Internal CrystalScatter program documentation

Michael Wagener, JCNS-1

January 27, 2025

Contents

Table 1: Document revision history

Date	**Short description**
April 2023	First edit phase.
June 2023	Updates.
August 2023	Some updates and insert Python code
July 2024	Update the new file structure.

All sources saved at https://github.com/neutron-simlab/CrystalScatter

1 Used source files

Here are some short informations of what is in each source file. From the history the executable name is *sas_scatter2* but the github name is set to CrystalScatter. This program has a graphical user interface and can calculate one image at a button click. If more images must be calculated, the background process *sas_scatter2Cons* will be used.

sas_scatter2	GUI	Main executable for the graphical usr interface.
sas_scatter2Cons	Cons	Background executable, all inputs are provided via parameter flags, values and configuration files.
sas_scatter2.pro	GUI	Qt project file for the GUI version.
sas_scatter2Cons.pro	Cons	Qt project file for the console version.
sc_main.cpp	GUI	The main() function of the main executable.
sc_mainCons.cpp	Cons	The main() function of the background process.
sc_globalConfig.h	GUI+Cons	Definitions for some functionallities.
sc_maingui.cpp / .h / .ui	GUI	All functions of the gui.
sc_calcgui.cpp / .h	GUI	Interface from the gui to the calculation class.
sc_calcCons.cpp / .h	Cons	Interface to the calculation class.
sc_calc_generic.cpp / .h	GUI+Cons	Interface class into the calculations.
sc_calc_generic_cpu.cpp	GUI+Cons	Class for the calculations, routines running on CPU.
sc_calc_generic_gpu.cu / .h	GUI+Cons	Class for the calculations, routines running on both CPU and GPU.
sc_gpu_generic.h	GUI+Cons	Calculations of all combinations of the comboboxes not optimized in other routines.
sc_gpu_*.h	GUI+Cons	Optimized calculations for special combinations of the comboboxes.
sc_lib_formfq_part*.h	GUI+Cons	Optimized formfq() functions for special particle types.
sc_lib_formpq_part*.h	GUI+Cons	Optimized formpq() functions for special particle types.
sc_math.h	GUI+Cons	Definition of some constants and the Double3 class for the x,y,z values.
sc_postproc.cpp / .h	GUI+Cons	Image Postprocessing: calculate the (r,phi)-Image, the FFT and iFFT.
sc_simplexfit2d.cpp / .h	GUI+Cons	All functions for the simplex 2d fit algorithm.
sc_readdata.cpp / .h	GUI+Cons	Functions to read different types of data files.
widimage.cpp / .h / .ui	GUI	Class for each image. The images are displayed as separated windows.
dlg*.cpp / .h / .ui	GUI	Dialog functions for some gui features.
myguiparam.cpp / .h	GUI	Helperclass for the parameters.
debughandler.h	GUI	Helperfunction for debugging

The comments in the form "//Z=number" are the linenumbers of the corresponding line in the pascal source file. The date of this source file varied.

2 The calculation process

In this chapter I'll describe the process inside the program if the user clicks on the *Calculate* Button. This is done in principle also for the simplex 2d fit and in the background process. But here I'll explain only the steps for the main gui.

In the code there are some #ifdef areas. The meaning and usage is explained in the file sc_globalConfig.h (unfortunately at the moment in german). In the following explanations I leave these sections out of the descriptions to make it not too complicate to understand.

The user has to fill out all parameters in the gui. If you change the comboboxes some of the other parameters might disappear if they are not needed for this calculation. If you are fine click the *Calculate* button. This will start the following routines:

- SC_MainGUI::on_butCalc_clicked()
  Save all parameters to a temporary file. This is done for one reason: if the calculation crashes, you can recover exactly these settings and try some different ones. Then it calls the next function.

- SC_MainGUI::local_butCalc_clicked()
  This procedure is called every time an image must be calculated from the gui (in simplex 2d fit, fft and some test cases). It first calls the preparation (prepareCalculation), then initialize some internal variables for the resulting image size and then starts the calculation (done in a thread!). After the thread is finished some cleanup is done (finishCalculation) and the image will be displayed (addImage).

- SC_MainGUI::prepareCalculation(bool progbar)
  This disables some start calculation buttons and enables the progress bar and the abort button. Then all input fields are copied into the calculation class (fillDataFromInputs). After this the preparation inside the calculation class is called (calcGui::prepareCalculation).

- SC_MainGUI::fillDataFromInputs()
  In the program I use (historically) some hashes/vectors for the variables:

  - SC_CalcGUI::inpValues<QString,double> used for single value data
  - SC_CalcGUI::inpVectors<QString,Double3> used for (x,y,z) tupel data
  - SC_CalcGUI::inpSingleValueVectors<QString,double> are the single values from the tupels above
  - calcHelper::params<QString,paramHelper*> this helper structure contains the values to have no gui access during the calculations and a pointer to the gui element. This hash is filled once in the constructor of the SC_CalcGUI class.
  - myGuiParam::allParams<myGuiParam*> *(vector)* this helper structure contains all parameter informations for each input widget provided in the main calculation tab in the gui.

  In this function the values are copied from the gui input fields into the hashes. All variables are accessed by thier name. The names comes from the first pascal program I've got and represents the variable names there.

- SC_CalcGUI::prepareCalculation(bool fromFit)
  In the normal calculation this calls SC_Calc_GENERIC::prepareData(_dataGetter dg). From the fit it call the same routine but use another parameter function.

- SC_Calc_GENERIC::prepareData(_dataGetter dg)
  In this procedure all values are copied from the gui elements into the calcHelper::params hash and then put them into the calculation class (into internal variables there). The function provided by the parameter gets two arguments: one string as the name of the parameter and a special value structure to get back all types of values (double, int, bool).

