SimpleITK in R

Richard Beare

June 13, 2018

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. Note that some of the text is written for readers unfamiliar with *R* and can be skipped by experienced users.

Contents

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.

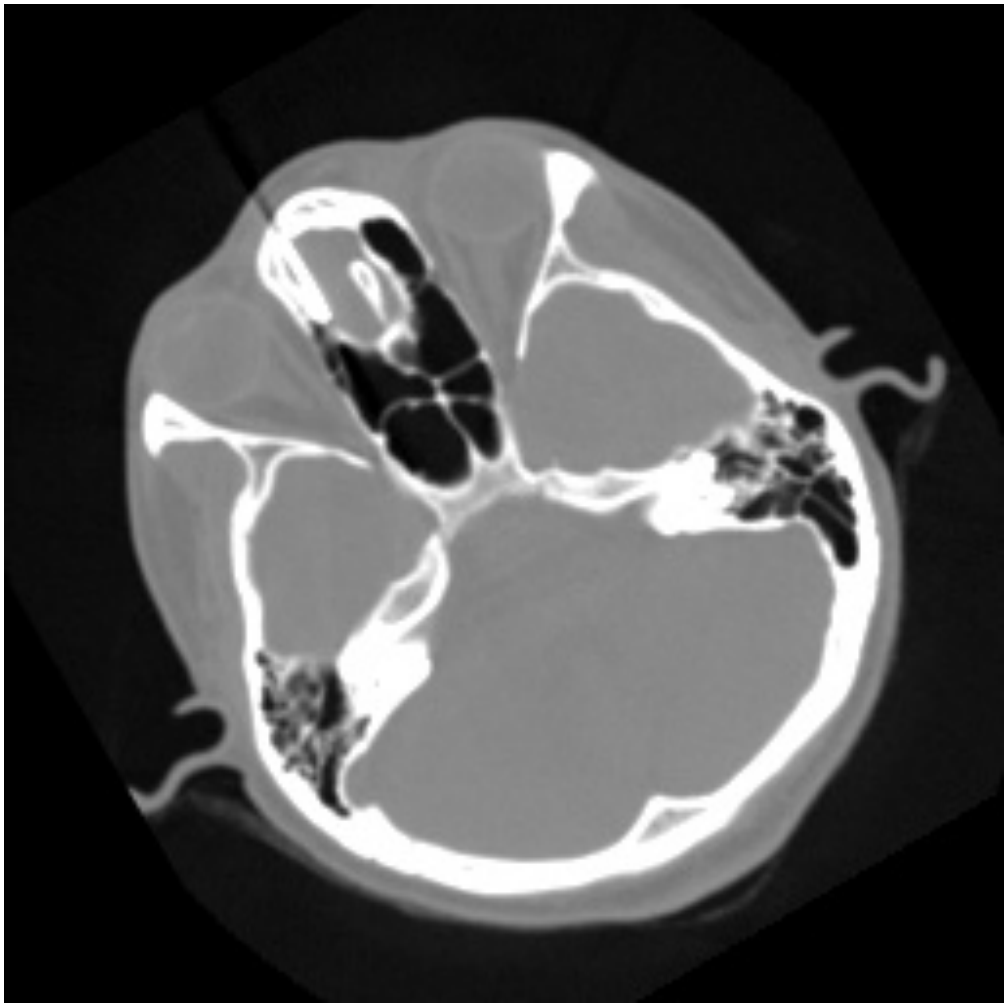
```
> library(SimpleITK)

[1] "show"
```

- Load an image

```
> im <- ReadImage(system.file("data/cthead1.png", package="SimpleITK"))
```

- Display

```
> im
```



- Get some information about the image

```
> print(im)

Image (0x8481120)
  RTTI typeinfo:   itk::Image<unsigned char, 2u>
  Reference Count: 1
  Modified Time: 1889
  Debug: Off
  Object Name:
  Observers:
    none
  Source: (none)
  Source output name: (none)
  Release Data: Off
  Data Released: False
  Global Release Data: Off
  PipelineMTime: 1877
  UpdateMTime: 1888
  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 (0x751dc70)
      RTTI typeinfo:   itk::ImportImageContainer<unsigned long, unsigned char>
      Reference Count: 1
      Modified Time: 1885
      Debug: Off
      Object Name:
      Observers:
        none
      Pointer: 0x7d7f250
      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]
```

