SimpleITK in R

Richard Beare

May 11, 2012

Abstract

R, also known as "Gnu S" is a widely used, open source, language based environment for statistics and computational modelling. It will be reasonably familiar to users of other interactive, interpreted environments, like Matlab or python. This article provides an introduction to the SimpleITK package that has been built using the Swig generated wrapping of the SimpleITK library.

Contents

1 Introduction **1**

2 Very basic *R* tutorial **2**

3 Getting started with SimpleITK **4**
- 3.1 Image anatomy and access methods . . . 4
- 3.2 Image operations with Simple ITK classes . . . 14
- 3.3 Still to come . . . 17
- 3.4 Caveats . . . 18

4 Building and Installing **18**

5 Development **18**

1 Introduction

R is an advanced language environment that supports extension via an advanced package mechanism and object-oriented and generic programming mechanisms. The traditional application domain of *R* is in interactive statistical analysis, but the language is general purpose and facilities are available to support many forms of computational work. There are already a number of packages for medical imaging and general purpose imaging, but none with the extent of low level operators provided by SimpleITK. *R* has quite nice features that makes interfacing to objects like images quite convenient. This package makes extensive use of *external references* and language operator overloading facilities.

2 Very basic *R* tutorial

R has extensive online documentation - see the Documentation links on the r-project pages. Here are some basic concepts to start the project. Skip to the next section if you are already familiar with *R*.

- Assignment - traditionally the assignment operator is `<-`, but `=` can be used in most places now:

  ```
  > a <- 1  # assign a variable
  > b = a
  ```

- Creating vectors - everything in *R* is at least a vector, and vectors can contain numbers or strings:

  ```
  > a <- c(1,2,34, 20, 10)
  > a

  [1]  1  2 34 20 10

  > b = c('a', 'k', 'hello')
  > b

  [1] "a"     "k"     "hello"

  > d <- 1:10
  > d

   [1]  1  2  3  4  5  6  7  8  9 10
  ```

  c is the concatenate operator and can be used with vectors and lists.

- Displaying objects - as seen above, typing a variable name invokes the generic *show* method, which typically provides an informative display of an obje. We'll see how this comes in handy later with images.

- Creating arrays

  ```
  > b<-array(1:20, dim=c(5,4))
  > b

       [,1] [,2] [,3] [,4]
  [1,]    1    6   11   16
  [2,]    2    7   12   17
  [3,]    3    8   13   18
  [4,]    4    9   14   19
  [5,]    5   10   15   20
  ```

- Vector and array subsetting - there are a rich set of these operations with capabilities similar to Matlab. Indexing starts from 1.

```
> a[1:2]

[1] 1 2

> a[3:1]

[1] 34  2  1

> a[-1]  # delete first element

[1]  2 34 20 10

> b[1,]  # first row of b

[1]  1  6 11 16

> b[1,c(1,4,2)]

[1]  1 16  6
```

- Lists - can contain different object types

```
> L1 = list('a', 1, 'hello')
> L1

[[1]]
[1] "a"

[[2]]
[1] 1

[[3]]
[1] "hello"

> is.list(L1)

[1] TRUE

> L1[[2]]

[1] 1
```

  Notice that we are using the double bracket operator to access list elements.

- Naming components - so far we have been illustrating standard, index-based, access. It is possible to name array, vector and list components which provides options for clear accessing.

```
> L1 <- list(first=1, second='hello', third=b)
> L1$second

[1] "hello"
```

```
> L1[["first"]]
```

```
[1] 1
```

```
> colnames(b) <- c("first", "second", "third", "last")
> b[,"last"]
```

```
[1] 16 17 18 19 20
```

These options provide useful ways of keeping consistency in complex analyses with evolving data structures.

- Other data structures. The main structure not discussed here is a special list, called a data frame, that is widely used by the statistical model-fitting procedures. Classes, methods and other language facilities are also available, but used mainly by package developers.

3 Getting started with SimpleITK

Building and installation instructions are later. Lets jump straight into some examples. In order to display images you need to install ImageJ with the nifti plugin, and be in your path. The results in this docment are displayed slightly differently, using internal *R* plotting routines, for compatability with the Sweave document processing.