- myCalcThread::run()
  This thread function calls the calculation procedure (SC_CalcGUI::doCalculation). This is done inside a thread to prevent the gui to freeze.

- SC_CalcGUI::doCalculation(int numThreads)
  This calls the real calculation procedure. This call is from the Thread to the Generic-Class.

- SC_Calc_GENERIC::doCalculation(int numThreads)
  This second intermediate function call separates the GPU part from the rest.

- SasCalc_GENERIC_calculation::doCalculation(int numThreads)
  In this procedure first a preparation function is called and then the subthreads for the pixel calculations are startet on the CPU (parameter numThreads>0) or the GPU (numThreads==0).

- SasCalc_GENERIC_calculation::prepareCalculation()
  This procedure runs allways on the CPU because here only the factors are calculated. This is very fast.

- SasCalc_GENERIC_calculation::doThreadCalculation(void *arg) *cpu*
  This is called in the subthread to convert the thread parameter to the ihex pixel index. Then loop over the other dimension of the image and call the doIntCalc_GENERIC_F() routine for calculation.

- SasCalc_GENERIC_calculation::doIntCalc_GENERIC_F(CALC, ihex, i) *cpu,gpu*
  The values ihex and i are the indices in the image. From these indices and some global parameters the current qx, qy, qz values are calculated. Then the real calculation function (doIntCalc_GENERIC_q_xyz) is called. The CALC variable is the reference to the static SasCalc_GENERIC_calculation class to get access to all global variables.

- SasCalc_GENERIC_calculation::doIntCalc_GENERIC_q_xyz(...) *cpu,gpu*
  This function calculates one pixel value and returns it. The calling function will store this value into the destination array.

- The following routines are slightly modified for the simplex 2d fit algorithm:
  SC_Calc_GENERIC::doFitCalculation(int numThreads)
  SasCalc_GENERIC_calculation::doThreadFitCalculation(void *arg)
  SasCalc_GENERIC_calculation::doIntFitCalc_GENERIC_F(CALC, ihex, i)

- SC_MainGUI::finishCalculation(bool showtime)
  This function is called after all pixel values are calculated and reenables the calculation button and disabled the progress bar and the abort button. Then it saves the calculation time used in the SC_CalcGUI::inpValues hash.

- int SC_Calc_GENERIC_calculation::minX()
  int SC_Calc_GENERIC_calculation::maxX()
  int SC_Calc_GENERIC_calculation::minY()
  int SC_Calc_GENERIC_calculation::maxY()
  These functions returns the dimension if the calculated image. For easier access from the gui, the routines are defined in SC_CalcGUI and calls internally the above functions in the calculation class.

- double *SC_Calc_GENERIC_calculation::data()
  With this function you get a pointer to the image data. The easier acces is the same as above.

3 Interfaces

In this chapter the three interfaces between the gui and the calculation procedures are described:

- variable settings - each variable can be set by their own call
- group settings - the groups shown in the gui can be set each with one call
- model settings - here we develop routine call sequences for specialized models

In this screenshot all possible parameters are shown. The parameters with grey lables are normally hidden and are not used for calculations in this selection of the comboboxes.

