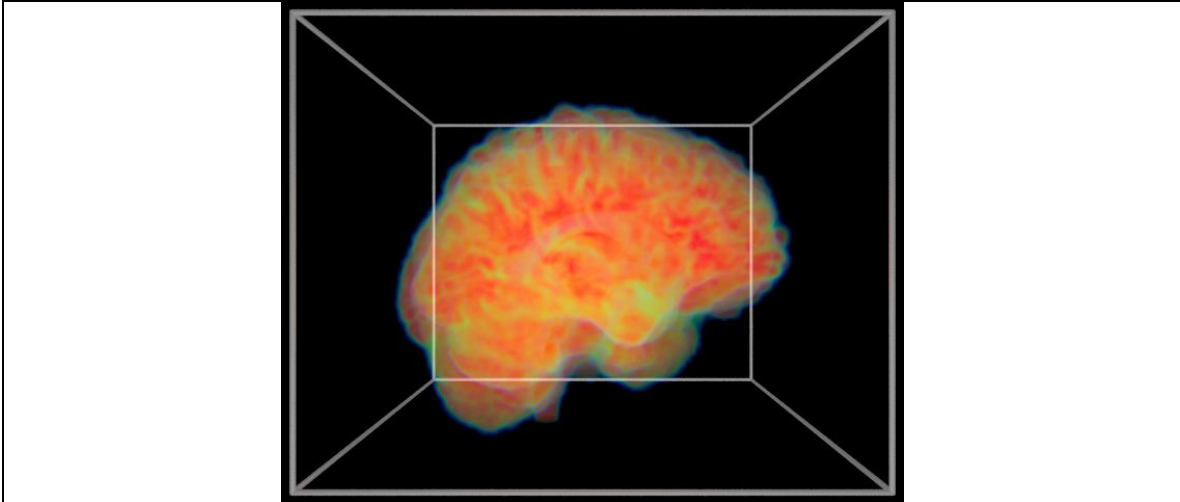
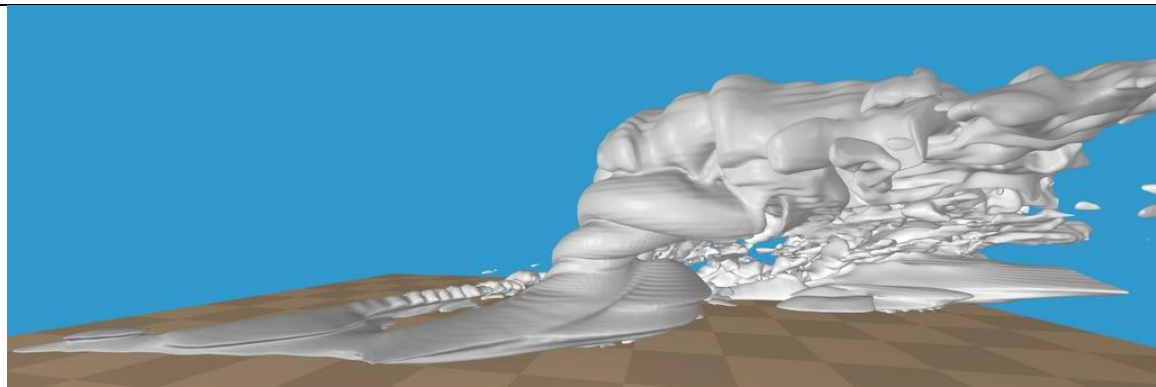

L18 – POV-Ray - Part 4

1. Volume Visualization

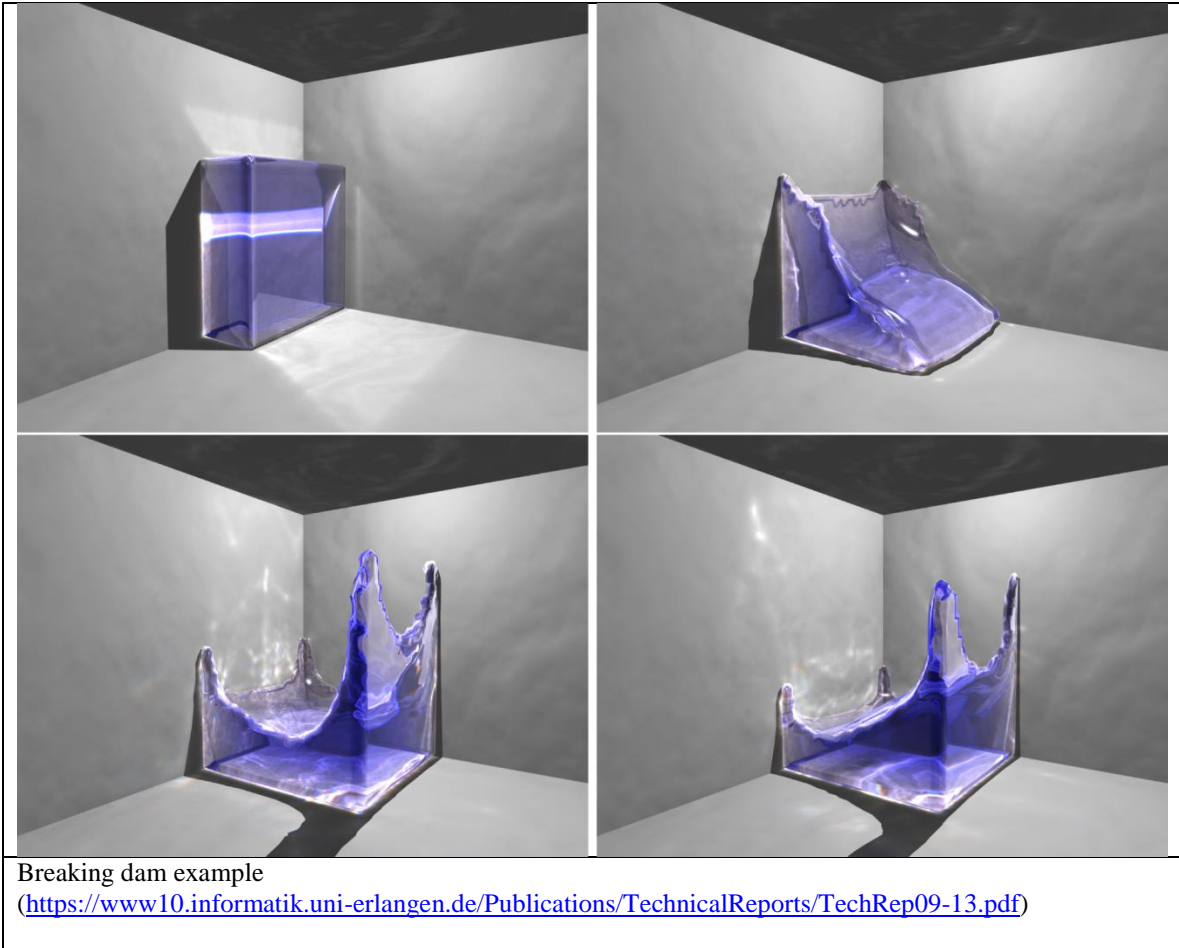
In this lecture we will talk about how to do volume visualization in POV-Ray. I haven't found a large number of examples on the web related to scientific visualization and POV-Ray, but I include a couple of images I gathered from around the web.