3.1 Image anatomy and access methods

- Load the SimpleITK library. This may require that the `R_LIBS` environment variable is set.

```
> library(SimpleITK)
```

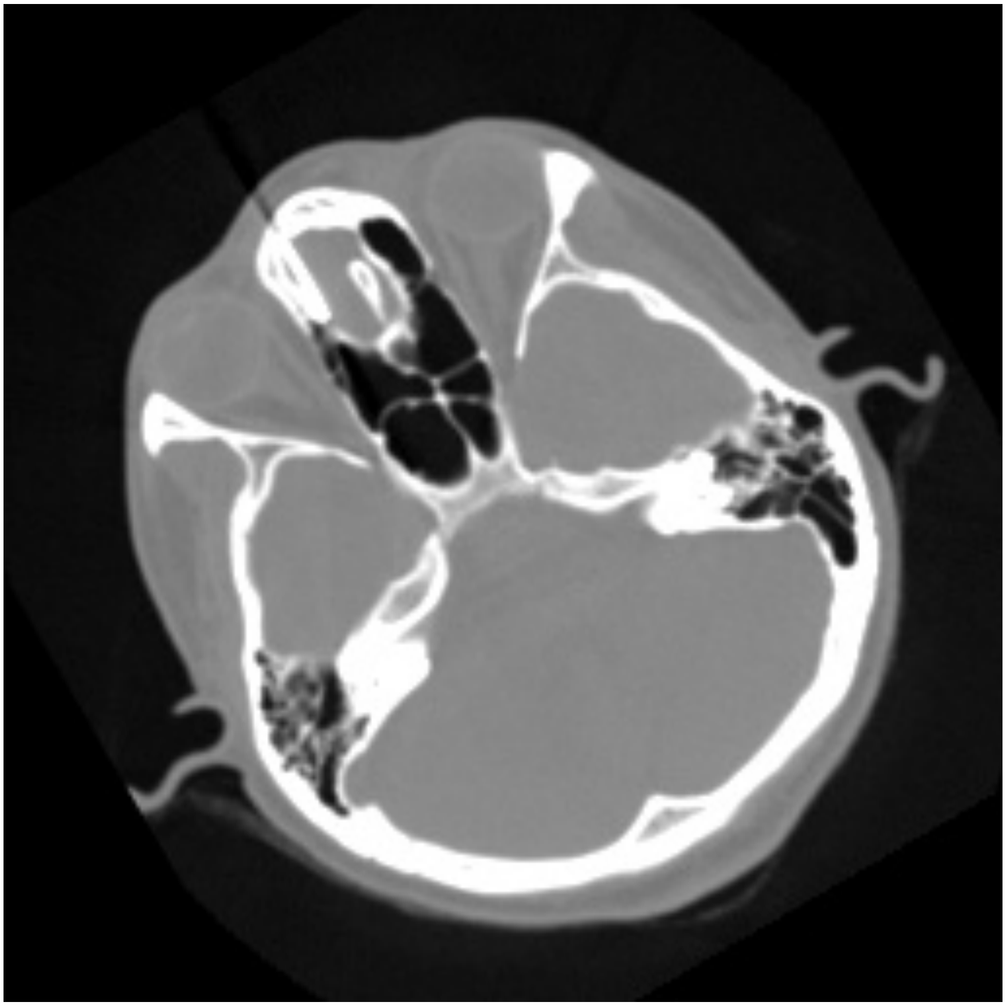
```
[1] "show"
```

- Load an image

```
> im <- ReadImage('/home/richardb/Projects/ITK4/SimpleITK/Testing/Data/Input/cthead1.pn
```