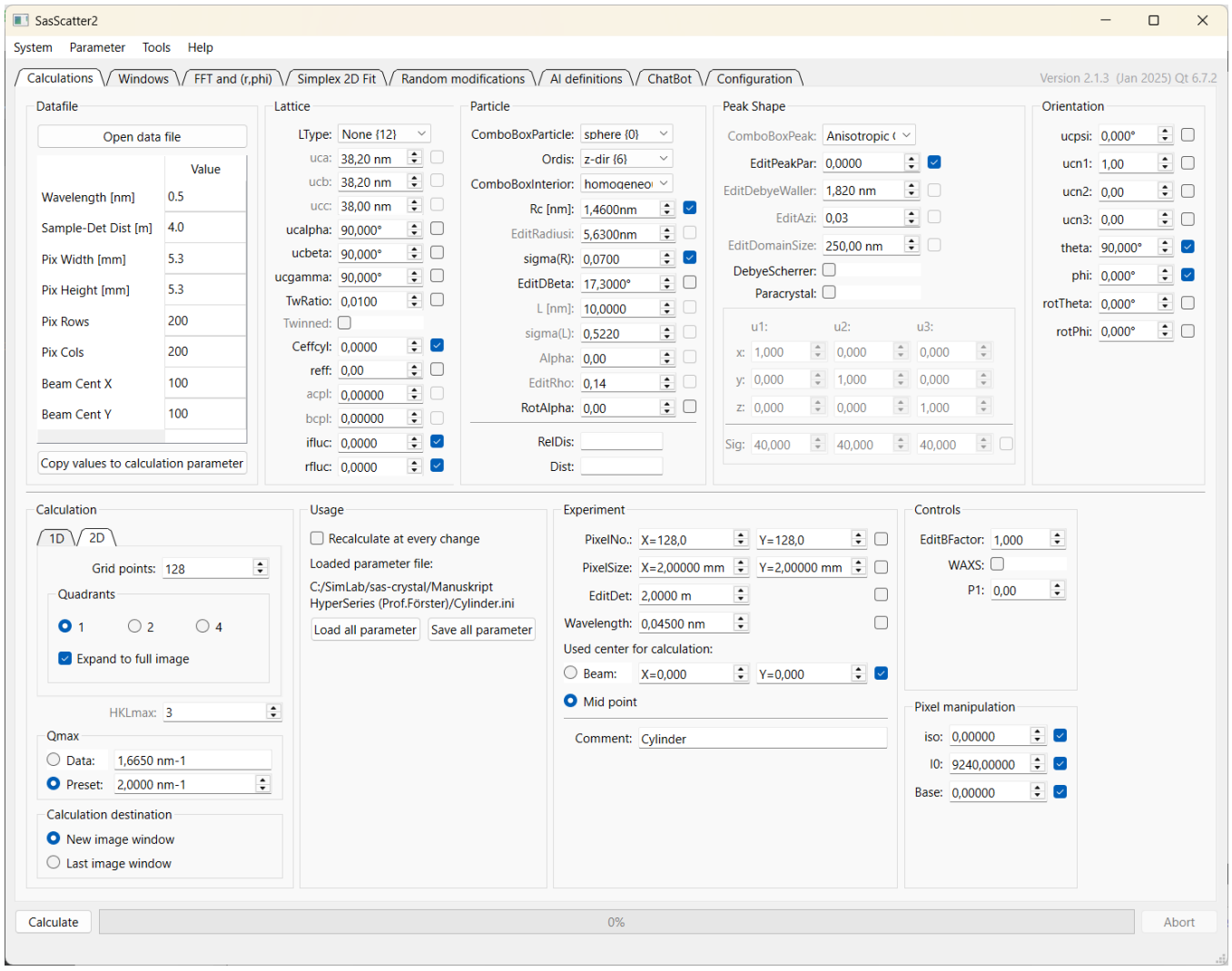


Figure 1: Calculation parameter grouping with no hidden values.

3.1 Variable settings - individual calls for each variable

There is one call to each variable. Some variables are converted directly to the correct internal range. There is no range checking done. This must be provided by the gui or the caller.

Lattice params

- setLType(int ltype) sets the lattice type from the combobox, ltype has a range from 0 to 22.
- setUCA(double uca) dimension a of the unit cell in nm.
- setUCB(double ucb) dimension b of the unit cell in nm.
- setUCC(double ucc) dimension c of the unit cell in nm.
- setUCalpha(double ucalpha) rotation angle alpha in degrees.
- setUCbeta(double ucbeta) rotation angle beta in degrees.

- setUCgamma(double ucgamma) rotation angle gamma in degrees.
- setCeffF(double editceff)
- setCheckBoxTwinned(bool twinned)
- setCeffCyl(double ceffcyl)
- setReff(double reff)
- setAcpl(double acpl)
- setBcpl(double bcpl)
- setIFluc(double ifluc)
- setRFluc(double rfluc)

Particle params

- setComboBoxParticle(int cbparticle) sets the particle type and has a range from 0 to 10.
- setOrdis(int cbordis) sets the ordis value and has a range from 0 to 13.
- setComboBoxInterior(int cbinterior) this has a range from 0 to 4.
- setRadiusF(double radius) radius in nm.
- setRadiusI(double radiusi) second radius in nm.
- setSigmaF(double sigma)
- setDBetaF(double dbeta) in degrees.
- setLength(double length) in nm.
- setSigmaL(double sigmal)
- setAlphash(double alpha)
- setRho(double rho)

PeakShape params

- setComboBoxPeak(int cbpeak) this has a range from 0 to 7.
- setPeakPar(double peakpar)
- setDisplacement(double debyewaller) displacement=debyewaller; dwfactor=debyewaller*debyewaller/3.0;
- setAzi(double azi)
- setDomainsize(double domainsize)
- setRadioButtonDebyeScherrer(bool debyescherrer)
- setRadioButtonPara(bool para)

PeakShapeMatrix params
The class *Double3* contains x, y and z values.

- setAx1(Double3) sets ax1, ay1, az1.
- setAx2(Double3) sets ax2, ay2, az2.
- setAx3(Double3) sets ax3, ay3, az3.
- setSigXYZ(Double3) sets sigx, sigy, sigz.

Orientation params

- setUCpsi(double ucpsi)
- setUCn1(double ucn1)
- setUCn2(double ucn2)
- setUCn3(double ucn3)
- setPolTheta(double theta)
- setPolPhi(double phi)
- setRotTheta(double rottheta)

- setRotPhi(double rotphi)

  Calculation params

- setGridPoints(int gridp) defines the dimension (one half of each axis) of the output image
- setHKLmax(int hklmax) defines the deep of recursions
- setQMax(double qmax)
- setRadQ1(bool radq1)
  setRadQ2(bool radq2)
  setRadQ4(bool radq4) call one of these with true if this quadrant should be calculated. *There are three calls because they are callbacks of the radiobuttons in the gui.*
- setExpandImage(bool expand) set this to true to expand the calculated image (Q1 / Q2) to all quadrants.

  Experiment params

- setpixnox(double pixelnox) detector pixel count horizontally.
- setpixnoy(double pixelnoy) detector pixel count vertically.
- setpixx(double pixelsizex) size of one detector pixel horizontally in m.
- setpixy(double pixelsizey) size of one detector pixel vertically in m.
- setdet(double detdist) detector distance in m.
- setwave(double wavelength) wavelength in nm.
- setBeamStop(double beamcx, double beamcy) sets the x and y coordinates of the beamstop in pixel indices.

  Control params

- setBFactorF(double bfactor)
- setCheckBoxWAXS(bool waxs)
- setP1(double p1)

  PixelManipulation params

- setIso(double iso)
- setIZero(double i0)
- setBase(double base)

3.2 Group settings - combined calls for each group of variables

There are calls for almost every group of parameters: Lattice, Particle, Peak Shape with the matrix, Orientation, Calculation with Qmax and Quadrants, Experiment, Controls, Pixel manipulation. The parameter names are the same as above.