Brain on fire. (<http://paulbourke.net/miscellaneous/povexamples/>)



Atmospheric sciences rendering of a supercell
(<http://www.linuxjournal.com/article/7486>)



2. Density Files

The way to visualize 3D data in POV-Ray is to store these data in a density file or **‘.df3’** file. This isn’t the most convenient file format, but, ok, it is at least workable. The format is as follows. The data is defined inside a unit cube from $\langle 0,0,0 \rangle$ to $\langle 1,1,1 \rangle$. In addition, the data values have to be unsigned integers. The integer density values can be written as either 1-byte (this is 8 bits, so has a range from 2^8-1 . That is, density values range from: 0-255), 2-byte (range: 0-65,535), or 4-byte integers (range: 0-4,294,967,295), but they **MUST** be integer values and unsigned!

The file format is as follows.

- (1) A header is provided that specifies the size of the 3D array. The array size must be written as 2-byte integers. So, you have three integers that are each 2-bytes long (total of 6-bytes) in the header. So, if the array were $64 \times 64 \times 64$, you would first write a line:
 64 64 64

- (2) This header is followed by the data values. As noted the data values can be 1-byte, 2-byte, or 4-byte integers. The convention used by POV-Ray is that values in the x-direction vary the fastest, and values in the z-direction vary the slowest.
- (3) The file must be written as a binary (unformatted) file that is Big Endian.
- (4) POV-Ray checks the size of the file to determine if the data values are 1-, 2-, or 4-bytes. For example, if you have an array that is $64 \times 64 \times 64$, and if you write the data values as 2-byte integers, the file size should be $(2 \text{ bytes}) \times (64 \times 64 \times 64 \text{ data values}) + 6 \text{ bytes} = 524,294 \text{ bytes}$. Recalling that the header is composed of three 2-byte integers (a total of 6 bytes).

So, let's consider that we have data in a $2 \times 2 \times 2$ array. The file would be written like:

```

2 2 2
V(x1, y1, z1)
V(x2, y1, z1)
V(x1, y2, z1)
V(x2, y2, z1)
V(x1, y1, z2)
V(x2, y1, z2)
V(x1, y2, z2)
V(x2, y2, z2)

```

Where $V(x,y,z)$ is the density value at position x,y,z . Of course, the file must be written as a binary file. The following example shows a simple Fortran90 subroutine to properly write out a density file. Note the following from this example for writing the output:

```

OPEN(UNIT=1, FILE=ofile, ACCESS='DIRECT', FORM='UNFORMATTED', RECL=6, STATUS='REPLACE', CONVERT
='BIG_ENDIAN')