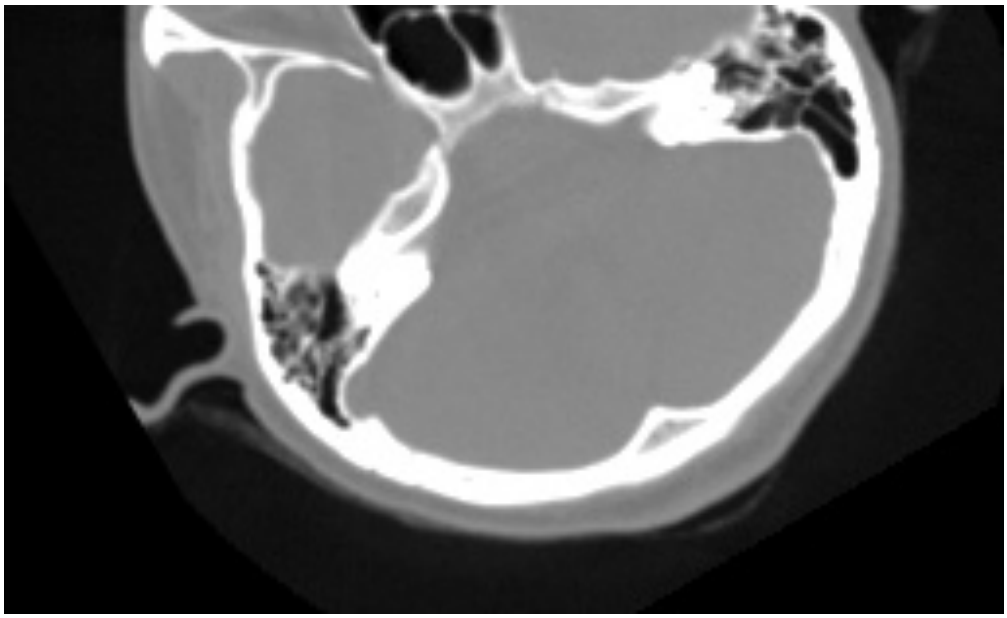
```
[1] 210
```

- Extract the first 100 columns

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

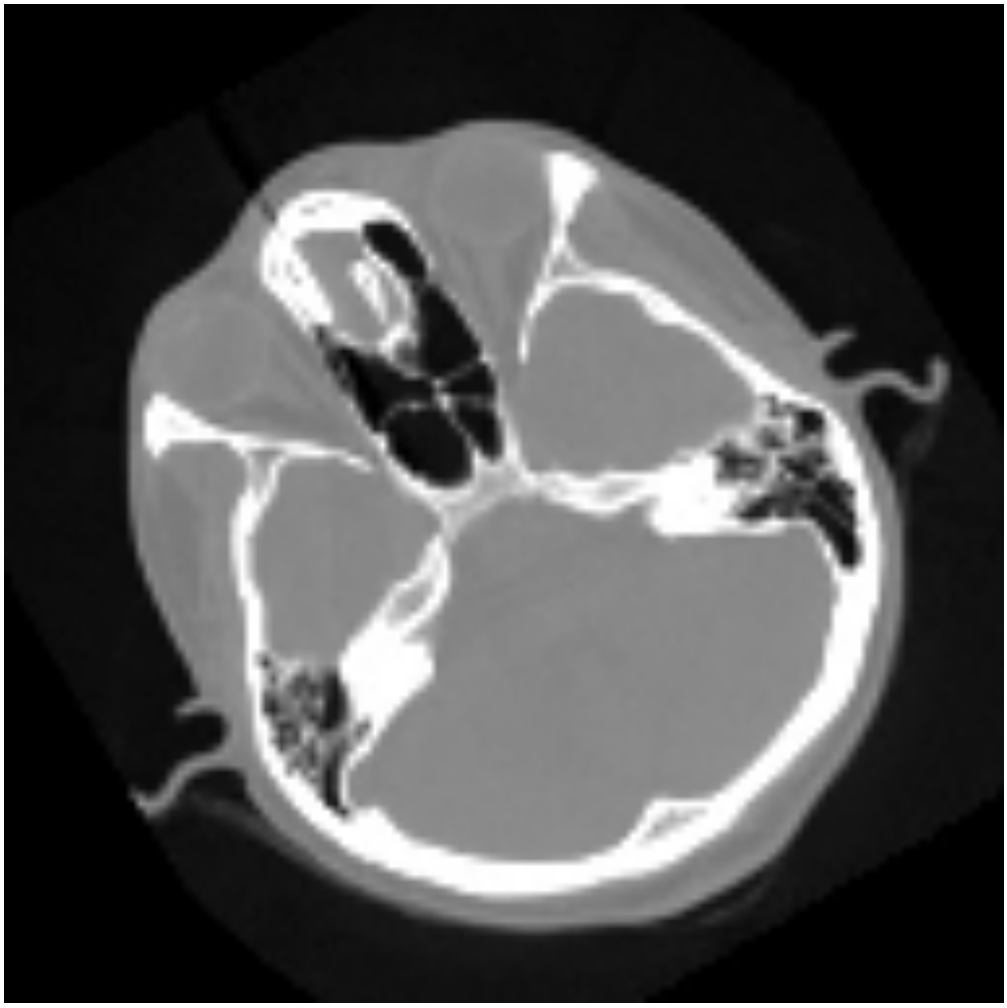
- Remove the first 100 rows

```
> 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 access techniques on images. The results of each of these operations is an image, not an array. These operations are constrained so that rules about image spacing and orientation are preserved. For example, errors are raised if directions are flipped or if spacing is not uniform.

- 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(Equal = Image_Equal, GetITKBase = Image_GetITKBase,
        GetPixelID = Image_GetPixelID, GetPixelIDValue = Image_GetPixelIDValue,
        GetDimension = Image_GetDimension, GetNumberOfComponentsPerPixel = Image_GetNu
        GetNumberOfPixels = Image_GetNumberOfPixels, GetOrigin = Image_GetOrigin,
        SetOrigin = Image_SetOrigin, GetSpacing = Image_GetSpacing,
        SetSpacing = Image_SetSpacing, GetDirection = Image_GetDirection,
        SetDirection = Image_SetDirection, TransformIndexToPhysicalPoint = Image_Transf
```

```
    TransformPhysicalPointToIndex = Image_TransformPhysicalPointToIndex,
    TransformPhysicalPointToContinuousIndex = Image_TransformPhysicalPointToContinu
    TransformContinuousIndexToPhysicalPoint = Image_TransformContinuousIndexToPhys
    GetSize = Image_GetSize, GetHeight = Image_GetHeight,
    GetWidth = Image_GetWidth, GetDepth = Image_GetDepth,