- **setLatticeParams**(int ltype, double uca, double ucb, double ucc, double ucalpha, double ucbeta, double ucgamma, double editceff, bool twinned, double ceffcyl, double reff, double acpl, double bcpl, double ifluc, double rfluc)
- **setParticleParams**(int cbparticle, int cbordis, int cbinterior, double radius, double radiusi, double sigma, double dbeta, double length, double sigmal, double alpha, double rho)
- **setPeakShapeParams**(int cbpeak, double peakpar, double debyewaller, double azi, double domainsize, bool debyescherrer, bool para)
- **setPeakShapeMatrix**(double ax1, double ax2, double ax3, double ay1, double ay2, double ay3, double az1, double az2, double az3, double sigx, double sigy, double sigz)
- **setOrientationParams**(double ucpsi, double ucn1, double ucn2, double ucn3, double theta, double phi, double rottheta, double rotphi)
- **setCalculationParams**(int gridp, int hklmax, double qmax, bool radq1, bool radq2, bool radq4, bool expand)

- **setExperimentParams**(double pixelnox, double pixelnoy, double pixelsizex, double pixelsizey, double detdist, double wavelength, double beamcx, double beamcy)
- **setControlParams**(double bfactor, bool waxs, double p1)
- **setPixelManipulationParams**(double iso, double i0, double base)

3.3 Model settings - examples of usage

This part is still under development. If you want to extract sone code pieces to write your own implementation, please ask for the correct statements.

3.3.1 Spheres - C++ calls

If you want a simple sphere model for the sample, you can extract some code and implement your own routines (see below) or you can call some of the above routines and get the resultant image from this package. Here is a list of routines to be called with some parameter settings for the result. Bold parameter names denotes the values you have to specify, here are the values for the image shown below.

- group parameter calls
  - setLatticeParams(ltype=12, uca=1, ucb=1, ucc=1, ucalpha=0, ucbeta=0, ucgamma=0, editceff=0.01, twinned=false, ceffcyl=0, reff=0, acpl=0, bcpl=0, ifluc=0, rfluc=0)
  - setParticleParams(cbparticle=0, cbordis=7, cbinterior=0, **radius**=16.7, radiusi=0, **sigma**=0.0639, dbeta=0, length=0, sigmal=0, alpha=0, rho=0)
  - setPeakShapeParams(cbpeak=7, peakpar=0, debyewaller=0, azi=0, domainsize=0, debyescherrer=false, bool para=false)
  - setPeakShapeMatrix(ax1=1, ax2=0, ax3=0, ay1=0, ay2=1, ay3=0, az1=0, az2=0, az3=1, sigx=40, sigy=40, sigz=40)
  - setOrientationParams(ucpsi=0, ucn1=1, ucn2=0, ucn3=0, theta=0, phi=0, rottheta=0, rotphi=0)
  - setCalculationParams(**gridp**=128, hklmax=1, **qmax**=2, radq1=false, radq2=false, radq4=true, expand=false)
  - setExperimentParams(**pixelnox**=128, **pixelnoy**=128, **pixelsizex**=0.0025, **pixelsizey**=0.0025, **detdist**=10, **wavelength**=0.154, **beamcx**=0, **beamcy**=0)
  - setControlParams(bfactor=0.01, waxs=false, p1=0)
  - setPixelManipulationParams(iso=0, i0=1, base=0)
- single value calls
  - setLType(ltype=12)
  - setComboBoxParticle(cbparticle=0)
  - setOrdis(cbordis=7)
  - setComboBoxInterior(cbinterior=0)
  - setRadiusF(**radius**=16.7)
  - setSigmaF(**sigma**=0.0639)
  - setRadioButtonDebyeScherrer(debyescherrer=false)
  - setRadioButtonPara(para=false)
  - setAx1(Double3(1,0,0))
  - setAx2(Double3(0,1,0))
  - setAx3(Double3(0,0,1))
  - setSigXYZ(Double3(40,40,40))
  - setUCpsi(ucpsi=0)
  - setUCn1(ucn1=1)
  - setUCn2(ucn2=0)
  - setUCn3(ucn3=0)

- setPolTheta(theta=0)
- setPolPhi(phi=0)
- setRotTheta(rottheta=0)
- setRotPhi(rotphi=0)
- setGridPoints(**gridp**=128)
- setQMax(**qmax**=2)
- setRadQ4(radq4=true)
- setExpandImage(expand=false)
- setpixnox(**pixelnox**=128)
- setpixnoy(**pixelnoy**=128)
- setpixx(**pixelsizex**=0.0025)
- setpixy(**pixelsizey**=0.0025)
- setdet(**detdist**=10)
- setwave(**wavelength**=0.154)
- setBeamStop(**beamcx**=0, **beamcy**=0)
- setBFactorF(bfactor=0.01)
- setCheckBoxWAXS(waxs=false)
- setP1(p1=0)
- setIso(iso=0)
- setIZero(i0=1)
- setBase(base=0)

Then a call to doCalculation() starts the calculations and you can retrieve the result with the data() pointer. With the above settings you will get this image:

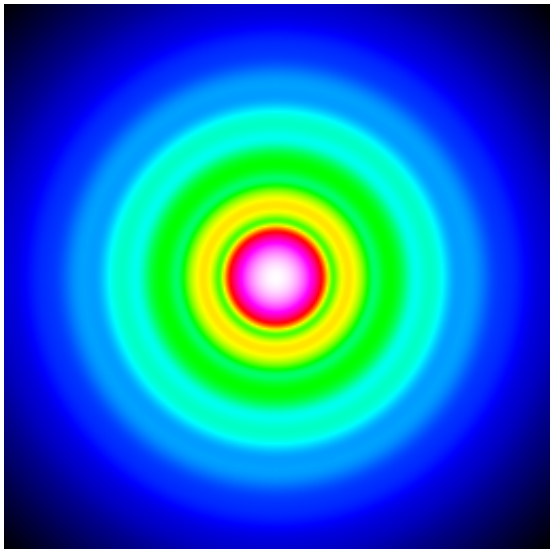


Figure 2: Example sphere image.