```

- We use `ACCESS='DIRECT'` because Fortran puts an additional control word in the binary file, if you use `ACCESS='SEQUENTIAL'`. Thus, sequential access files would not be of the correct size.
- We use `RECL=6` for the header as it should be 6-bytes long, and `RECL=2` for the data values if we are writing 2-byte integers.
- Many compilers have a flag `CONVERT='BIG_ENDIAN'`, which makes things much easier. Otherwise you should use caution remembering to swap byte order if you are working on a linux machine before writing out the files.
- Since we are using direct access, I also like to make sure I include `STATUS='REPLACE'` to replace the file before writing. Otherwise, again, one may end up with a file size that doesn't match expectations.

```

!-----!
! WRITEDENSFILE
!
! Subroutine to write out .df3 format file (density file) for use with POVray
! Note, this subroutine is designed to write out 2-byte integers for the density values.
!
!-----!
SUBROUTINE writedensfile(NX,NY,NZ,outdata,ofile)
IMPLICIT NONE
INTEGER(KIND=2), INTENT(IN) :: NX, NY, NZ           !array dimensions
INTEGER(KIND=2), DIMENSION(NX,NY,NZ), INTENT(IN) :: outdata !density values to write out
INTEGER(KIND=4) :: x, y, z                         !loop variables
INTEGER(KIND=4) :: counter                         !record counter for direct access output
INTEGER(KIND=2) :: bitx, bity, bitz, bitd          !test bit size
CHARACTER(LEN=60), INTENT(IN) :: ofile             !name of output file to write to

! First test bit size, header MUST always be written as 2-byte integers
! In this subroutine I use KIND=2 to indicate 2-byte integers. But,
! note that some compilers use a sequential KIND ordering, and KIND=2
! may not always indicate 2-byte integers. This is intended to check
! that possibility.
bitx = bit_size(NX)
bity = bit_size(NY)
bitz = bit_size(NZ)
bitd = bit_size(outdata)
IF (bitx == 16 .AND. bity == 16 .AND. bitz == 16 .AND. bitd == 16) THEN
  write(*,*) "Writing density file: ", TRIM(ADJUSTL(ofile)), "...
  write(*,*) "Expected file size is: ", 2*(NX*NY*NZ) + 6, " (bytes)"
ELSE
  write(*,*) "ERROR: INTEGER bit size does not equal 16. Exiting Now..."
  write(*,*) "Bit Size of NX: ", bitx
  write(*,*) "Bit Size of NY: ", bity
  write(*,*) "Bit Size of NZ: ", bitz
  write(*,*) "Bit Size of output data: ", bitd
  STOP
ENDIF

! First write out 6-byte header with array dimensions
! Output file is binary and MUST be BIG_ENDIAN
OPEN(UNIT=1,FILE=ofile,ACCESS='DIRECT',FORM='UNFORMATTED',RECL=6,STATUS='REPLACE',CONVERT='BIG_ENDIAN')
WRITE(1,rec=1) NX, NY, NZ
CLOSE(1)

! Now write out density values
counter = 4
OPEN(UNIT=1,FILE=ofile,ACCESS='DIRECT',FORM='UNFORMATTED',RECL=2,CONVERT='BIG_ENDIAN')
DO z=1,NZ
  DO y=1,NY
    DO x=1,NX
      write(1,rec=counter) outdata(x,y,z)
      counter = counter + 1
    ENDDO
  ENDDO
ENDDO
CLOSE(1)

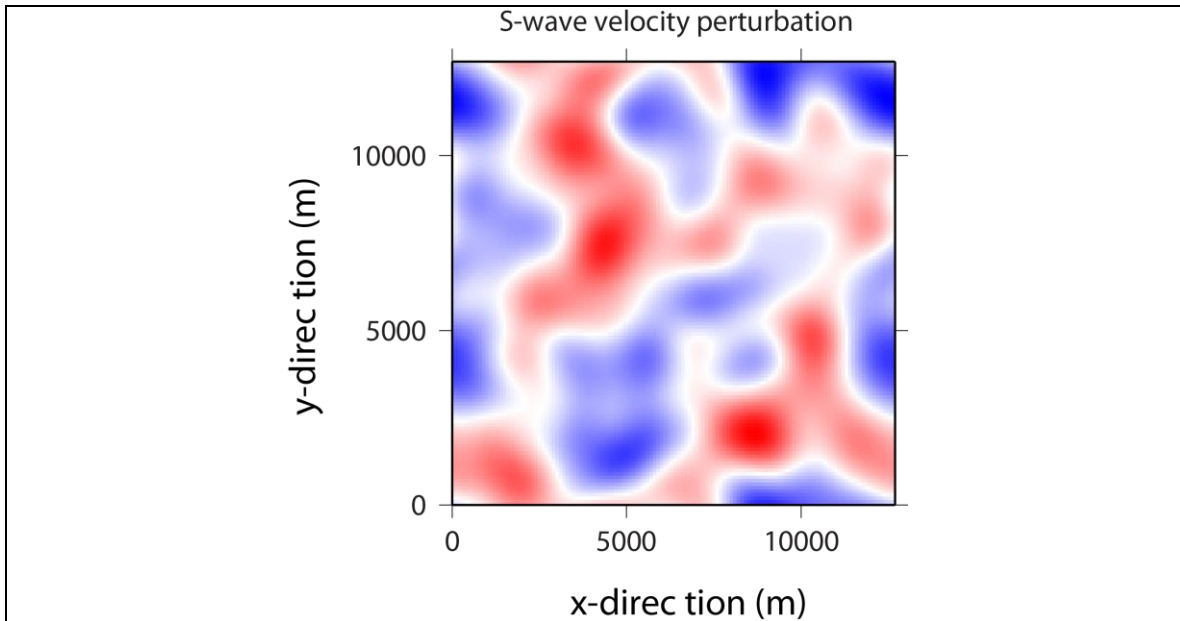
END SUBROUTINE writedensfile
!-----!

```

3. Isosurfaces – Basic Syntax

Now that we know what format of file we are trying to write out, let's generate a file. Then, we can create an image using the POV-Ray function [isosurface](#).

First, let's take a look at how the isosurface function works. Let's consider an example from some of my own work. In order to simulate scattering of seismic waves, one thing that can be done is that random seismic velocity perturbations can be applied to the background seismic velocity model. The details are not important; basically one just convolves a function with a random set of numbers. For example, the following image shows a 2D slice of random velocity perturbations where a random set of numbers was convolved with a Gaussian function. Here we have just shown perturbations to the velocity as scaled by color (red is negative perturbations and blue is positive perturbations).



But, we can also write the 3D values of these perturbations out as a density file. The file is called random.df3 and is available for download on the course webpage. Now we can use the isosurface function to trace out volumes of constant velocity perturbation. Consider the next example POV-Ray script.