- Display

```
> im
```



- Get some information about the image

  > print(im)

```
Image (0x355cb00)
  RTTI typeinfo:   itk::Image<unsigned char, 2u>
  Reference Count: 1
  Modified Time: 305
  Debug: Off
  Observers:
    none
  Source: (none)
  Source output name: (none)
  Release Data: Off
  Data Released: False
  Global Release Data: Off
  PipelineMTime: 172
  UpdateMTime: 304
  RealTimeStamp: 0 seconds
  LargestPossibleRegion:
```

```
      Dimension: 2
      Index: [0, 0]
      Size: [256, 256]
    BufferedRegion:
      Dimension: 2
      Index: [0, 0]
      Size: [256, 256]
    RequestedRegion:
      Dimension: 2
      Index: [0, 0]
      Size: [256, 256]
    Spacing: [0.352778, 0.352778]
    Origin: [0, 0]
    Direction:
  1 0
  0 1

    IndexToPointMatrix:
    0.352778 0
  0 0.352778

    PointToIndexMatrix:
    2.83465 0
  0 2.83465

    Inverse Direction:
    1 0
  0 1

    PixelContainer:
      ImportImageContainer (0x1f0e340)
        RTTI typeinfo:   itk::ImportImageContainer<unsigned long, unsigned char>
        Reference Count: 1
        Modified Time: 301
        Debug: Off
        Observers:
          none
        Pointer: 0x4fda070
        Container manages memory: true
        Size: 65536
        Capacity: 65536
```

```
> im$GetSpacing()

[1] 0.3527778 0.3527778

> im$GetSize()

[1] 256 256
```

These vector quantities are translated directly to R vectors. The same applies to filters, as we'll see later.

