This page contains details on the SLAC code used to generate the donut images. Important elements of the code are discussed.

The distortions in optical images are expressed in terms of various optical aberrations (such as defocus, tilt, power, astigmatism, coma, etc,). These are all optical aberrations that are a result of imperfect lens systems when an image is projected onto the focal plane. The SLAC code was written to allow you to generate the type of images you may see on wavefront sensors, which will allow you to measure these effects and if possible, correct for them by adjusting the optics.

The package comes with a C program **Zernike.cc** which contains polynomials that represent the various optical aberrations. In the code, the following list defines the Zernike polynomials (the first 37 terms):

```
std::string descriptions[37] =      {"Piston (Bias)         1",
                                    "Tilt X              4^(1/2) (p) * COS (A)",
                                    "Tilt Y              4^(1/2) (p) * SIN (A)",
                                    "Power (Defocus)      3^(1/2) (2p^2 - 1)",
                                    "Astigmatism Y        6^(1/2) (p^2) * SIN (2A)",
                                    "Astigmatism X        6^(1/2) (p^2) * COS (2A)",
                                    "Coma Y              8^(1/2) (3p^3 - 2p) * SIN (A)",
                                    "Coma X              8^(1/2) (3p^3 - 2p) * COS (A)",
                                    "Trefoil Y           8^(1/2) (p^3) * SIN (3A)",
                                    "Trefoil X           8^(1/2) (p^3) * COS (3A)",
                                    "Primary Spherical     5^(1/2) (6p^4 - 6p^2 + 1)",
                                    "Secondary Astig X     10^(1/2) (4p^4 - 3p^2) * COS (2A)",
                                    "Secondary Astig Y     10^(1/2) (4p^4 - 3p^2) * SIN (2A)",
                                    "TetraFoil X          10^(1/2) (p^4) * COS (4A)",
                                    "TetraFoil Y          10^(1/2) (p^4) * SIN (4A)",
                                    "Secondary Coma X      12^(1/2) (10p^5 - 12p^3 + 3p) * COS (A)",
                                    "Secondary Coma Y      12^(1/2) (10p^5 - 12p^3 + 3p) * SIN (A)",
                                    "Secondary Trefoil X   12^(1/2) (5p^5 - 4p^3) * COS (3A)",
                                    "Secondary Trefoil Y   12^(1/2) (5p^5 - 4p^3) * SIN (3A)",
                                    "Pentafoil X          12^(1/2) (p^5) * COS (5A)",
                                    "Pentafoil Y          12^(1/2) (p^5) * SIN (5A)",
                                    "Secondary Spherical   7^(1/2) (20p^6 - 30p^4 + 12p^2 - 1)",
                                    "Tertiary Astig Y      14^(1/2) (15p^6 - 20p^4 + 6p^2) * SIN (2A)",
                                    "Tertiary Astig X      14^(1/2) (15p^6 - 20p^4 + 6p^2) * COS (2A)",
                                    "Secondary Tetrafoil Y  14^(1/2) (6p^6 - 5p^4) * SIN (4A)",
                                    "Secondary Tetrafoil X  14^(1/2) (6p^6 - 5p^4) * COS (4A)",
                                    "Sextafoil Y          14^(1/2) (p^6) * SIN (6A)",
                                    "Sextafoil X          14^(1/2) (p^6) * COS (6A)",
                                    "Tertiary Coma Y       16^(1/2) (35p^7 - 60p^5 + 30p^3 - 4p) * SIN (A)",
                                    "Tertiary Coma X       16^(1/2) (35p^7 - 60p^5 + 30p^3 - 4p) * COS (A)",
                                    "Tertiary Trefoil Y    16^(1/2) (21p^7 - 30p^5 + 10p^3) * SIN (3A)",
                                    "Tertiary Trefoil X    16^(1/2) (21p^7 - 30p^5 + 10p^3) * COS (3A)",
```

```
"Secondary Pentafoil Y  16^(1/2) (7p^7 - 6p^5) * SIN (5A)",
"Secondary Pentafoil X  16^(1/2) (7p^7 - 6p^5) * COS (5A)",
"Septafoil Y            16^(1/2) (p^7) * SIN (7A)",
"Septafoil X            16^(1/2) (p^7) * COS (7A)",
"Tertiary Spherical     9^(1/2) (70p^8 - 140p^6 + 90p^4 - 20p^2 + 1)"};
```

The code actually explains the convention being used for the coefficients of he Zernike polynomials:

```
// convention in Noll is that iZ odd is sin, and iZ even is cos
 // but Noll also has iZ starting from 1, but we start at 0, so it is reversed!
```

The code **DonutEngine.cc** sets up the parameters that are used in the calculation. It explains which telescopes can be simulated:

```
// physical parameters
 // iTelescope = 0 for Blanco + DECam
 //            = 1 for Blanco + MosaicII
 //            = 2 for LSST
 //            = 3 for Magellan MegaCam
 //            = 4 for Magellan IMACS F/2
 //            = 5 for DESI
```