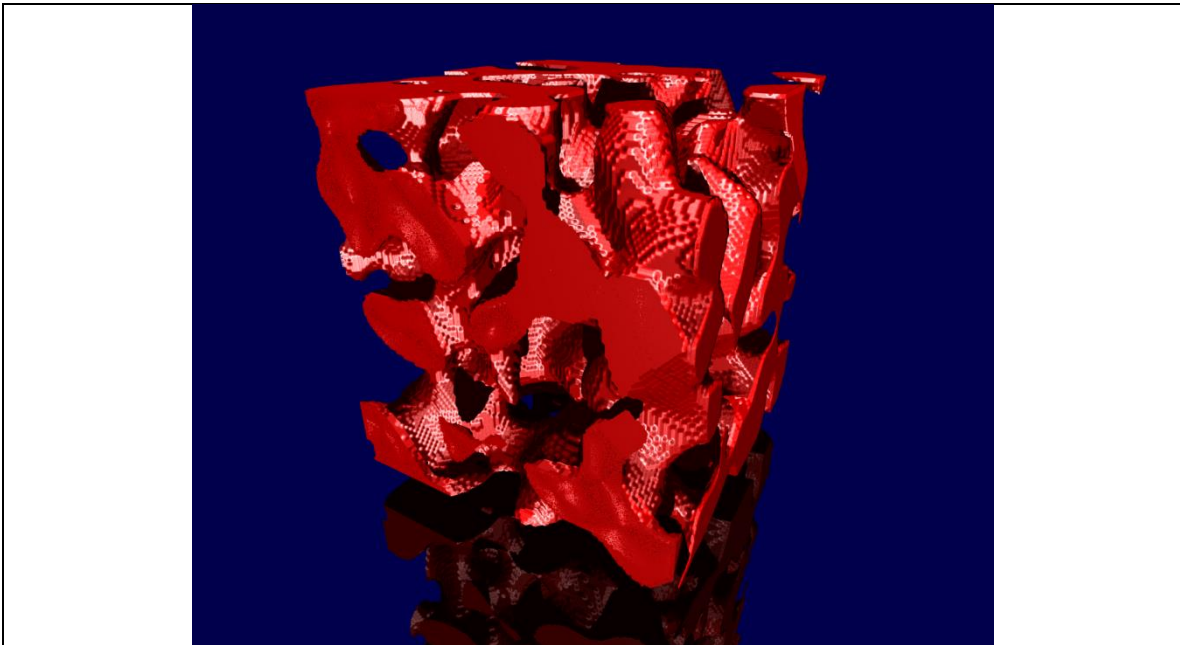
```
camera {perspective location <0,5,-10> look_at <0,0,0>}

light_source {<0,0,-10> color rgb <1,1,1> parallel}
light_source {<0,10,0> color rgb <1,1,1> parallel}
light_source {<5,-5,5> color rgb <1,1,1> parallel}
background {color rgb <0,0,1>}

plane {<0,1,0>, -2.5 pigment {color rgb <0,0,0> }
      finish {reflection 0.3}
}
```

```
#declare F1= function {  
    pigment {  
        density_file df3 "./random.df3"  
        interpolate 1  
        translate -0.5  
    }  
}  
  
isosurface {  
    function { 1-F1(x,y,z).gray }  
  
    contained_by {box{-0.5,0.5}}  
    threshold 0.5  
    accuracy 0.1  
    max_gradient 100  
    scale 6  
    pigment {rgb <1,0,0>}  
    finish {  
        specular 1.0  
    }  
    rotate <0,30,0>  
    translate <0,1,0>  
}
```

Running this script will produce the following image.

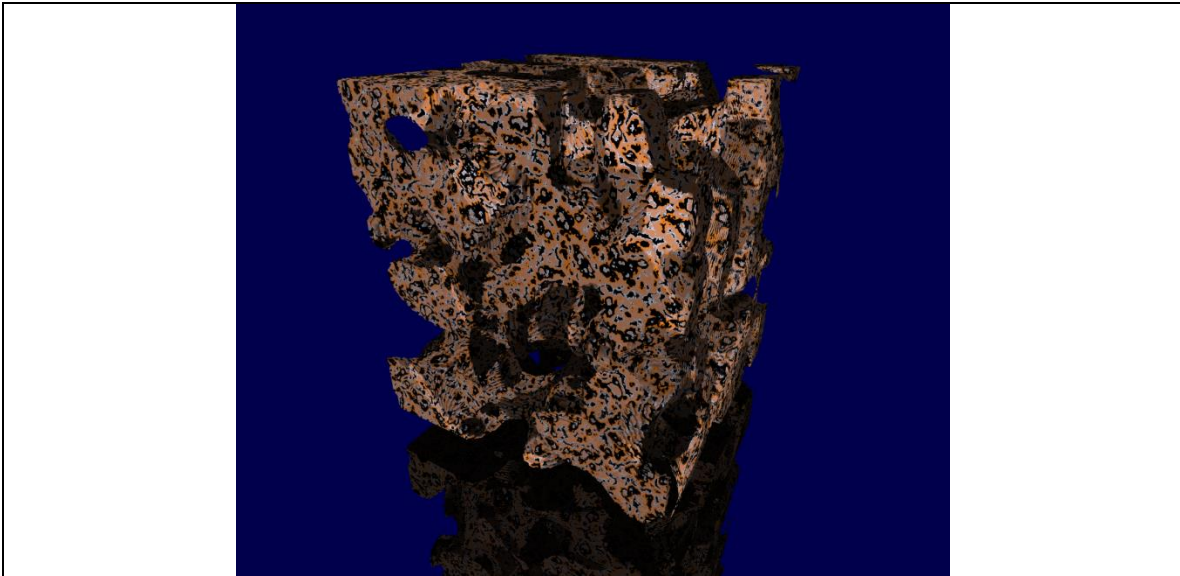


For fun you can try and make it look more rock-like. Add the following line to your code:

```
#include "textures.inc"
```

Then make the pigment something like:

```
pigment {Pink_Granite frequency 2}
```



Other pigment and texture options can be found in the reference cards on the class webpage.