- Get one pixel value

```
> im[100, 120]
```

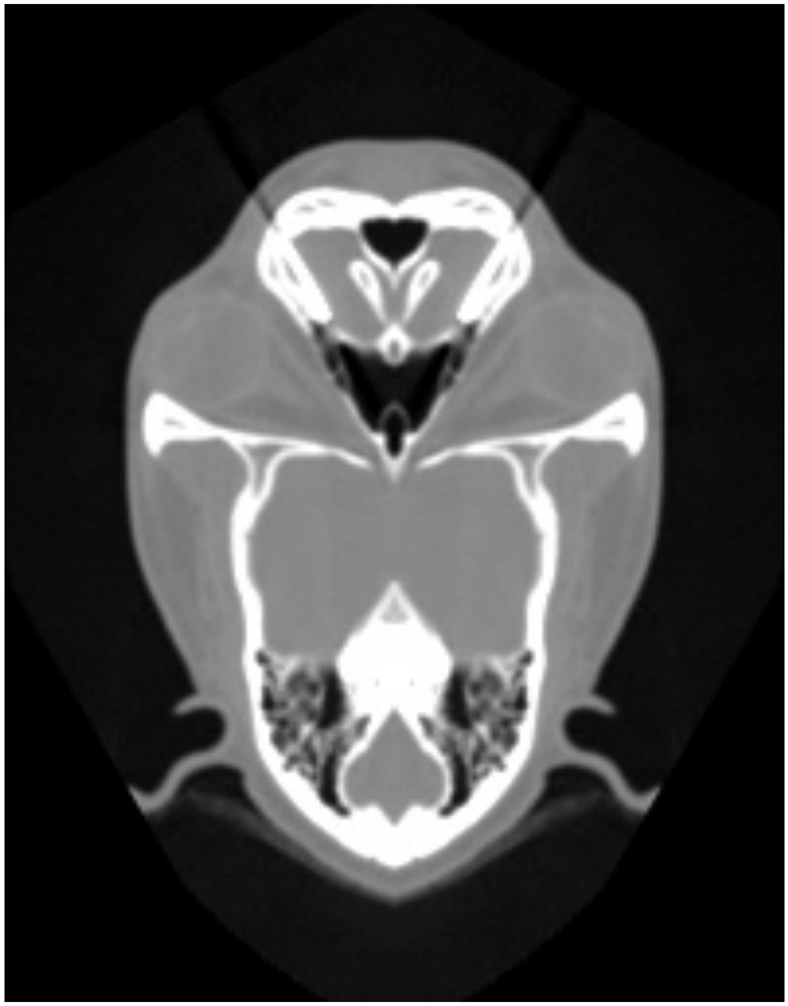
```
[1] 210
```

- Extract the first 100 columns

```
> im[1:100,]
```

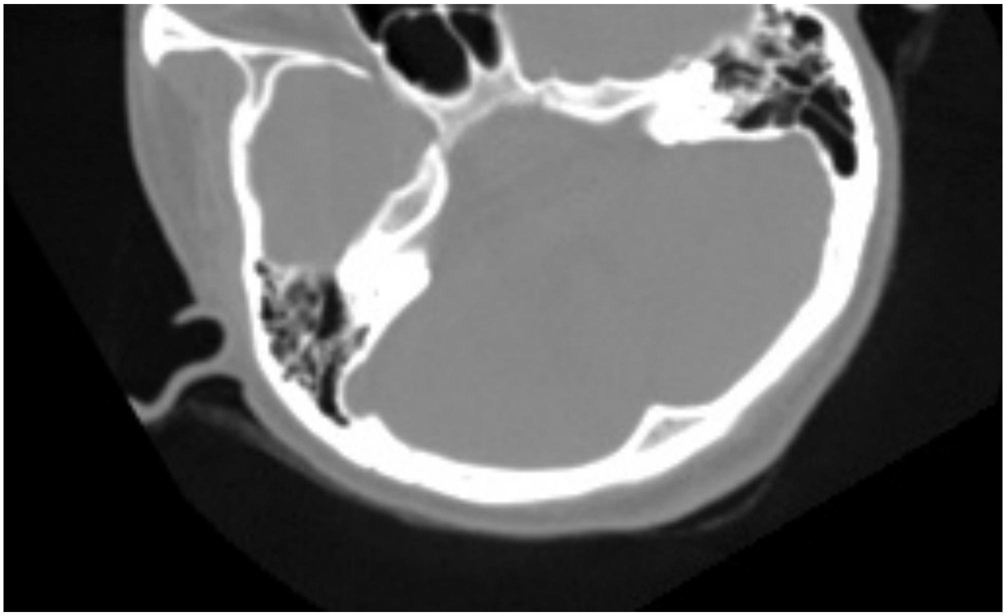
- First 100 columns, then same data flipped

```
> im[c(1:100,100:1),]
```



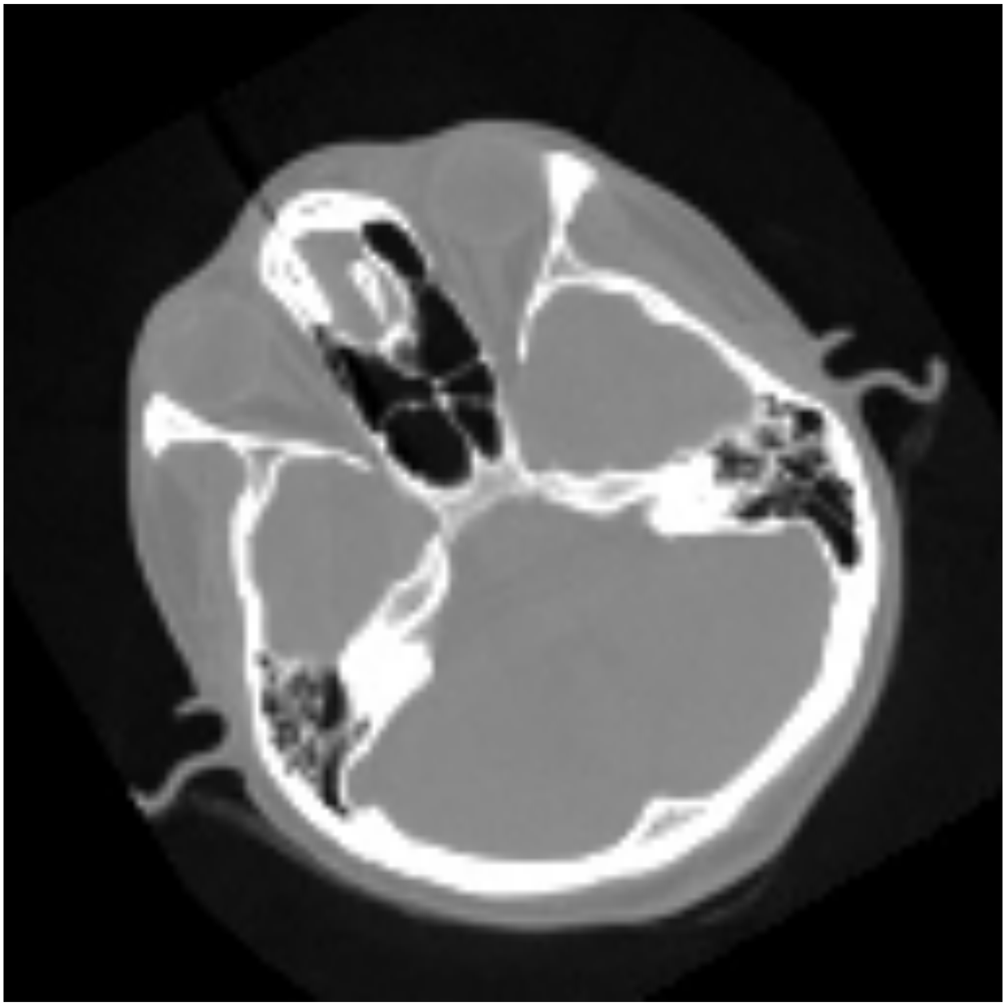
- Remove the first 100 rows

```
> im[,-(1:100)]
```