For LSST, the following default parameters are used:

```
} else if (iT==2){
   _outerRadius = 1.10874;   // this the radius of the pupil as found in Zemax!
   _innerRadius = 0.61 * _outerRadius;   // 4/4/2016 look at Zemax gives 0.612 - use 0.61 for
LSST donuts
   _zLength = 2.734706;
   _lambdaz = _waveLength * _zLength;
   _fLength = 10.31007;
   _pixelSize = 10.0e-6;
```

In order to understand the process of donut creation, we can run the test,py script and look at the output. I set the printLevel to 5 in order to get more detailed feedback on what steps are taken during the creation of the donut. Here is the relevant section to test.py that calculates the first image:

```
# make donuts
z4 = 10.
z5 = 0.2
```

```
z6 = 0.2
z7 = -0.15
z8 = .05
z9 = 0.3
z10 = -.05
z11 = .2
inputDict =
{'writeToFits':True,'outputPrefix':'unittest.0001','iTelescope':5,'nZernikeTerms':37,'nbin':512,'nPix
els':64,'pixelOverSample':8,'scaleFactor':1.,'rzero':0.125, 'nEle':1.0e6, 'background':4000.,
'randomFlag':True, 'randomSeed':2314809,
'ZernikeArray':[0.,0.,0.,z4,z5,z6,z7,z8,z9,z10,z11],'xDECam':1.318,'yDECam':0.86,'printLevel':5}

m = makedonut(**inputDict)
donut1 = m.make()
```

What is done:

- An input array is created where the appropriate values are set.
- A make donut object is created: m = makedonut(**inputDict)
- The donut is made: donut1 = m.make().

The following output is generated when the donut is made:

```
iTelescope = 5
waveLength = 7e-07
nZernikeTerms = 37
nbin = 512
nPixels = 64
outputPrefix = unittest.0001
debugFlag = 0
printLevel = 5
gridCalcMode = 1
pixelOverSample = 8
scaleFactor = 1
inputPupilMask =
zemaxToDECamSignFlip = 1
calcRzeroDerivative = 0
ngridperPixel = 8
deltaX = 1.875e-06
pupilscale = 1.36523
Lu = 2.43557
Lf = 533333
nhalfPixels = 32
DonutEngine: Parameters (Zernikes scaled) are = 1e+06 0.125 4000 0 0 10 0.2 0.2 -0.15 0.05
0.3 -0.05 0.2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

DonutEngine: calcPupilMask x,y = 1.318 0.86
DonutEngine: calcPupilFunc
DonutEngine: calcOptics
DonutEngine: calcAtmos
DonutEngine: calcConv
DonutEngine: calcPixelate
DonutEngine: calcAll is done

The following functions are executed in this order during the donut creation. This information can be used to understand how the code works:

- DonutEngine: Parameters .... . This output is generated in "void DonutEngine::fillPar". The comments in the code state:
- ```
  // save input parameters, and also reevaluate State Machine based on values of par
  // and on values stored in _xDECam,_yDECam
  ```
-
- The output is generated by the following lines of code:
- ```
  if (_printLevel>=2){
    std::cout << "DonutEngine: Parameters (Zernikes scaled) are = " << _nEle << " " << _rzero << " " << _bkgd << " " ;
    for (int iZ=0;iZ<nZernikeSize;iZ++){
      std::cout << _ZernikeArr[iZ] << " ";
    }
    std::cout << std::endl;
  }
  ```
- which shows that when we generated the image, nEle = 1e+06, rzero = 0.125, and bkgr = 4000. These values are defined in inputDict. You also see that z2 = z3 = 0, and that z1 does not appear in ZernikeArr.
- DonutEngine: calcPupilMask x,y = 1.318 0.86. This output is generated in "void DonutEngine::calcPupilMask()". The code generates the pupil mask based on the parameters provided. The code refers to three different masks that are applied: the spiderMask, the annularMask, and the filtExchMask. Depending on the telescope to be studied, the masks are used to determine for each pixel of pupilMask if it is "true" or "false". The following code segment illustrates this:
- ```
  // combine annulus, spider and filter Exchanger
  if (spiderMask && annulusMask && filtExchMask){
    _pupilMask(i) = 1.0;
  } else {
    _pupilMask(i) = 0.0;
  }
  ```
- If pupilMask = true, the mask is clear. A comment in the code ("//x1true mean pupil==1 (ie. clear)") suggests this.
- DonutEngine: calcPupilFunc . This output is generated in "void DonutEngine::calcPupilFunc()". In this function, the Zernike terms are calculated for each pixel. See:
- ```
  // Zernike terms
  for (int iZ=0;iZ<nZernikeSize;iZ++){
    if (_ZernikeArr[iZ] != _last_ZernikeArr[iZ]){
  ```