If you want to extract the source code for the same result, then you can use the following parts. The values provided as inputs are written in capital letters and the code presented here is cleaned up a little bit.

- First you have to calculate the coefficients. This is done in sc_calc_generic_cpu.cu in the routine SasCalc_GENERIC_calculation::coefficients() and is called only once for each image:
  Inputs: RADIUS=16.7, SIGMA=0.0639 from the user/experiment
  Inputs: dim=3=sphere, cs=0=homogeneous from the gui

```
zz = (1-sqr(SIGMA))/sqr(SIGMA);
xrz = RADIUS/(2.0*(zz+1));
xr2z = xrz*xrz;
xrn[0] = 1;
z12v[0] = 1;
fkv[0] = 1;
gam3[0] = sqrt(M_PI)/2.0;
n4 = nmax;  // nmax=120, the Dimension of carr4p[]
/* ** isotropic case for spheres ** */
if ( dim==3 )
{ if ( cs==0 )
  { for ( n=1; n<=nmax; n++ )
    { z12v[n] = z12v[n-1]*((z+1)-2+2*n)*((z+1)-1+2*n);
      fkv[n] = fkv[n-1]*n;
      gam3[n] = gam3[n-1]*(2*n+1)/2.0;
      xrn[n] = -xrn[n-1]*xr2z;
      carr4p[n] = 9*sqrt(M_PI)*pow(4.0,n)*z12v[n]*xrn[n]/
                  (2.0*(n+3)*(n+2)*(n+3/2.0)*gam3[n]*fkv[n]);
      if ( fabs(carr4p[n])<min ) { if ( n<n4 ) n4 = n; }
    } // for n
    limq4 = pow(fabs(carr4p[n4]),-1/(2.0*n4));
    return;
  } // if cs==0
} // if dim==3
```

  Outputs: limq4, carr4p[] used in the next routine.

- Then you have to calculate for each pixel the formfactor. This is done in sc_libs_formpq_partSphere.h in the routine SasCalc_GENERIC_calculation::formpq_partSphere():
  Inputs: RADIUS=16.7, SIGMA=0.0639 from the user/experiment
  Inputs: Q calculated from pixelindices, detectorsize, detectordistance and wavelength
  Inputs: limq4, carr4p[] from the coefficients() routine above
  Inputs: part=0=sphere, select the routine, cs=0=homogeneous from the gui

```
zr = (1-sqr(SIGMA))/(sqr(SIGMA));
if ( cs==0 )
{ if ( Q<0.4*limq4 )
  { double pqsum = 1.0;
    double oldpqsum = 0.0;
    double qq2 = 1.0;
    for ( int nser=1; nser<=100; nser++ )
    { qq2 = qq2*q*q;
      pqsum = pqsum+carr4p[nser]*qq2;
      double delser = fabs((pqsum-oldpqsum)/pqsum);
      if ( delser<0.0001 ) break;
      oldpqsum = pqsum;
    }
    return pqsum;
  }
  else
```

```
  { const double argq = Q*params.radius/(zr+1);
    double pqr;
    if ( zr*(zr-1)*(zr-2)*(zr-3)*(zr-4)*(zr-5) == 0 )
      pqr = 0.1;
    else
      pqr = (1/(2.0*zr*(zr-1)*(zr-2)*(zr-3)))*pow(argq,-4);
    const double pq1 = pqr*(1+cos((zr-3)*atan(2.0*argq)) /
                                          pow(1.0+4*argq*argq,(zr-3)/2.0));
    const double pq2 = (pqr/((zr-4)*argq))*sin((zr-4)*atan(2.0*argq)) /
                                           pow(1.0+4*argq*argq,(zr-4)/2.0);
    const double pq3 = (pqr/((zr-4)*(zr-5)*argq*argq))*(1-cos((zr-5) *
                            atan(2.0*argq))/pow(1.0+4*argq*argq,zr-5)/2.0);
    return 9.0*(pq1-2.0*pq2+pq3);
  }
} // if cs==0
```

Output: formpq_partSphere() return value

- Finally you have to calculate the pixel value. If lattice is set to none and ordis set to isotropic, the system use the optimized routine calc_partSphere_lattNone_ordisIsotropic() in sc_gpu_pSph_lNon_oIso.h:
  Inputs: formpq_partSphere() return value
  Inputs: base=0=*pixelvalue offset*, izero=1=*pixelvalue factor* from the gui

```
//pq = formpq()
pixelval = base + izero*pq;
```

Output: Pixelvalue at the given (radius,sigma,q) position.