    CopyInformation = Image_CopyInformation, GetMetaDataKeys = Image_GetMetaDataKe
    HasMetaDataKey = Image_HasMetaDataKey, GetMetaData = Image_GetMetaData,
    SetMetaData = Image_SetMetaData, EraseMetaData = Image_EraseMetaData,
    GetPixelIDTypeAsString = Image_GetPixelIDTypeAsString,
    ToString = Image_ToString, GetPixelAsInt8 = Image_GetPixelAsInt8,
    GetPixelAsUInt8 = Image_GetPixelAsUInt8, GetPixelAsInt16 = Image_GetPixelAsInt
    GetPixelAsUInt16 = Image_GetPixelAsUInt16, GetPixelAsInt32 = Image_GetPixelAsI
    GetPixelAsUInt32 = Image_GetPixelAsUInt32, GetPixelAsInt64 = Image_GetPixelAsI
    GetPixelAsUInt64 = Image_GetPixelAsUInt64, GetPixelAsFloat = Image_GetPixelAsF
    GetPixelAsDouble = Image_GetPixelAsDouble, GetPixelAsVectorInt8 = Image_GetPix
    GetPixelAsVectorUInt8 = Image_GetPixelAsVectorUInt8,
    GetPixelAsVectorInt16 = Image_GetPixelAsVectorInt16,
    GetPixelAsVectorUInt16 = Image_GetPixelAsVectorUInt16,
    GetPixelAsVectorInt32 = Image_GetPixelAsVectorInt32,
    GetPixelAsVectorUInt32 = Image_GetPixelAsVectorUInt32,
    GetPixelAsVectorInt64 = Image_GetPixelAsVectorInt64,
    GetPixelAsVectorUInt64 = Image_GetPixelAsVectorUInt64,
    GetPixelAsVectorFloat32 = Image_GetPixelAsVectorFloat32,
    GetPixelAsVectorFloat64 = Image_GetPixelAsVectorFloat64,
    GetPixelAsComplexFloat32 = Image_GetPixelAsComplexFloat32,
    GetPixelAsComplexFloat64 = Image_GetPixelAsComplexFloat64,
    SetPixelAsInt8 = Image_SetPixelAsInt8, SetPixelAsUInt8 = Image_SetPixelAsUInt8