- Subsample by 2

```
> im[seq(1,256, by=2), seq(1,256, by=2)]
```



As you can see, we can use array acess techniques to images. The results of each of these operations is an image, not an array. Evidently, we can do some pretty crazy things using this notation, which means that it is very difficult to decide what to do with image metadata, such as spacing and origin. Currently nothing clever is being done with either - spacing is left as per the input, which could easily be wrong. Origin is left as default image constructor.

- Lets explore the image class in a little more detail to find out what access methods are available:

> getMethod('$', class(im))

```
Method Definition:

function (x, name)
{
    accessorFuns = list(`__seteq__` = Image___seteq__, GetITKBase = Image_GetITKBase,
        GetPixelIDValue = Image_GetPixelIDValue, GetDimension = Image_GetDimension,
        GetNumberOfComponentsPerPixel = Image_GetNumberOfComponentsPerPixel,
        GetOrigin = Image_GetOrigin, SetOrigin = Image_SetOrigin,
        GetSpacing = Image_GetSpacing, SetSpacing = Image_SetSpacing,
        GetDirection = Image_GetDirection, SetDirection = Image_SetDirection,
```

```
        TransformIndexToPhysicalPoint = Image_TransformIndexToPhysicalPoint,
        TransformPhysicalPointToIndex = Image_TransformPhysicalPointToIndex,
        GetSize = Image_GetSize, GetHeight = Image_GetHeight,
        GetWidth = Image_GetWidth, GetDepth = Image_GetDepth,
        GetPixelIDTypeAsString = Image_GetPixelIDTypeAsString,
        ToString = Image_ToString, GetPixelAsInt8 = Image_GetPixelAsInt8,
        GetPixelAsUInt8 = Image_GetPixelAsUInt8, GetPixelAsInt16 = Image_GetPixelAsInt
        GetPixelAsUInt16 = Image_GetPixelAsUInt16, GetPixelAsInt32 = Image_GetPixelAsIn
        GetPixelAsUInt32 = Image_GetPixelAsUInt32, GetPixelAsInt64 = Image_GetPixelAsIn
        GetPixelAsUInt64 = Image_GetPixelAsUInt64, GetPixelAsFloat = Image_GetPixelAsFl
        GetPixelAsDouble = Image_GetPixelAsDouble, SetPixelAsInt8 = Image_SetPixelAsInt
        SetPixelAsUInt8 = Image_SetPixelAsUInt8, SetPixelAsInt16 = Image_SetPixelAsInt
        SetPixelAsUInt16 = Image_SetPixelAsUInt16, SetPixelAsInt32 = Image_SetPixelAsIn
        SetPixelAsUInt32 = Image_SetPixelAsUInt32, SetPixelAsInt64 = Image_SetPixelAsIn
        SetPixelAsUInt64 = Image_SetPixelAsUInt64, SetPixelAsFloat = Image_SetPixelAsFl
        SetPixelAsDouble = Image_SetPixelAsDouble, GetBufferAsInt8 = Image_GetBufferAsI
        GetBufferAsUInt8 = Image_GetBufferAsUInt8, GetBufferAsInt16 = Image_GetBufferAs
        GetBufferAsUInt16 = Image_GetBufferAsUInt16, GetBufferAsInt32 = Image_GetBuffer
        GetBufferAsUInt32 = Image_GetBufferAsUInt32, GetBufferAsInt64 = Image_GetBuffer
        GetBufferAsUInt64 = Image_GetBufferAsUInt64, GetBufferAsFloat = Image_GetBuffer
        GetBufferAsDouble = Image_GetBufferAsDouble)
    idx = pmatch(name, names(accessorFuns))
    if (is.na(idx))
        return(callNextMethod(x, name))
    f = accessorFuns[[idx]]
    function(...) {
        f(x, ...)
    }
}
<environment: namespace:SimpleITK>

Signatures:
        x
target  "_p_itk__simple__Image"
defined "_p_itk__simple__Image"
```

This provides a list of accessor functions that can be used via the $ notation illustrated above. Most classes create by the swig processing work this way.

