Electronic Supplementary Material (ESI) for Physical Chemistry Chemical Physics.

Electronic Supplementary Information

Coincident velocity map image reconstruction illustrated by the single-photon valence photoionisation of CF_3SF_5

Andras Bodi,[*,†] *Patrick Hemberger,*[†] *and Richard P. Tuckett*[‡]

[†]Laboratory for Synchrotron Radiation and Femtochemistry, Paul Scherrer Institute, 5232 Villigen, Switzerland

[‡]School of Chemistry, University of Birmingham, Edgbaston, Birmingham, B15 2TT, U.K.

[*] To whom correspondence should be addressed, e-mail: andras.boedi@psi.ch.

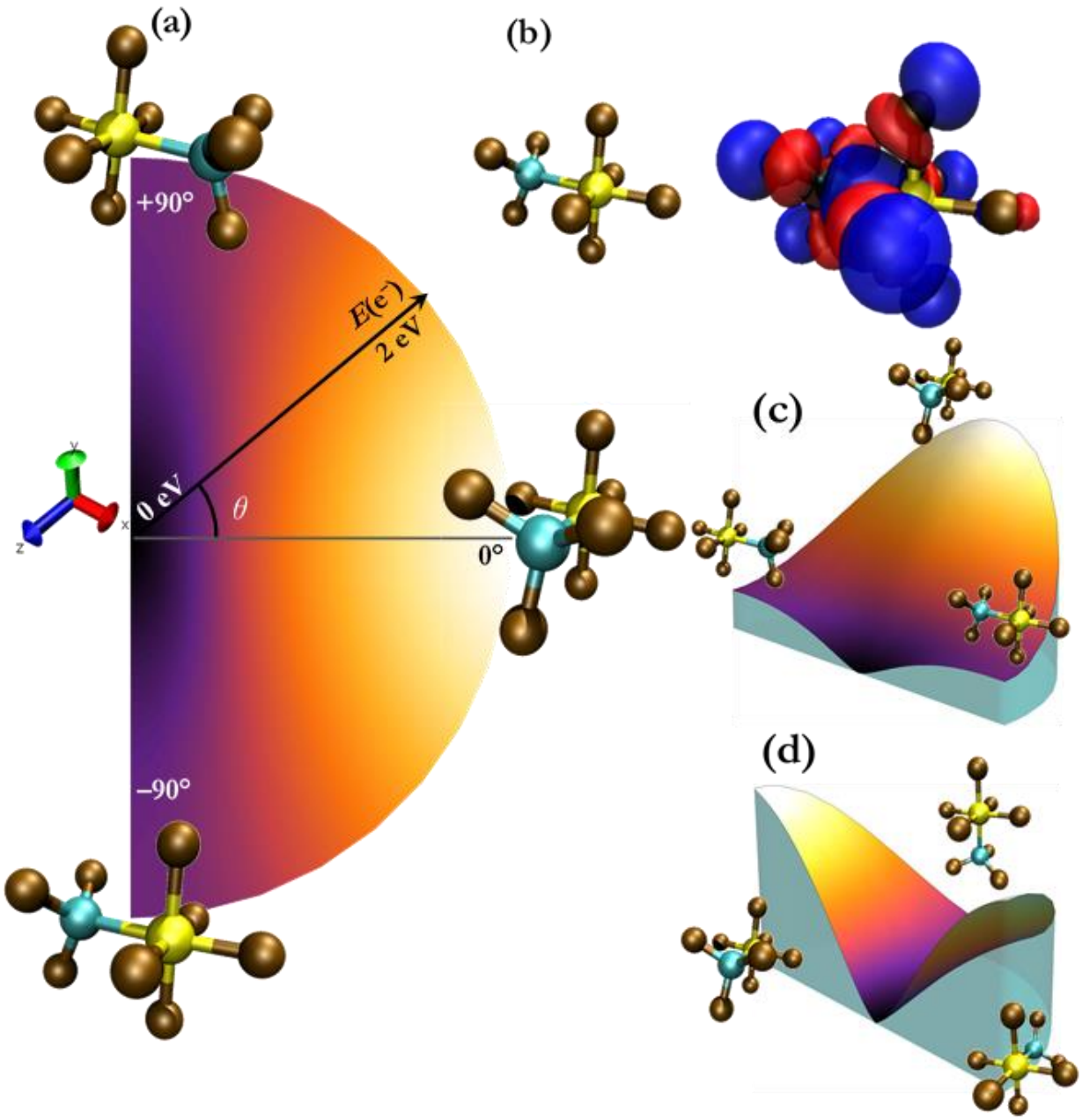


Figure S1. Calculated photoionisation cross sections at a photon energy of 14.1 eV and corresponding to ionisation from the first Dyson orbital (b), when the molecule is rotated along the *y* (a and c) or *x* (d) axis. The polarisation vector lies along the *z* axis.

