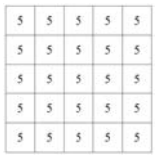
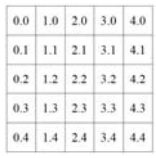
How to use intrinsic variables (icol, irow and other i's)

Some of the most versatile scripting possibilities are rarely recognized, and often feared by most users: The suite of intrinsic variables or 'i-commands'. This tutorial should help to understand these and demonstrated useful applications.

intrinsic variables; icol; iwidth; irow; iheight; iplane; idepth; ipoints; iradius; itheta; binning

Intrinsic variables are number variables which values depend on the image or image expression they are used in. Some of them (e.g. iheight, iwidth) are constant all over an image, while others (e.g. icol, irow) even depend on the pixel-position they are used for. To complicate things, intrinsic variables have different meanings depending on the way they are used in expressions. There are, however, certain points which are always valid:
Intrinsic variables can only be used in expressions of known 'physical' size.
Intrinsic variables can only be used on one side of an expression, either left or right of the equal sign.
Maybe the best way to learn how intrinsic variables work is to look at example scripts like the following. It simply creates some images and assigns the values of the intrinsic variables to the images pixels:

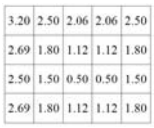
```
Image img1
img1 := RealImage("icol",4,5,5)
img1 = icol
img1.ShowImage()
img1.SetDisplayType(5)     // Show image as spread sheet
Image img2
Img2 := RealImage("iwidth",4,5,5)
Img2 = iwidth
Img2.ShowImage()
Img2.SetDisplayType(5)     // Show image as spread sheet
Image img3
Img3 := RealImage("irow",4,5,5)
Img3 = irow
Img3.ShowImage()
Img3.SetDisplayType(5)     // Show image as spread sheet
Image img4
Img4 := RealImage("iheight",4,5,5)
Img4 = iheight
Img4.ShowImage()
Img4.SetDisplayType(5)     // Show image as spread sheet
Image img5
Img5 := RealImage("ipoints",4,5,5)
Img5 = ipoints
Img5.ShowImage()
Img5.SetDisplayType(5)     // Show image as spread sheet
Image img6
Img6 := RealImage("maths",4,5,5)
Img6 = icol + 0.1 * irow
Img6.ShowImage()
Img6.SetDisplayType(5)     // Show image as spread sheet
```

					
icol	iwidth	irow	ihight	ipoints	'maths'

As you can see from the example, you can think of the variables as numbers which depend on the position within and properties of the image expression. The physical size of the image expression is given by the image within the expression. This example is for 2D images only, but the extension into 3D is given by the two variables iplane (for the total number of slices) and idepth (for the z-position of a voxel).
Another set of useful intrinsic variables are iradius and itheta, which give the distance and the angle of a point with respect to the image centre. Here it is important to notice how the 'centre' of a pixel is defined within DM, and where (within a pixel) the pixel's value is 'located'. The rules are as follows:
The value of a pixel always refers to the top/left corner position of the pixel (- the grid crossing points!)
The centre of an image is defined by half the distance from boarder to boarder (- the centre of the *area*!)
This has severe influence on the values returned by iradius and itheta, especially as far as one-dimensional images (aka line profiles) are concerned. Again it is best understood by looking on the results of the following example script:

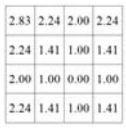
```
Image img1
img1 := RealImage("iradius 5x5",4,5,5)
img1 = iradius
img1.ShowImage()
img1.SetDisplayType(5)     // Show image as spread sheet
Image img2
Img2 := RealImage("iradius 4x5",4,4,5)
Img2 = iradius
Img2.ShowImage()
Img2.SetDisplayType(5)     // Show image as spread sheet
Image img3
Img3 := RealImage("iradius 4x4",4,4,4)
Img3 = iradius
Img3.ShowImage()
Img3.SetDisplayType(5)     // Show image as spread sheet
Image img4
Img4 := RealImage("iradius 5x4",4,5,4)
Img4 = iradius
Img4.ShowImage()
Img4.SetDisplayType(5)     // Show image as spread sheet
Image img5
Img5 := RealImage("iradius 5x1",4,5,1)
Img5 = iradius
Img5.ShowImage()
Img5.SetDisplayType(5)     // Show image as spread sheet
Image img6
Img6 := RealImage("iradius 4x1",4,4,1)
Img6 = iradius
Img6.ShowImage()
Img6.SetDisplayType(5)     // Show image as spread sheet
```

iradius 5x5

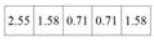


iradius 5x4

iradius 4x5



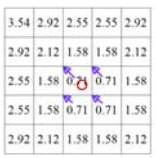
iradius 4x4



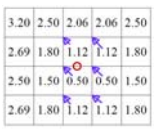
iradius 5x1

iradius 4x1

At first glance, these values look odd, but if you remember the two rules, then they are exactly as one could expect. For easier visualization, the centre has been marked as red dot, and the 'positions' of the values as blue arrows in the following table:

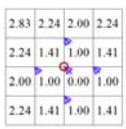


iradius 5x5



iradius 5x4

iradius 4x5

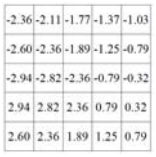


iradius 4x4

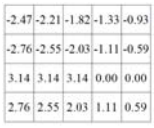
iradius 5x1

iradius 4x1

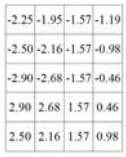
The same 'shift' of coordinates appears on using itheta, which gives the angle (rad) between a horizontal line and the connection of the image centre with the pixel.



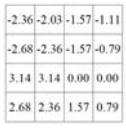
itheta 5x5



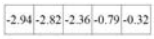
itheta 5x4



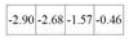
itheta 4x5



itheta 4x4



itheta 5x1



itheta 4x1

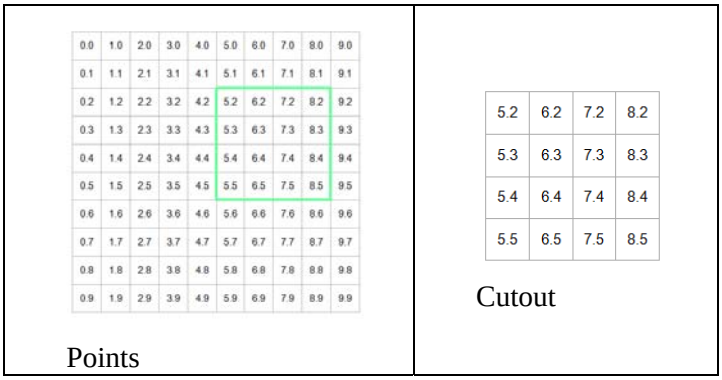
As said before, this way of defining the centre of an image has to be considered in special for performing calculations with 1D-line profiles, because the centre is then not the centre on the line, but the centre of a rectangular area of 1 pixel height instead.

A combination of the intrinsic variables can be used to create all kind of images. In combination with the tert() command, they can be used to easily create binary masks, as the following example demonstrates:

```
Image mask
mask := RealImage("Mask",4,1024,1024)
mask = Tert( Mod(Abs(Sin(itheta)*iheight/iradius),2) <= 1 , 1 , 0)
mask.ShowImage()
```

The use of intrinsic variables becomes even more versatile when they are used as index coordinates. Just look at the following script:

```
Image img1, img2
img1:= RealImage("Points",4,10,10)
img1= icol+0.1*irow
img1.ShowImage()
img1.SetDisplaytype(5)
img2:= RealImage("Cutout",4,4,4)
img2= img1[icol+5,irow+2]
img2.ShowImage()
img2.SetDisplaytype(5)
```



5.2	6.2	7.2	8.2
5.3	6.3	7.3	8.3
5.4	6.4	7.4	8.4
5.5	6.5	7.5	8.5

In the first part of the script we simple created the image 'Points', but in the 7th line we did something new: We indexed the image using intrinsic variables. Note that instead of using four coordinates (top, left, bottom, right) we have used only two parameters, one for the x-dimension, the other for the y-dimension. The value of the parameters, however, iterate because they are intrinsic variables. Their 'range' is defined by the size of the image on the left side of the equal sign (img2). In a pixel-to-pixel assignment of values, img2 thus gets the values of img1 at the shifted position. Using this kind of assignment with intrinsic variables, all kind of transformations can be easily achieved. These transformations work much faster than pixel-by-pixel manipulation using GetPixel() & SetPixel() commands and should therefore be used whenever possible! (And with some thinking, there is nearly always a possibility…). The following script gives just a few examples of often carried out transformations:

```
Image img := RealImage("Source",4,8,8)
img = icol+0.1*irow
img.ShowImage()
img.SetDisplayType(5)

Image flip_h := RealImage("horizontally flipped",4,8,8)
flip_h = img[iwidth-icol,irow]                                    // flip image horizontally
flip_h.ShowImage()
flip_h.SetDisplayType(5)

Image sample_2 := RealImage("sampled 2x2",4,8/2,8/2)
sample_2 = img[icol*2,irow*2]                                  // sample every other data point
sample_2.ShowImage()
sample_2.SetDisplayType(5)

Image trans := RealImage("transposed",4,8,8)
trans = img[irow,icol]                                              // transpose the image
trans.ShowImage()
trans.SetDisplayType(5)
```