- Finally, lets allocate an image

```
> im2 <- Image(10,10, 20, 'sitkUInt16')
> print(im2)

Image (0x3cac3f0)
  RTTI typeinfo:   itk::Image<unsigned short, 3u>
  Reference Count: 1
  Modified Time: 433
  Debug: Off
```

```
    Observers:
      none
    Source: (none)
    Source output name: (none)
    Release Data: Off
    Data Released: False
    Global Release Data: Off
    PipelineMTime: 0
    UpdateMTime: 0
    RealTimeStamp: 0 seconds
    LargestPossibleRegion:
      Dimension: 3
      Index: [0, 0, 0]
      Size: [10, 10, 20]
    BufferedRegion:
      Dimension: 3
      Index: [0, 0, 0]
      Size: [10, 10, 20]
    RequestedRegion:
      Dimension: 3
      Index: [0, 0, 0]
      Size: [10, 10, 20]
    Spacing: [1, 1, 1]
    Origin: [0, 0, 0]
    Direction:
  1 0 0
  0 1 0
  0 0 1

    IndexToPointMatrix:
    1 0 0
  0 1 0
  0 0 1

    PointToIndexMatrix:
    1 0 0
  0 1 0
  0 0 1

    Inverse Direction:
    1 0 0
  0 1 0
  0 0 1

    PixelContainer:
      ImportImageContainer (0x4a53b60)
        RTTI typeinfo:   itk::ImportImageContainer<unsigned long, unsigned short>
        Reference Count: 1
```

```
Modified Time: 434
Debug: Off
Observers:
  none
Pointer: 0x4ff1660
Container manages memory: true
Size: 2000
Capacity: 2000
```

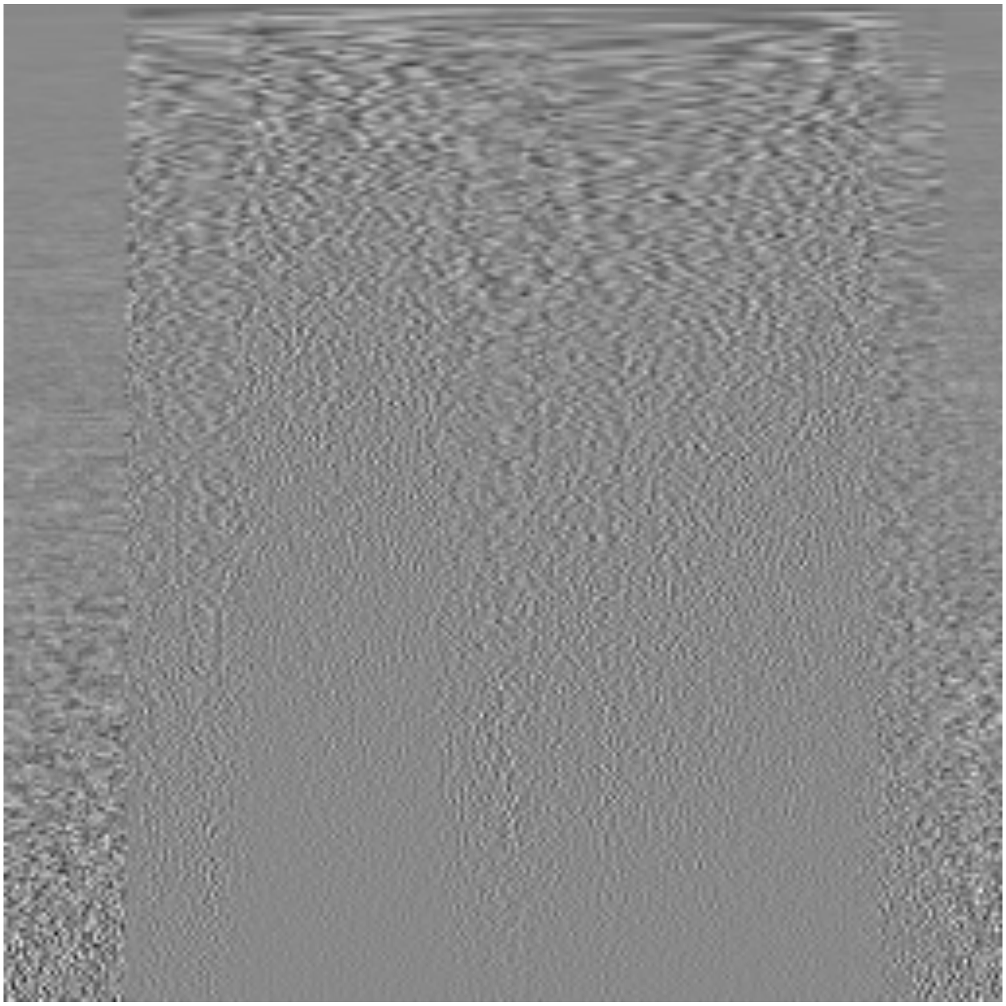
The important points to note here is that the enumerated type describing the pixel type is represented as a string.