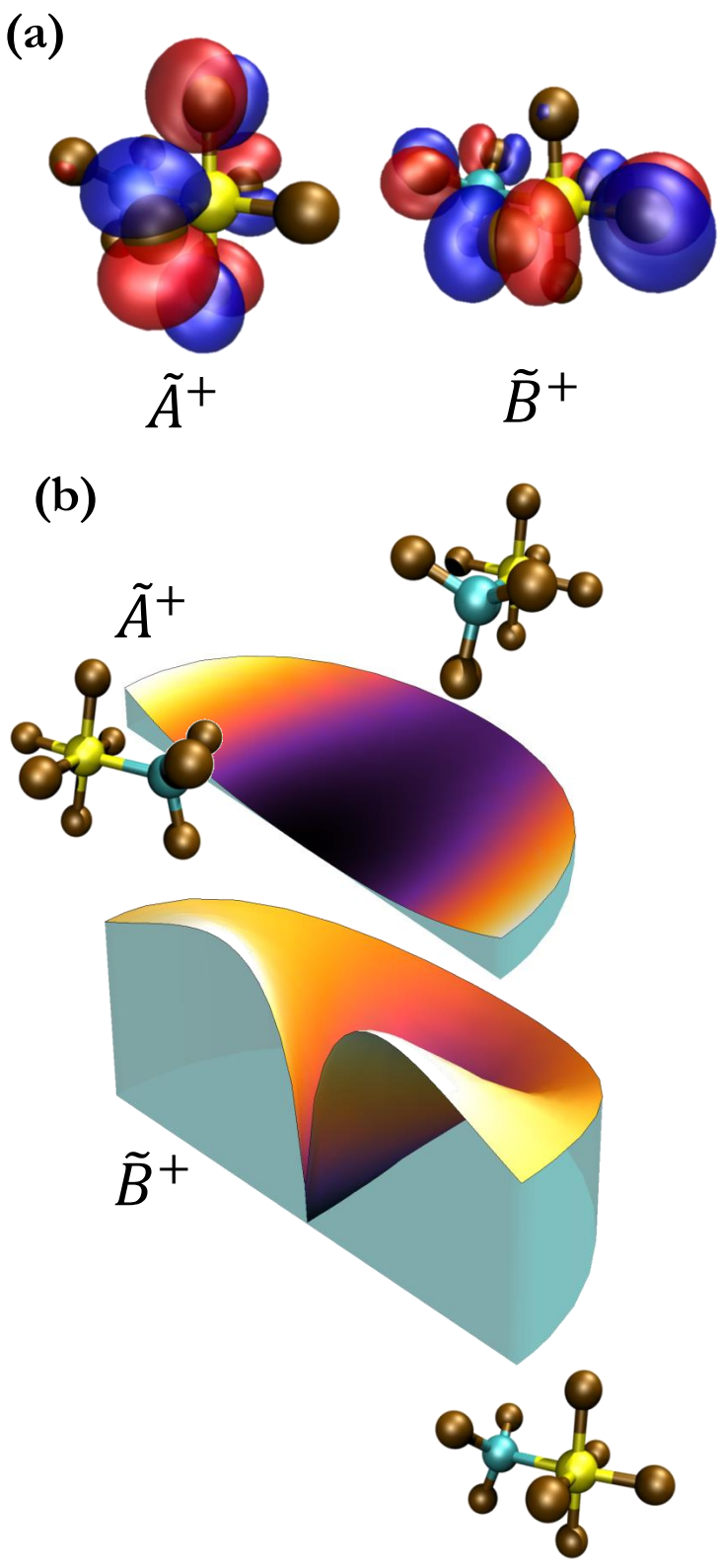


Figure S2. Dyson orbitals for (a) ionization to the $\tilde{A}^+$ and $\tilde{B}^+$ electronically excited states of $CF_3SF_5^+$, together with (b) computed angular dependence of the photoionisation cross sections. As opposed to the $\tilde{X}^+$ state, the angular dependence of these cross sections is smeared out by the long parent ion lifetime in these bound states. The absolute cross sections are predicted to be significantly smaller than for the $\tilde{X}^+$ state, whilst the more intense photoion signal is explained by more favorable Franck–Condon factors to these states.

The Python 2.7 code for coincident VMI reconstruction is provided in `i2Invert.py`. Its dependencies include the `numpy` and `scipy` libraries, both of which are available in the Anaconda platform, https://www.continuum.io/Anaconda-Overview, under Windows and more widely under Linux. For updates please check https://www.psi.ch/sls/vuv/pepico. A few examples illustrate the usage of the code below (see also `tutorial.txt`).