So far we have used the intrinsic variables always on the right hand side of an assignment. However, they can also be used on the left hand side as shown in the next example:

```
Image img := RealImage("Source",4,8,8)
Image zero := RealImage("Zero Image",4,8/2,8/2)
zero = 0
img = icol+0.1*irow
img.ShowImage()
img.SetDisplayType(5)
img.UpdateImage()
Sleep(2)
img[icol*2,irow*2] = zero
```

This script creates a test image, and after two seconds every other data point is set to zero. Note, that now that we are using intrinsic variables on the left side of the equation, it is the image expression on the right side which defines the range of the variables! Remember: We can not use intrinsic variables on both sides at the same time.

Another very useful application of intrinsic variables is the expansion of some data along an additional dimension. The following script takes a line profile and then creates a 2D image with each line being a copy of the line profile. It then creates another image by rotating the line profile along the centre. Note, however, that no data interpolation is performed, and that this second image therefore is heavily aliased.

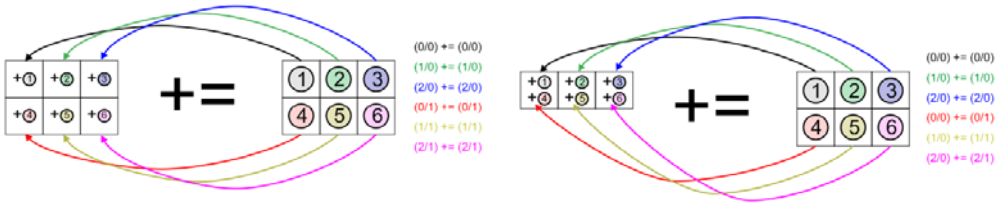
```
Image line, img1, img2
line := RealImage("Line",4,64,1)
line = icol*Random()
img1 := RealImage("Image 1",4,64,64)
img1 = line[icol,0]
img1.ShowImage()
img2 := RealImage("Image 2",4,128,128)
img2 = line[iradius,0]
img2.ShowImage()
```

Finally, intrinsic variables can also be used for a kind of implicit for-loops. The following script sums all lines of an image into one line profile:

```
Image line, img
img := RealImage("Image",4,64,64)
img = icol+iradius
line := RealImage("Line",4,64,1)
line[icol,0]  += img
img.ShowImage()
line.ShowImage()
Result("\n Sum of Image = "+Sum(img))
Result("\n Sum of Line  = "+Sum(line))
```

This time the syntax is harder to understand, but it is maybe best understood if one thinks of it the following way: Image expressions are always evaluated pixel-by-pixel serially. For normal assignments, this leads to a simple pixel-to-pixel assignment, and it does not matter that it is performed serially. Using the intrinsic variable, however, maps different coordinates into the same, which lead to a summing during serial calculation.

Img1 += Img2 Img1[icol,0] += Img2



This kind of implicit loop is much faster than a regular for-loop in a script and should thus be preferably used. For example, any binning of data can be easily and fast achieved by the following script function:

```
Image AnyBin(Image in, Number bin)
   {
   Number sx,sy
   in.GetSize(sx,sy)
   Image out := RealImage(in.GetName()+" "+bin+"x"+bin+" binned",4,sx/bin,sy/bin)
   out = 0
   out[icol/bin,irow/bin] += in
   If ((mod(sx,bin)+mod(sy,bin))!=0) Result("\n Warning: Uneven binning, boarder pixels are affected.")
   return out
   }

Image img,binned
img := RealImage("Image",4,64,64)
img = Random()
img.ShowImage()
binned := AnyBin(img,4)
binned.ShowImage()
Result("\n Sum of Image = "+Sum(img))
Result("\n Sum of Image,binned = "+Sum(binned))
```