```
      Matrix zernikeTemp =  _zernikeObject->_zernikeTerm[iZ+1]; // need [iZ+1] since ZernikeTerm includes the Piston term
      for (int i=0;i<_nbin*_nbin;i++){
   _pupilWaveZernike(i) = _pupilWaveZernike(i) +  ( _ZernikeArr[iZ] - _last_ZernikeArr[iZ] ) * zernikeTemp(i);
      }
    }
  }
```

- The function is calculated in the following code segment:
- ```
  // calculate the pupilFunc(tion) from the pupilWave(front)
    Complex I = Complex(0.0,1.0);
    Complex twopiI = Complex(0.0,2.0*_M_PI);
    for (int i=0;i<_nbin*_nbin;i++){
      if (_pupilMask(i)==0.0){
        _pupilFunc(i) = 0.0;
        _pupilFuncStar(i) = 0.0;
      } else {
        _pupilFunc(i) = _pupilMask(i) * exp(twopiI  *_pupilWaveZernike(i));   // no lambda here, so units are in waveLength
        _pupilFuncStar(i) = conj(_pupilFunc(i));
      }
    }
  ```
- This code shows that
  - _pupilFunc(i) = 0.0 if _pupilMask(i)==0.0.
  - If _pupilMask(i) is not equal to 0: _pupilFunc(i) = _pupilMask(i) * exp(twopiI *_pupilWaveZernike(i)). Note:
    - twopiI is a complex number: Complex twopiI = Complex(0.0,2.0*_M_PI).
    - the value of _pupilFunc(i) is complex.
- DonutEngine: calcOptics. This output is generated in "void DonutEngine::calcOptics()". This function calculates the PSF (point spread function) from the pupil function obtained in the previous step. The results of the PSF calculation (due to optics) are stored in ftsOptics: _ftsOptics = _fftOutputArray. Details about the PSF can be found for example in http://web.ipac.caltech.edu/staff/fmasci/home/astro_refs/PSFtheory.pdf. Here are some specific comments made in those notes:
  - " “Point Spread Functions” describe the two-dimensional distribution of light in the telescope focal plane for astronomical point sources. Modern optical designers put a great deal of effort into reducing the size of the PSF for large telescopes."
  - " The PSF for a perfect optical system, based on circular elements, would be an “Airy Pattern,” which is derived from Fraunhofer diffraction theory (scalar approximation applied to plane waves)."
- DonutEngine: calcAtmos. This output is generated in "void DonutEngine::calcAtmos()". This function is similar to the previous function, calcOptics, except that it calculates the PSF due to the atmosphere. The result of the PSF calculation (due to atmosphere) are stored in ftsAtmos: _ftsAtmos = _fftOutputArray.
- DonutEngine: calcConv. This output is generated in "void DonutEngine::calcConvolute()". In this step, the results of the calculations carried out in the previous steps are combined. The relevant section of code is
  ```

- ```
  // convolution (now doing it as F-1{F(Optics) F(Atmos) F(Pixels)
  MatrixC prodFts(_nbin,_nbin);
  for (int i=0;i<_nbin*_nbin;i++){
    prodFts(i) = _ftsOptics(i) * _ftsAtmos(i) * _ftsPixel(i);
  }
  ```

- Note:
  - ftsPixel is calculated in DonutEngine::calcFTPixel.
  - ftsOptics is calculated in DonutEngine::calcOptics.
  - ftsAtmos is calculated in DonutEngine::calcAtmos
- DonutEngine: calcPixelate. This output is generated in "void DonutEngine::calcPixelate()". In this function, the Fourier grid is converted to pixels. The details of the conversion are determined by ngridperPixel which is the number of grid points per pixel. Here is a part of the code of this function:
- ```
    int index(0);
    int stride = (int) _ngridperPixel;
    int pixIndex(0);
    Real gridNorm = _ngridperPixel*_ngridperPixel;
    for (int j=0;j<_nbin;j=j+stride){
      for (int i=0;i<_nbin;i=i+stride){

    _valPixelCenters(pixIndex) = _convOpticsAtmosPixel(index) * gridNorm;

    index = index + stride;
    pixIndex++;

     }
  ```
- DonutEngine: calcAll. This output is generated in "void DonutEngine::calcAll(Real* par)". The actual image is calculated in this function:
- ```
  // calculate image
  for (int i=0;i<_nPixels*_nPixels;i++){
    _calcImage(i) = _nEle * _valPixelCenters(i) + _bkgd;
  }
  ```

- Note: valPixelCenters(i) must have been evaluated at the location of the image sensors. In this step, the image is scaled by nEle and bkgd is added.

Note: many of the previous functions use various Fourier transforms. The relevant

```
_fft2PlanC =  new fftw2dctc(_fftInputArray,_fftOutputArray,-1);
_ifft2PlanC = new fftw2dctc(_ifftInputArray,_ifftOutputArray,1);
_fft2rtcPlanC = new fftw2drtc(_fftrtcInputArray,_fftrtcTempArray,_fftrtcOutputArray);
```

fftw2dctc and fftw2drtc are defined in FFTWClass.h (see: fftw2dctc(MatrixC& in, MatrixC& out, int sign0) and fftw2drtc(Matrix& in0, MatrixC& temp0, MatrixC& out0))
```