4. Isosurfaces – More detail

Let's think about how to make isosurface objects a little more closely now. For example, let's consider how to make a sphere (i.e., without the POV-Ray sphere function).

We can write a code that will loop through x , y , and z calculating $x^2 + y^2 + z^2$. For simplicity, let's allow x , y , and z to vary between -0.5 and $+0.5$, and then calculate $v(x,y,z) = x^2 + y^2 + z^2$ at each point.

At the center of our grid, $v(0,0,0) = 0$, and it should have a maximum value of $v(0.5,0.5,0.5) = 0.75$.

But, remember that we need to write out a **.df3** file that is an integer. So, we should really rescale our real values to something useful.

For example, consider that we are working in Fortran90 and we calculated our function $v(x,y,z) = x^2 + y^2 + z^2$. That is, we put the values in the REAL array **v**. We can next rescale the values (now ranging from 0 to 0.75) as follows:

```
v(:,:,:) = (v(:,:,:) / MAXVAL(v(:,:,:))) * 65535
```

Where I chose the value 65,535 as the range of available 2-byte integers.

Now, our array `v` should vary between 0 and 65,535.

Before writing our array out to the file `sphere.df3` we need to make the values 2-byte integers, so we might have declared an integer array as:

```
INTEGER(KIND=2) , DIMENSION(NX,NY,NZ) :: odata
```

And we can simply write:

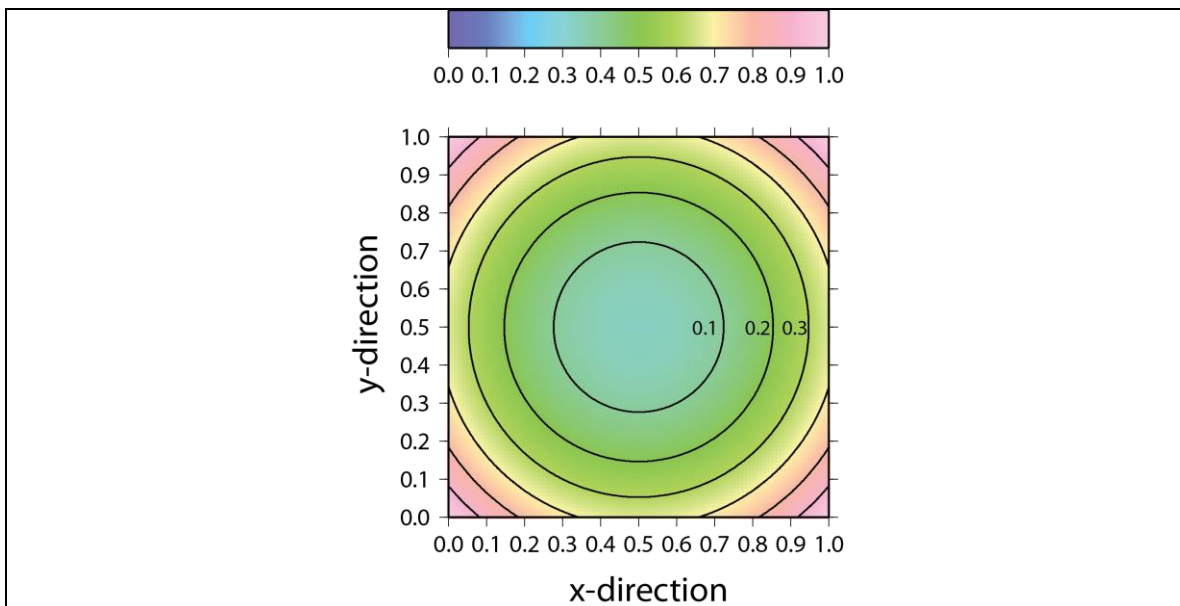
```
odata(:,:,:) = NINT(v(:,:,:))
```

and we can call our subroutine for writing out `.df3` files:

```
CALL writedensfile(NX,NY,NZ,odata,'sphere.df3')
```

OK, but remember, POV-Ray does some things to our data. It scales the values as real numbers between 0 and 1 and inside a unit interval cube. Important Note: It seems that POV-Ray scales values based on the maximum integer value available. Hence, if you are using 2-byte integers, POV-Ray scales a value of 65,535 to 1.0. That is, POV-Ray does not scale values based on the largest data value in your `.df3` file.

So, looking at values at `Z=51`, which is about the middle of the volume:



Now, the above diagram is very instructive for determining what POV-Ray is going to draw isosurfaces on. Note that I drew in contour lines spaced at a 0.1 contour interval.


```

//Set up the scene
//-----//
#include "textures.inc"

camera {location <0,2,-1> look_at <0,-0.5,-0.2>}
light_source {<-5,0,-5> color rgb <1,1,1> spotlight }
background {color rgb <0,0,0>}

plane {<0,1,0>, -2 pigment {rgb <0,0,0> }
      finish {reflection .3}}
//-----//

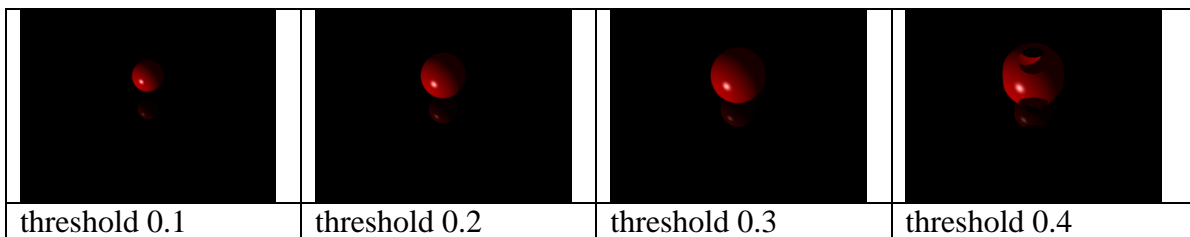
// declare the density file as a function
//-----//
#declare F1= function {
    pigment {
        density_file df3 "./sphere.df3"
        interpolate 1
    }
}
//-----//

// draw the isosurface
//-----//
isosurface {
    function {F1(x,y,z).grey }

    contained_by { box{<0.01,0.01,0.01>,<.99,.99,.99> } }
    threshold 0.1
    open
    accuracy 0.001
    max_gradient 50
    scale 1
    pigment {rgb <1,0,0>}
    finish { phong 0.8 }
    rotate <0,0,0>
    translate <-.5,-1.5,0>
}
//-----//

```

Try some different scales:



Note that setting the threshold at 0.4 the sphere is now larger than the containing box. Hence, its edges get clipped. If you don't use the **open** option above, then the edges get filled in with the edges of the containing box.

Note that the **threshold** sets the value to draw the isosurface around.

Now, consider the following example. Can you explain the result?

```
//Set up the scene
//-----//
#include "textures.inc"

camera {location <0,2,-2> look_at <0,-0.5,0>}

light_source {<-5,0,-5> color rgb <1,1,1> spotlight }

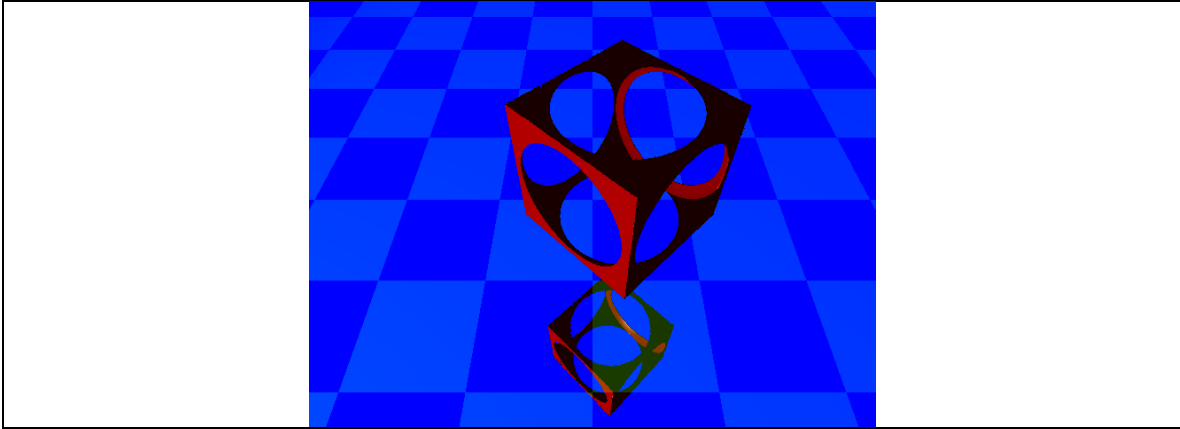
background {color rgb <0,0,1>}

plane {<0,1,0>, -1.5 pigment {checker rgb <0,0,0> }
      finish {reflection 1}}
//-----//

// declare the density file as a function
//-----//
#declare F1= function {
    pigment {
        density_file df3 "./sphere.df3"
        interpolate 1
    }
}
//-----//

// draw the isosurface
//-----//
isosurface {
    function {.6-F1(x,y,z).gray }

    contained_by {box{<0,0,0>,<1,1,1>}}
    threshold 0.0
    accuracy 0.001
    max_gradient 50
    scale 1
    pigment {rgb <1,0,0>}
    finish { phong 0.8 }
    rotate <0,45,0>
    translate <-.5,-.5,0>
}
//-----//
```



Now you can try something more interesting. For example, the webpage has a file called **bretzel.df3** which is calculated for a 5-hole Bretzel function:

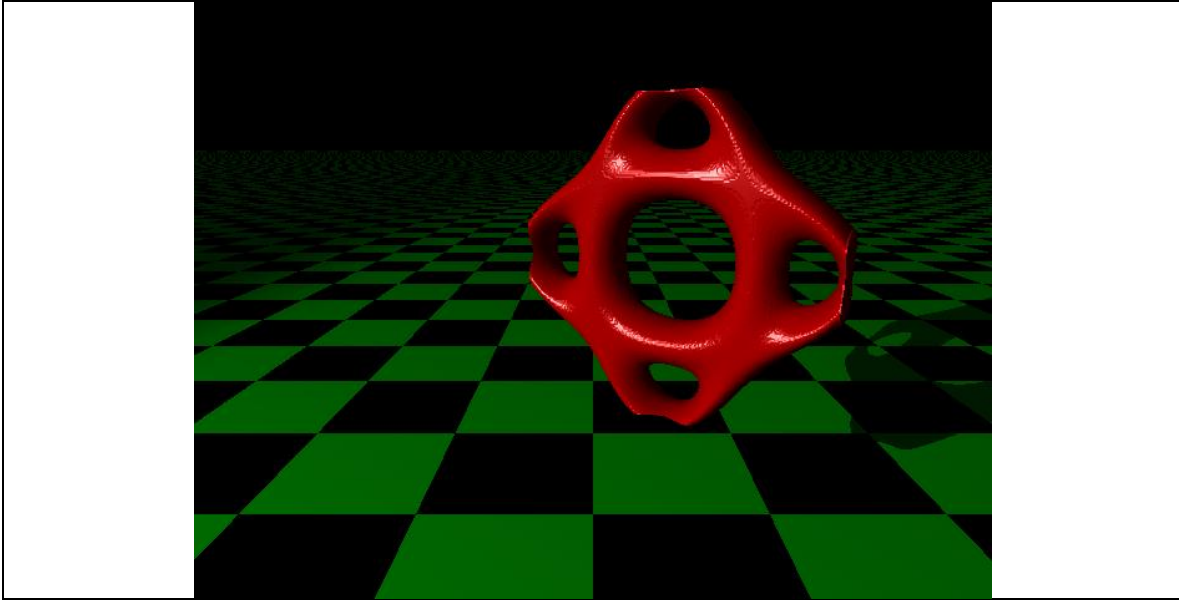
$$v(x, y, z) = \left(\left(x^2 + \frac{y^2}{4} - 1 \right) \left(\frac{x^2}{4} + y^2 - 1 \right) \right)^2 + z^2 - 0.1$$

Try imaging it as follows

```
// declare the density file as a function
//-----//
#declare F1= function {
    pigment {
        density_file df3 "./bretzel.df3"
        interpolate 1
    }
}
//-----//

// draw the isosurface
//-----//
isosurface {
    function {F1(x,y,z).grey }

    contained_by { box{<0.001,0.001,0.001>,<.999,.999,.999> } }
    threshold 0.001
    //open
    accuracy 0.001
    max_gradient 50
    scale 1.2
    pigment {rgb <1,0,0>}
    finish { phong 0.8 }
    rotate <0,30,0>
    translate <-.5,-1.,0>
}
//-----//
```



The class webpage also has a few other interesting examples. Try them out!

File	Equation	Threshold
<code>goursat.df3</code>	$v(x,y,z) = x^4 + y^4 + z^4 + a(x^2 + y^2 + z^2)^2 + b(x^2 + y^2 + z^2) + c$ Where, $a = -0.27$; $b = -0.5$; $c = -1.0$	0.18
<code>cassini.df3</code>	$v(x,y,z) = ((x - a)^2 + z^2)((x + a)^2 + z^2) - y^4$ Where, $a = 0.3$	0.2
<code>barth.df3</code>	Uggh	1-F1, and use 0.3416

5. Glowing Media

Another way to try and visualize 3D volumes is to give the density values a color map. Here we can let the density values emit their own color (e.g., the brain on fire image at the beginning of this lecture). This can give some interesting ways to look at data. The following example was blatantly copied off the web (<http://paulbourke.net/miscellaneous/povexamples/>). Try it on the Bretzel function!

```
#declare NX = 191;           // x-dimension
#declare NY = 191;           // y-dimension
#declare NZ = 191;           // z-dimension
#declare DIAG = <NX,NY,NZ>;

global_settings
{
    ambient_light <1,1,1>
    assumed_gamma 1
}
camera
```

```

{ location <0,-7/4*NY,2/3*NZ>
  up z
  right x // default: 4/3*x
  sky <0,0,1>
  look_at <0,0,0>
}
light_source
{ <2*NX,-NY,2*NZ>
  color rgb <1,1,1>
  media_interaction on
  media_attenuation on
  shadowless
}

//-----

#declare DENS = interior
{ media
  { intervals 100 // number of ray-intervals
    ratio 0.5
    samples 3,3 // max,min of samples per voxel
    method 3 // 1, 2 or 3 (3 requires samples 3,3)
    emission 3*<1,1,1>/100
    absorption <1,1,1>/1000
    scattering { 1, <0,0,0> }
    confidence 0.999 // default: 0.9
    variance 1/10000 // default: 1/128
    density

    { density_file df3 "./bretzel.df3"
      interpolate 1
      color_map // colour map
      { [0.002 rgb <0.0,0.0,0.0>] // 0.0 ~ 'black'
        [0.003 rgb <1.0,0.0,0.0>]
        [0.004 rgb <1.0,0.6,0.0>]
        [0.0045 rgb <0,0,0.0>]
        [1 rgb <0,0,0>] // 1.0 ~ 'black'
      }
    }
  }
}

box
{ <0,0,0>, <1,1,1>
  pigment { rgbt <0,0,0,1> }
  hollow
  interior { DENS }
  scale DIAG
  translate -DIAG/2
  //rotate <90,0,0>
  //rotate <0,0,360*clock> // rotation around z-axis
}

//-----

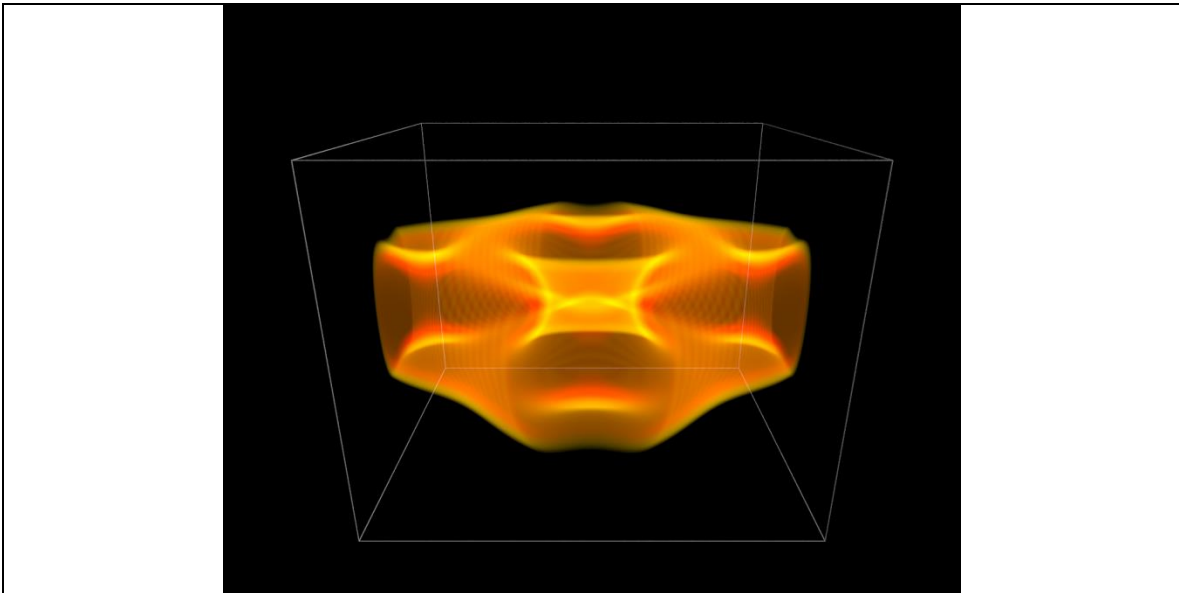
#declare RADIUS = 0.2;
#declare FRAME = texture