- Translating images to R arrays:

```
> arr <- as.array(im)
> class(im)

[1] "_p_itk__simple__Image"

> class(arr)

[1] "matrix"

> # now we can do something crazy
> s <- svd(arr)
```

- And back again

```
> nim <- as.image(s$u)
> nim
```



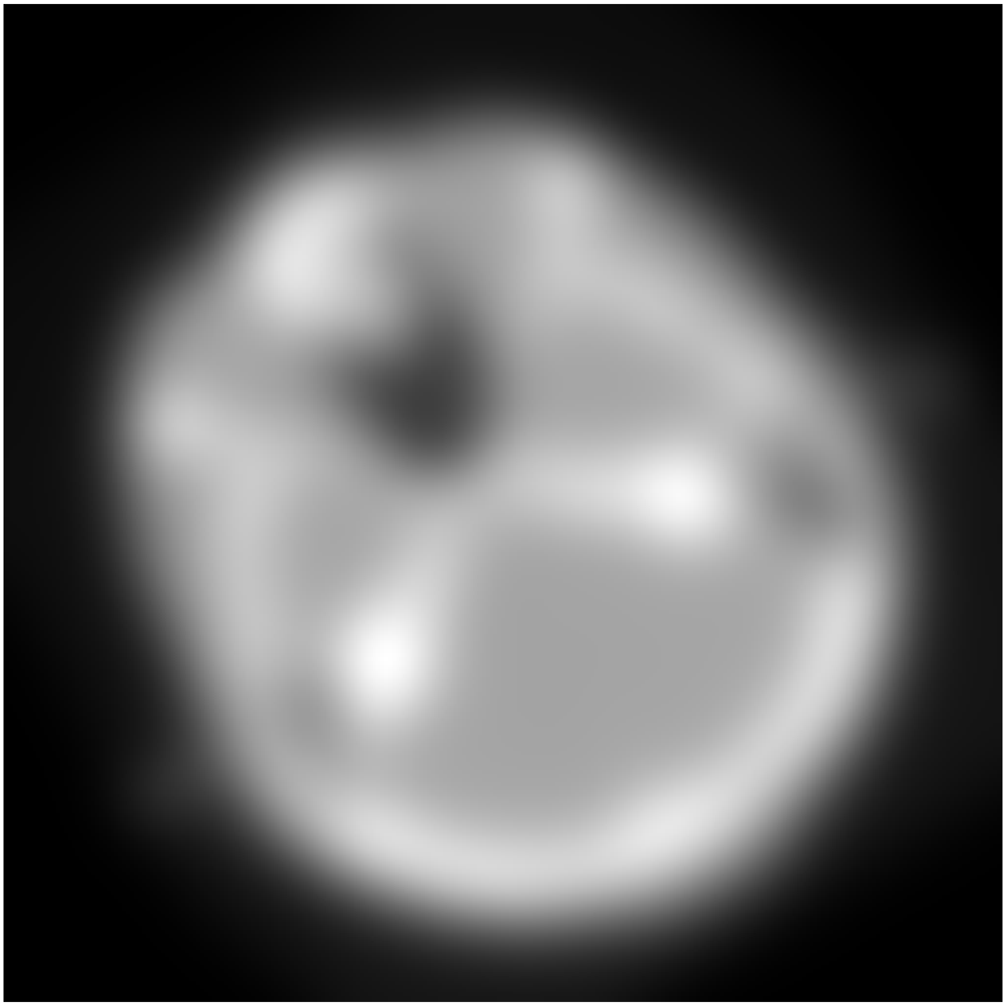
Points to note here - *R* only supports integer and double types (logical types are also integers). Therefore conversion of any image will end up as an array of one of these types. Similarly, conversion of arrays to images also ends up as one of these types. By default, any *R* matrix will be double precision, but can be coerced to integer using the *as.integer* or *storage.mode* functions. Image pixel types can be converted using the *Cast* filters.