- Around these calculations you have to program a two dimensional loop. This is done in sc_calc_generic_gpu.cu in the routine SasCalc_GENERIC_calculation::doCalculation() together with the preparation, optimized for GPU or multithreading. This is the first loop dimension calling another routine for the second loop dimension. This is done for the threads. From this last loop dimension the qx,qy,qz values are calculated and then the real calculation routine is called. *Inputs:* all values from the gui

```
zzmin = radq4 ? -gridp : 0;
zzmax = gridp;
iimin = radq1 ? 0 : -gridp;
iimax = gridp;
for ( int ihex=zzmin; ihex<zzmax; ihex++ )
{
    //doIntCalc_GENERIC( ihex++ );
    for ( int i=iimin; i<iimax; i++ )
    {
        //doIntCalc_GENERIC_F( *this, ihex, i );
        double qx, qy, qz;
        if ( CALC.useBeamStop )
        {   // Use the beam stop values
            double mdet = ihex*CALC.pixnoy/(2.0*CALC.zmax);
            double ndet = i   *CALC.pixnox/(2.0*CALC.zmax);
            double xdet = CALC.pixx_m * (ndet - CALC.beamX0);
            double ydet = CALC.pixy_m * (mdet - CALC.beamY0);
            double rdet = sqrt(xdet*xdet+ydet*ydet);
            double phidet = atan2(ydet,xdet);
            double thetadet = atan2(rdet,CALC.det);
            qx = 2*M_PI*cos(phidet)*sin(thetadet)/CALC.wave;
            qy = 2*M_PI*sin(phidet)*sin(thetadet)/CALC.wave;
            qz = 2*M_PI*(1-cos(thetadet))/CALC.wave;
        }
```

```
        else
        {   // Use midpoint
            qx = CALC.qmax * i    / (double)(CALC.zmax);
            qy = CALC.qmax * ihex / (double)(CALC.zmax);
            qz = 1e-20;
        }
        double pixval = doIntCalc_GENERIC_q_xyz(CALC, qx, qy, qz);
        CALC.setXYIntensity( ihex, i, pixval ); // set image pixel
    } // for i
} // for ihex
```

3.3.2 Spheres - Python code example

The following code shows the same calculations in Python. There is no multithreading optimization done.

- Include all used libraries. The platform is used to make some changes under windows because of the shorter exponent (see below).

```
import math
import numpy as np
import platform
import time
import matplotlib.pyplot as plt
```

- Definition of all input parameter and image dimensions.

```
# Global input parameters
radius = 16.7 # nm
sigma  = 0.0639
part   = 0 # Sphere (Particle combobox)
dim    = 3 # Sphere (selected from the part value)
cs     = 0 # homogeneous
qmax   = 2 # maximum Q value
base   = 0 # pixelvalue offset
izero  = 1 # pixelvalue factor

# Dimensions of the result image
gridpoints = 100
iimin = -gridpoints
iimax = +gridpoints
zzmin = -gridpoints
zzmax = +gridpoints
image = np.zeros( (gridpoints*2, gridpoints*2), dtype=np.longdouble )

# Other values
nmax   = 120   # maximum array length
carr4p = []    # intermediate array
min4   = 1E-40 # min for the calculation to find the limit
iswin  = False # flag set if the program runs under windows
```

- This routine calculates the coefficients and runs only once for each image. The *if dim==3:* and *if cs==0:* are optional in this case but needed if other coefficient calculations will be used. The numpy logdouble data type is used to have more precision and a larger exponent. But: *x86 machines provide hardware floating-point with 80-bit precision, and while most C compilers provide this as their long double type, MSVC (standard for Windows builds) makes long double identical to double (64 bits).*[1]

[1] Taken from https://numpy.org/doc/stable/user/basics.types.html#extended-precision

```
def coefficients():
    global limq4, carr4p, n4
    z    = np.longdouble( (1.-(sigma*sigma))/(sigma*sigma) )
    xrz  = np.longdouble( radius/(2.0*(z+1.)) )
    xr2z = np.longdouble( xrz*xrz )
    xrn  = np.longdouble(1)
    z12v = np.longdouble(1)
    fkv  = np.longdouble(1)
    gam3 = np.longdouble( math.sqrt(math.pi)/2.0 )
    n4   = nmax  # nmax=120, the Dimension of carr4p[]
    carr4p = [ 1. ]
    #/* ** isotropic case for spheres ** */
    #if dim==3: needed if other calculations are implemented
    #    if cs==0:
    spi9 = np.longdouble( 9. * math.sqrt(math.pi) )
    n=0
    for n in range(1, nmax+1):
        z12v = z12v * (z-1. + 2.*n) * (z + 2.*n)
        #  RuntimeWarning: overflow encountered in scalar multiply
        if np.isinf(z12v) or np.isnan(z12v):
            if n < n4:
                n4 = n-1
            break
            # On Windows the exponent can only be +-308,
            # On Linux the exponent can be up to +-4000
            # Both with the numpy.longdouble() !!
        fkv  = fkv*n
        gam3 = gam3*(2.*n+1)/2.0
        xrn  = -xrn*xr2z
        carr4p.append( spi9 * np.power(4.0,n,dtype=np.longdouble) *
                       (z12v*xrn) / (2. * (n+3.) * (n+2.) * (n+3./2.) *
                        gam3 * fkv) )
        if iswin and (z12v >= 1e307):
            # This will prevent the runtime warning shown above on Windows
            if n < n4:
                n4 = n
            break
        if math.fabs(carr4p[n])<min4:
            if n<n4:
                n4 = n
    limq4 = math.pow(math.fabs(carr4p[n4]),-1./(2.0*n4))
    return
    # if cs==0
    # if dim==3
```

- In the formpq routine the formfactor will be calculated. The *if part==0:* and *if cs==0:* are optional in this case but needed if other formfactor calculations will be used.

```
def formpq(i,ihex):
    global limq4, carr4p, n4
    # Here only use the midpoint
    qx = qmax * i    / gridpoints
    qy = qmax * ihex / gridpoints
    qz = 1e-20;
    q  = math.sqrt(qx*qx + qy*qy + qz*qz) + 1e-9
```