    SetPixelAsInt16 = Image_SetPixelAsInt16, SetPixelAsUInt16 = Image_SetPixelAsUI
    SetPixelAsInt32 = Image_SetPixelAsInt32, SetPixelAsUInt32 = Image_SetPixelAsUI
    SetPixelAsInt64 = Image_SetPixelAsInt64, SetPixelAsUInt64 = Image_SetPixelAsUI
    SetPixelAsFloat = Image_SetPixelAsFloat, SetPixelAsDouble = Image_SetPixelAsDo
    SetPixelAsVectorInt8 = Image_SetPixelAsVectorInt8, SetPixelAsVectorUInt8 = Ima
    SetPixelAsVectorInt16 = Image_SetPixelAsVectorInt16,
    SetPixelAsVectorUInt16 = Image_SetPixelAsVectorUInt16,
    SetPixelAsVectorInt32 = Image_SetPixelAsVectorInt32,
    SetPixelAsVectorUInt32 = Image_SetPixelAsVectorUInt32,
    SetPixelAsVectorInt64 = Image_SetPixelAsVectorInt64,
    SetPixelAsVectorUInt64 = Image_SetPixelAsVectorUInt64,
    SetPixelAsVectorFloat32 = Image_SetPixelAsVectorFloat32,
    SetPixelAsVectorFloat64 = Image_SetPixelAsVectorFloat64,
    SetPixelAsComplexFloat32 = Image_SetPixelAsComplexFloat32,
    SetPixelAsComplexFloat64 = Image_SetPixelAsComplexFloat64,
    MakeUnique = Image_MakeUnique, SetPixel = Image_SetPixel,
    GetPixel = Image_GetPixel)
idx = pmatch(name, names(accessorFuns))
if (is.na(idx))
    return(callNextMethod(x, name))
f = accessorFuns[[idx]]
```

```
    function(...) {
        f(x, ...)
    }
}
<bytecode: 0x79a0108>
<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 (0x58758d0)
  RTTI typeinfo:   itk::Image<unsigned short, 3u>
  Reference Count: 1
  Modified Time: 2033
  Debug: Off
