for volumetric t1 mapping. *Magnetic Resonance Imaging*, 17(8):1163–1171, 1999. 1.2
- [7] Alistair M. Howseman, David L. Thomas, Gaby S. Pell, Stephen R. Williams, and Roger J. Ordidge. Rapid t-1 mapping using interleaved echo planar imaging. *Magnetic Resonance in Medicine*, 41(2):368–374, 1999. 1.2
- [8] Jung-Jiin Hsu and Gary H. Glover. Rapid mri method for mapping the longitudinal relaxation time. *Journal of Magnetic Resonance*, 181(1):98–106, 2006. 1.2

- [9] Peter B. Kingsley. Methods of measuring spin-lattice (t1) relaxation times: An annotated bibliography. *Concepts in Magnetic Resonance*, 11(4):243–276, 1999. 10.1002/(SICI)1099-0534(1999)11:4;243::AID-CMR5;3.0.CO;2-C. [1.2](#)
- [10] D. C. Look and D. R. Locker. Time saving in measurement of nmr and epr relaxation times. *Review of Scientific Instruments*, 41(2):250–251, 1970. [1.2](#)
- [11] Mamoru Niitsu, Norbert G. Campeau, Stephen J. Riederer, Izumi Anno, and Yuji Itai. Rapid t1 estimation using tagged magnetization-prepared gradient-echo mr imaging. *Magnetic Resonance in Medicine*, 26(2):377–385, 1992. [1.2](#)
- [12] Gaby S. Pell, Regula S. Briellmann, Heath Pardoe, David F. Abbott, and Graeme D. Jackson. Composite voxel-based analysis of volume and t2 relaxometry in temporal lobe epilepsy. *NeuroImage*, 39(3):1151–1161, 2008. [1](#)
- [13] Gaby S. Pell, Regula S. Briellmann, Anthony B. Waites, David F. Abbott, and Graeme D. Jackson. Voxel-based relaxometry: a new approach for analysis of t2 relaxometry changes in epilepsy. *NeuroImage*, 21(2):707–713, 2004. [1](#)
- [14] N. J. Shah, M. Zaitsev, S. Steinhoff, and K. Zilles. A new method for fast multislice t1 mapping. *NeuroImage*, 14(5):1175–1185, 2001. TY - JOUR. [1.2](#)
- [15] Karsten Specht, Martina Minnerop, Jonas Mller-Hbenthal, and Thomas Klockgether. Voxel-based analysis of multiple-system atrophy of cerebellar type: complementary results by combining voxel-based morphometry and voxel-based relaxometry. *NeuroImage*, 25(1):287–293, 2005. [1](#)
- [16] S. Steinhoff, M. Zaitsev, K. Zilles, and N. J. Shah. Fast t1 mapping with volume coverage. *Magnetic Resonance in Medicine*, 46(1):131–140, 2001. 10.1002/mrm.1168. [1.2](#)
- [17] Chun Y. Tong and Frank S. Prato. A novel fast t1-mapping method. *Journal of Magnetic Resonance Imaging*, 4(5):701–708, 1994. 10.1002/jmri.1880040513. [1.2](#)
- [18] David C. Zhu and Richard D. Penn. Full-brain t1 mapping through inversion recovery fast spin echo imaging with time-efficient slice ordering. *Magnetic Resonance in Medicine*, 54(3):725–731, 2005. 10.1002/mrm.20602. [1.2](#)
- [19] Chen Zuoqun, F. S. Prato, and C. McKenzie. T1 fast acquisition relaxation mapping (t1-farm): an optimized reconstruction. *IEEE Transactions on Medical Imaging*, 17(2):155–160, 1998. 1558-254X. [1.2](#)