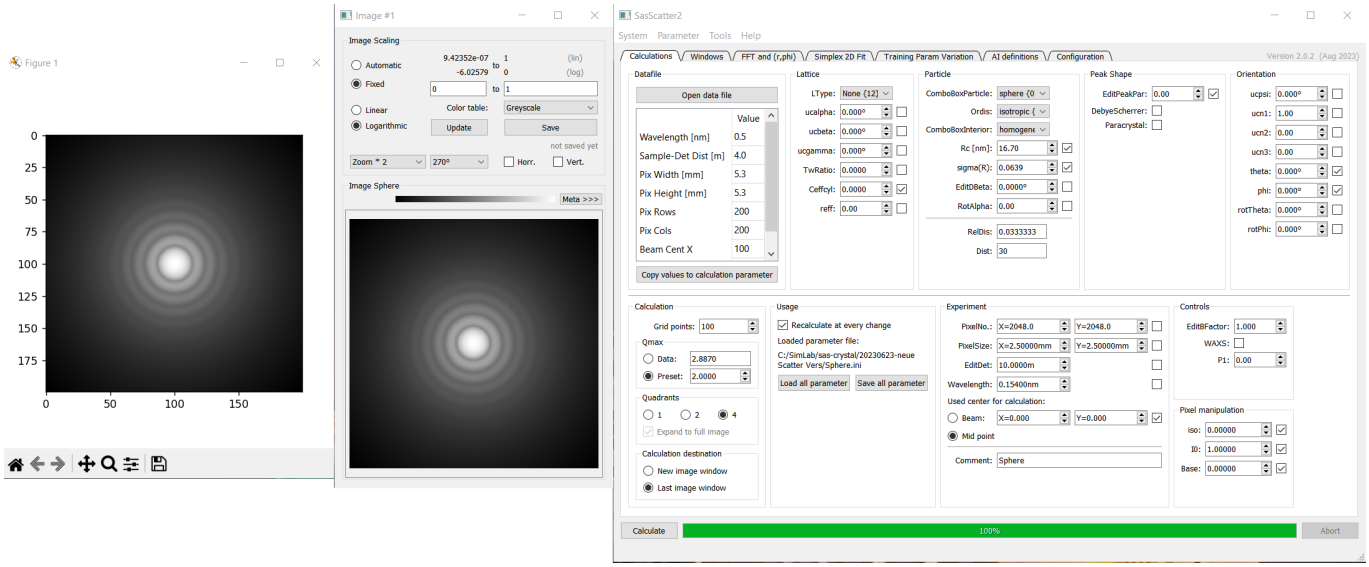
```
    zr = np.longdouble( (1.-(sigma*sigma))/(sigma*sigma) )
    #if part==0: needed if other formfactors are implemented
    #    if cs==0:
    if q < 0.4*limq4:
        pqsum = 1.
        oldpqsum = 0.
        qq2 = 1.
        for nser in range(1,n4):
            qq2 = qq2*q*q;
            pqsum = pqsum+carr4p[nser]*qq2;
            delser = math.fabs((pqsum-oldpqsum)/pqsum);
            if delser<0.0001:
                break;
            oldpqsum = pqsum;
        return pqsum;
    else:
        argq = q*radius/(zr+1)
        pqr = (1./(2.*zr*(zr-1)*(zr-2)*(zr-3)))*math.pow(argq,-4)
        pq1 = pqr*(1+math.cos((zr-3)*math.atan(2.*argq))/
              math.pow(1.+4*argq*argq,(zr-3)/2.))
        pq2 = (pqr/((zr-4)*argq))*math.sin((zr-4)*math.atan(2.*argq))/
              math.pow(1.+4*argq*argq,(zr-4)/2.)
        pq3 = (pqr/((zr-4)*(zr-5)*argq*argq))*(1-math.cos((zr-5)*
            math.atan(2.*argq))/math.pow(1.+4*argq*argq,zr-5)/2.)
        return 9.*(pq1-2*pq2+pq3)
    # if cs==0
    # if part==0
```

- This main program will calculate the coefficients, then loop over the image pixel indices, calculates the formfactor and the pixelvalue, then scale the image to the whole range of used values. The image is displayed with a logarithmic value scaling. There are some prints which can be removed. For both calculation part, the runtime will be measured with the simple time function. At the end the image will be displayed with matplotlib.

```
if __name__ == '__main__':
    if platform.system() == 'Windows':
        iswin = True

    print("Start coef")
    canf = time.time()
    coefficients()
    cend = time.time()

    print("Start loop")
    lanf = time.time()
    imgmin = 10000
    imgmax = 0
    for i in range(iimin,iimax):
        for ihex in range(zzmin,zzmax):
            pq = formpq(ihex,i)
            pix = math.log10(base + izero*pq)
            if pix < imgmin:
                imgmin = pix
            if pix > imgmax:
                imgmax = pix
```

```
            image[i-iimin,ihex-zzmin] = pix
    for i in range(iimin,iimax):
        for ihex in range(zzmin,zzmax):
            image[i-iimin,ihex-zzmin] = (image[i-iimin,ihex-zzmin] -
                                         imgmin) / (imgmax - imgmin)
    lend = time.time()

    print("times: coeff ", cend-canf)
    print("          loop ", lend-lanf)

    plt.imshow(image, cmap='gray', vmin=0, vmax=1)
    plt.show()
```

The above code generates this image (left) compared to the image from the C++ (middle). In the right the gui with all parameters is shown.