3.2 Image operations with Simple ITK classes

Finally, onto the crux of the matter. Let's look at doing some image filtering. There are two basic approaches with SimpleITK - the procedural and the filter approach

- Gaussian blurring:

```
> res <- SmoothingRecursiveGaussian(im, 3)
> res
```



- or

```
> filt <- SmoothingRecursiveGaussianImageFilter()
> # check the accessors
> getMethod('$', class(filt))

Method Definition:

function (x, name)
{
    accessorFuns = list(SetSigma = SmoothingRecursiveGaussianImageFilter_SetSigma,
        GetSigma = SmoothingRecursiveGaussianImageFilter_GetSigma,
        SetNormalizeAcrossScale = SmoothingRecursiveGaussianImageFilter_SetNormalizeAc
        NormalizeAcrossScaleOn = SmoothingRecursiveGaussianImageFilter_NormalizeAcross
        NormalizeAcrossScaleOff = SmoothingRecursiveGaussianImageFilter_NormalizeAcros
        GetNormalizeAcrossScale = SmoothingRecursiveGaussianImageFilter_GetNormalizeAc
        GetName = SmoothingRecursiveGaussianImageFilter_GetName,
        ToString = SmoothingRecursiveGaussianImageFilter_ToString,
        Execute = SmoothingRecursiveGaussianImageFilter_Execute)
    idx = pmatch(name, names(accessorFuns))
```

```
    if (is.na(idx))
        return(callNextMethod(x, name))
    f = accessorFuns[[idx]]
    function(...) {
        f(x, ...)
    }
}
<environment: namespace:SimpleITK>

Signatures:
        x
target  "_p_itk__simple__SmoothingRecursiveGaussianImageFilter"
defined "_p_itk__simple__SmoothingRecursiveGaussianImageFilter"

> filt$SetSigma(5)

itk::simple::SmoothingRecursiveGaussianImageFilter
  Sigma: 5
  NormalizeAcrossScale: 0

> filt$NormalizeAcrossScaleOn()

itk::simple::SmoothingRecursiveGaussianImageFilter
  Sigma: 5
  NormalizeAcrossScale: 1

> res2 <- filt$Execute(im)
> res2
```

Notice that we can explore the accessor functions in the same way as images. Also note that calling the accessor functions without assigning the result to a variable causes the *show* method to display a representation of the object.

- Cryptic error messages - unfortunately it isn't easy to figure out what arguments are expected by the procedural interface. For example, if we assumed that the sigma parameter was a vector, we'd get the following unhelpful response:

```
> try(res3 <- SmoothingRecursiveGaussian(im, c(3, 3)))
> geterrmessage()
```

```
[1] "Error in SmoothingRecursiveGaussian(im, c(3, 3)) : \n  could not find function \"
```

Note that the *try* and *geterrmessage* commands are to allow Sweave to complete. They aren't needed in interactive sessions.

3.3 Still to come

Image arithmetic.

Testing.

3.4 Caveats

Beware of images from saved workspaces. External references, which is how images are represented, are not preserved when objects are saved to disk. Thus, attempting to use images from a saved workspace will result in ungraceful crashes.

4 Building and Installing

Fetch SimpleITK from the git repository and fetch the patches from gerrit containing my updates: http://review.source.kitware.com/5162.

Then go to SimpleITK/Wrapping/Rpackages/ and run

`R CMD INSTALL -library=/path/to/R/library SimpleITK` and you will (eventually) end up with an installed package. This uses the SuperBuild process, but swig has been replaced with the development version containing my changes. So far only tested on linux.

Soon this document should also get created as part of the build process.

5 Development