```

```

        { pigment { rgb <0.5,0.5,0.5> }
        }
union
{
  sphere { <00,00,00>, RADIUS texture { FRAME } }
  sphere { <NX,00,00>, RADIUS texture { FRAME } }
  sphere { <NX,NY,00>, RADIUS texture { FRAME } }
  sphere { <00,NY,00>, RADIUS texture { FRAME } }
  sphere { <00,00,NZ>, RADIUS texture { FRAME } }
  sphere { <NX,00,NZ>, RADIUS texture { FRAME } }
  sphere { <NX,NY,NZ>, RADIUS texture { FRAME } }
  sphere { <00,NY,NZ>, RADIUS texture { FRAME } }
  translate -DIAG/2
  //rotate <0,0,360*clock>           // rotation around z-axis
}
union
{
  cylinder { <00,00,00>, <NX,00,00>, RADIUS texture { FRAME } }
  cylinder { <NX,00,00>, <NX,NY,00>, RADIUS texture { FRAME } }
  cylinder { <NX,NY,00>, <00,NY,00>, RADIUS texture { FRAME } }
  cylinder { <00,NY,00>, <00,00,00>, RADIUS texture { FRAME } }
  cylinder { <00,00,00>, <00,00,NZ>, RADIUS texture { FRAME } }
  cylinder { <NX,00,00>, <NX,00,NZ>, RADIUS texture { FRAME } }
  cylinder { <NX,NY,00>, <NX,NY,NZ>, RADIUS texture { FRAME } }
  cylinder { <00,NY,00>, <00,NY,NZ>, RADIUS texture { FRAME } }
  cylinder { <00,00,NZ>, <NX,00,NZ>, RADIUS texture { FRAME } }
  cylinder { <NX,00,NZ>, <NX,NY,NZ>, RADIUS texture { FRAME } }
  cylinder { <NX,NY,NZ>, <00,NY,NZ>, RADIUS texture { FRAME } }
  cylinder { <00,NY,NZ>, <00,00,NZ>, RADIUS texture { FRAME } }
  translate -DIAG/2
  //rotate <0,0,360*clock>           // rotation around z-axis
}
}

```



6. Animations

Rendering individual snapshots for animations is quite simple in POV-Ray. As a quick example, let's consider simply making a POV-Ray script showing Jupiter's satellite Io. Name this script **planetrotate.pov**.

```
// cameras, lights, background
//-----//
camera { location <0,0.8,-7> look_at <0,0,0>}
light_source { <0,35,-65> color rgb <1,1,1>}
//-----//

// mirrored surface
//-----//
plane {
    <0,1,0>, -2.1
    pigment {color rgb <0,0,0> }
    finish {reflection 0.3}
}
//-----//

// Io
//-----//
#declare Io =
sphere {
    <0,0,0>, 1.0
    pigment {image_map {png "./05_Io_Surface.png" map_type 1
    interpolate 2}}
    finish {ambient 0.2 diffuse 1}
    scale 2.5
}
//-----//

// display and set the clock for animation
//-----//
object {Io
rotate <0,360*clock,0>
}
```

The main thing we did here that was any different was set the rotate function at the bottom to:

```
rotate <0,360*clock,0>
```


This will allow us to animate a rotation around the y-axis. Next open up new file and save it as a **.ini** file. For example, name it **rotateIo.ini**. Note that in **.ini** files a semi-colon (;) is used for comments.

```
; POV-Ray animation ini file
Antialias=Off
Antialias_Threshold=0.1
Antialias_Depth=2

Input_File_Name="planetrotate.pov"
Initial_Frame=1
Final_Frame=60
Initial_Clock=0
Final_Clock=1

Cyclic_Animation=on
Pause_when_Done=off
```

Now just click run on this **.ini** file and it will start cranking out the individual files!

7. Some links to useful POV-Ray examples

http://www.antoniosiber.org/intro_to_scivis_1/comploids_scivis_siber_presentation.pdf

http://www.f-lohmueller.de/pov_tut/all_shapes/shapes652e.htm

Animation tutorial: http://www.f-lohmueller.de/pov_tut/animate/pov_anie.htm