A Content of the git

If you get access to my git (*https://iffgit.fz-juelich.de/wagener/sas-crystal*), you get a lot of directories with all tests, helper programs and development steps. Here I write some informations to each directory[2] and some files found in the git:

Directory	**Description**
`20200526-unitcellrotations/`	Informations about unit cell rotations.
`20200724-fit2d/`	First steps towards the fit algorithm.
`20200818-ReadKWSImages/`	First reading of KWS data files.
`20200826-FCCundBCT/`	Differences between FCC and BCT calculations.
`20210322-RadiantundFFT/`	Discussions about the radial average images and the FFT.
`20210601-HDF5/`	First steps to read HDF5 data files.
`20210601-neuerAlgorithmusR-PHI/`	Updates of the radial average algorithm.
`20210616-NeueRoutinen/`	New routines of the basic pascal program.
`20211105-BCTneu/`	Tests with the new routines.
`20211206-FitRezept/`	Tests for the multi-input-file function of the fit algorithm.
`20211206-FitRezept/AutoFit/`	Some tests for the automatic fit routines.
`20220303-AI/`	Current steps for the new AI approach.
`dl_report/`	Helper program to analyze the output files from the Tensorflow run.
`docker-tests/`	Docker scripts for the experiments with Tensorflow.
`docker-tests/cedric/`	Helper script to generate 3 random vectors standing perpendicular to each other. This is used for the AI generation.
`doku/`	First documentations and default save path for the images.
`hdf_explorer_qt-master/`	Helper program found in a git repo to explore HDF5 files.
`pas2cpp/`	Helper program to convert Pascal code to C++ code.
`pas2cppNeu/`	New version of the helper program to convert Pascal code to C++ code.
`Pascal-Sourcecodes/`	Collection of all pascal files provides by Prof. Förster.
`sas/`	First GUI program without GPU usage.
`sas_imageing/`	Helper program (experimental) to erode or dilate an image for the AI algorithm.
`sas_report/`	Helper program to generate a report from some logfiles written during 2D fit.
`sas_scatter/`	Former version of the GUI / console program before restructuring the algorithms.
`sas_scatter2-doc/`	Source of this documentation.
`sas_scatter2/`	Software currently used and developed.
`.gitignore`	List of files / directories ignored by the git.
`HDF5Notizen-H5Einit.h`	After first cmake run of the HDF5 library, the file *H5Einit.h* has unexpected line breaks. This is a copy of the edited file.
`HDF5Notizen.txt`	Notes and descriptions to install the HDF5 library.
`History.txt`	Description of the first few weeks of development.
`*.sas_ai`	Saved files for the AI definitions.
`*.ini`	Saved files for all parameters.
`README.md`	Readme for the git web interface.
`Scatter_pascal.docx`	Some informations from the scatter program with explanations from Prof. Förster.
`Vergleich.docx`	Comparison of `Scatter_pascal.docx` and `crystal3d1.pas`.

[2] Some names are in german and some text files too, this is historical.

B Used keys in system settings

The following keys and groups are used to store informations beyond program run. They are stored in

- **Windows registry:** `\HKEY_CURRENT_USER\SOFTWARE\JCNS-1-SasCrystal\`*MasterKey*`\...`
- **Linux files:** `$HOME/.config/JCNS-1-SasCrystal/`*MasterKey*`.conf`

Goup	Key	Default	Description
Masterkey: *GUISettings*			
	LastMethod	-1	
	ConfigParamFile		
	GUIgeometry		bytearray with geometry informations for the main window
	ImgAutoPosit	false	ui-togAutoPosit
	OnlyNewWindow	false	ui-togOnlyNewWindow
	LimitRuntimeFlag	false	ui-togLimitRuntime
	LimitRuntimeValue	60	ui-inpLimitRuntime
	DefColTbl	1	ui-cbsDefaultColTbl
	GridPoints	64	ui-inpGridPoints
	Quadrant	2	ui-radQ1,2,4
	ExpandImg	true	ui-togExpandImage
	FFTLinInput	true	ui-radFFTLinInput
	FFTScaleInput	true	ui-togFFTScaleInput
	FFTScaleOutput	true	ui-togIFFTscaled
	FFTSwapOutput	true	ui-togIFFTSwap
	FFTsize	2	ui-cbsFFTsizeMan
	DispRphi	true	ui-togDispRphi
	FFToutput	0	0=Real, 1=Imag, 2=Abs, 3=Spec
Fit-Globals	BeamStop	0	ui-inpFitBStop
Fit-Globals	Border	0	ui-inpFitBorder
Fit-Globals	MaxIter	10	ui-inpFitMaxIter
Fit-Globals	StepSize	1.0	ui-inpFitStepSize
Fit-Globals	Tolerance	0.5	ui-inpFitTolerance
Fit-Globals	Repetitions	1	ui-inpFitRepetitions (not used)
Fit-Globals	UseMask	false	ui-togFitUseMask
Fit-*	*For all known calculation methods*		
Fit-*	*	0:0:0:0	*For all fittable parameters*
AI	LastSubDir	.	ui-inpSubDir
AI	Grayscale	false	ui-togGrayscale
AI	FileInputEna	false	ui-grpFileInput
AI	FileInputLast		ui-inpFileName
AI	FileClass	{M}	ui-inpFileClass
AI	GenerateIFFT	false	ui-togAIUseFFTOutput
AI	LinOut	true	ui-radAILinOutput
AI	ScaleOut	false	ui-togAIScaleOutput
AI	Cascading	true	ui-radLoopsCascade (not used, allways true)
AI	LastSaveFile	.	
AI	LastSaveBkgFile	.	
	LastImage	dataPath	
Masterkey: *Parameter*			
	LastParam	.	
	CompareF1	.	
	CompareF2	.	
	LastAutoFit	.	ui-inpInputfile
	AutoFitRunLogs	true	ui-togRunLogs

Goup	Key	Default	Description
	AutoFitLatexEna	true	ui-grpLatexEnabled
	AutoFitLatexImages	true	ui-grpImages
	AutoFitLatexOrgImgStep	true	ui-togOrgImgStep
	AutoFitLatexCalcImgStep	true	ui-togCalcStep
	AutoFitLatexResiduenStep	false	ui-togResiStep
	AutoFitLatexTrend	true	ui-togLatexTrend
	AutoFitLatexInput	true	ui-togLatexInput
	AutoFitLatexComment	false	ui-togLatexComments
	AutoFitDelFiles	false	ui-togDelFiles
	AutoFitEditor		