  Object Name:
  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 (0x48b0190)
      RTTI typeinfo:   itk::ImportImageContainer<unsigned long, unsigned short>
      Reference Count: 1
      Modified Time: 2034
      Debug: Off
      Object Name:
      Observers:
        none
      Pointer: 0x75acd50
      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.

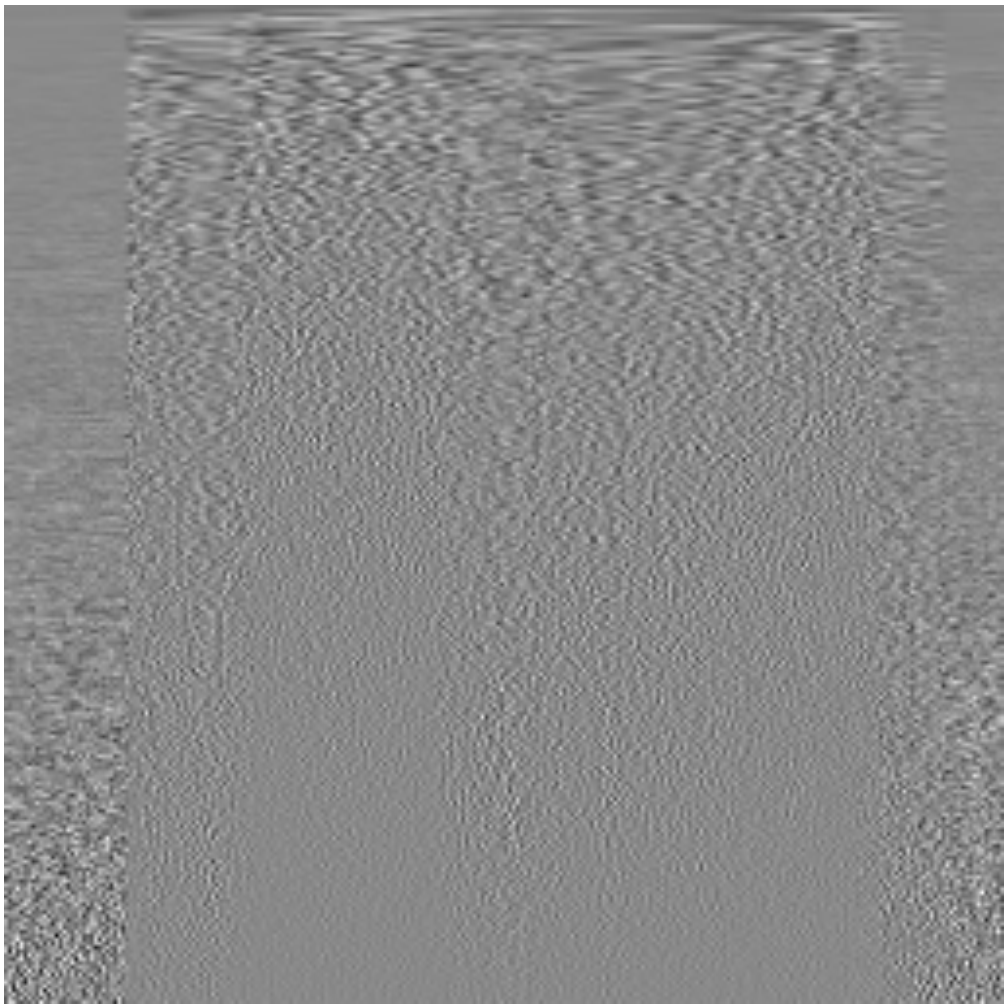
- Translating images to R arrays:

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

[1] "_p_itk__simple__Image"
attr(,"package")
[1] "SimpleITK"

> class(arr)
```

```
[1] "matrix"

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

- And back again

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



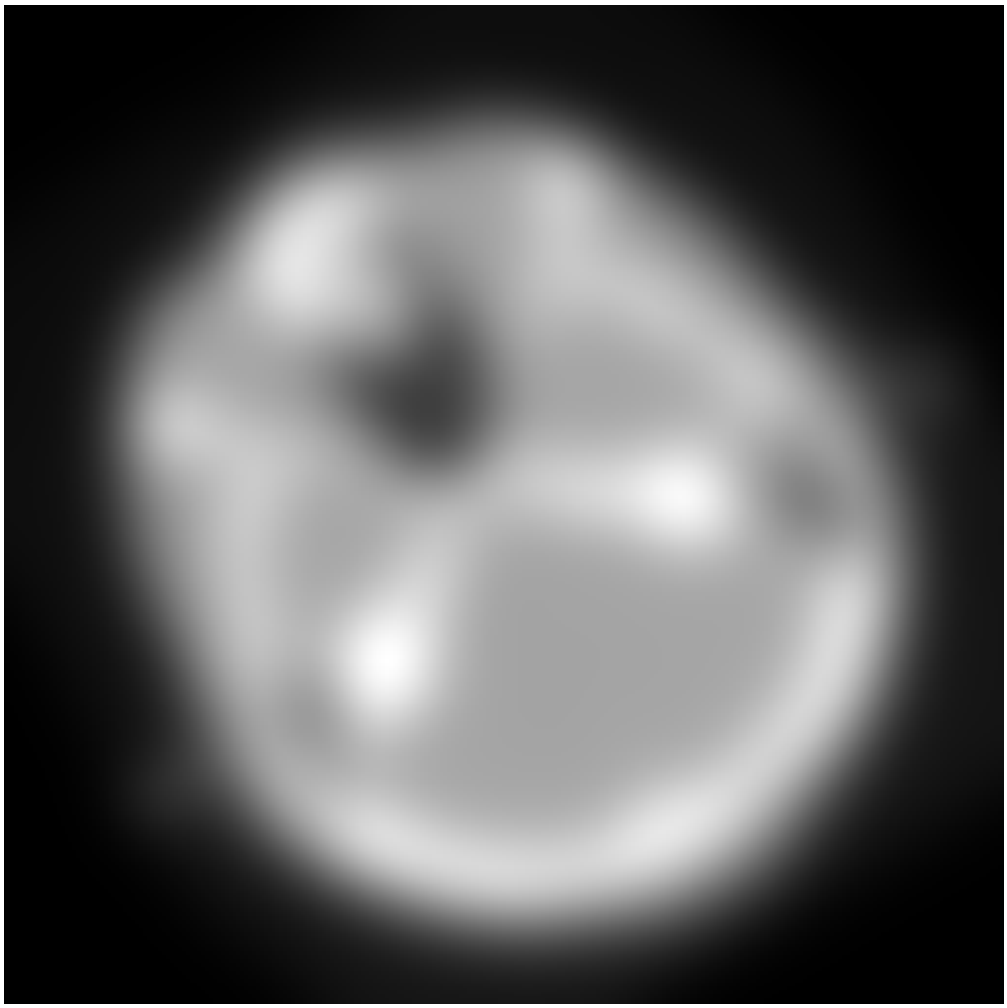
Points to note here - *R* only supports integer and double types (logical types are also integers). Therefore conversion of any image will end up as an array of one of these types. Similarly, conversion of arrays to images also ends up as one of these types. By default, any *R* matrix will be double 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$SetSigma(5)
> filt$NormalizeAcrossScaleOn()
> 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: processing vignette 'SimpleITK_tutorial.Rnw' failed with diagnostics:\n ch
```

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. Visit https://www.itk.org/SimpleITKDoxygen/html/Wrapping.html for the latest instructions on building and installing.

5 Development