We start with the momentum distribution coefficient vector `coeff_1D.txt` together with the corresponding set of beta values `beta_1D.txt`, and also use the coefficients as an arbitrary radial distribution to initialise a 1D image reconstruction job:

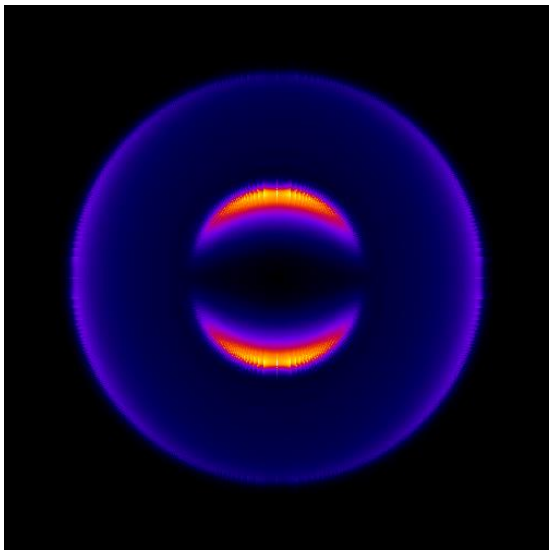
```
init1 coeff_1D.txt coeff_1D.txt na beta_1D.txt
```

This distribution contains two peaks, a low energy, parallel transition at $\beta = 2$, and a higher kinetic energy peak at $\beta = 0.5$. The radial distribution vector and the radial distribution vector of the P2-multiplied image can now be calculated and saved. These are the vectors are normally obtained by analysing the experimental velocity map images:

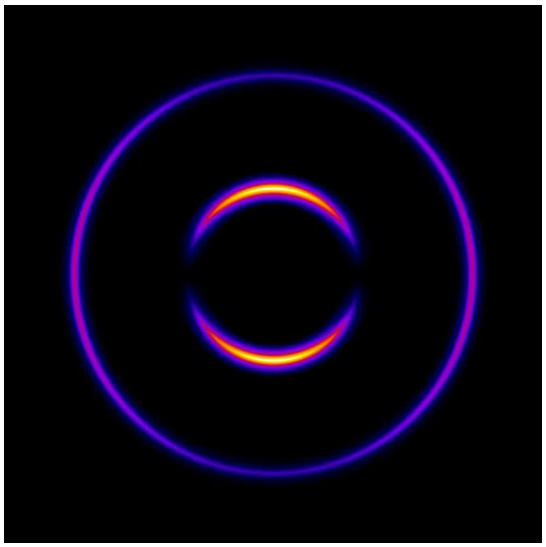
```
calc raddist_1D.txt
calc raddistP2_1D.txt beta
```

We can also use the coefficient and beta vectors to calculate the complete projected image as well as the equatorial slice of the image:

```
createimg 500 vmi_1D.txt
createimgslice 500 vmi_slice_1D.txt
```



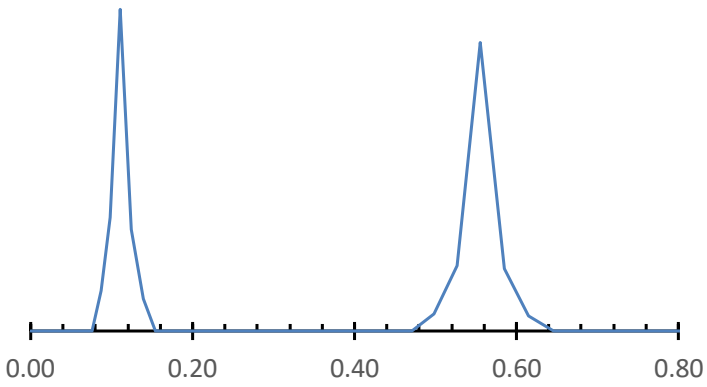
Complete velocity map image based on the coefficient and beta vectors as supplied.



The equatorial slice of the same projection.

`SaveSpec` can be used to save the energy spectrum using the coefficient vector and assuming an energy range of 1 (eV) imaged onto the detector:

```
savespec spect_1D.txt 1
```



Energy spectrum obtained based on the `coeff_1D.txt` coefficient vector assuming an imaged energy range of 1.

We can now use the radial distribution of the image and the P2-multiplied image to initialise a 1D job and fit the coefficient and beta vectors:

```
init1 raddist_1D.txt na raddistP2_1D.txt beta_1D.txt
fit coeff
fit beta
…
```

The results are saved under `coeff_1D_fit.txt` and `beta_1D_fit.txt`, and can be compared with the initial input used to calculate the radial distributions.

It is also possible to create a high resolution, 1500 × 1500 image, and let the program evaluate the radial distribution and P2-multiplied radial distribution vectors of a length of 200 elements:

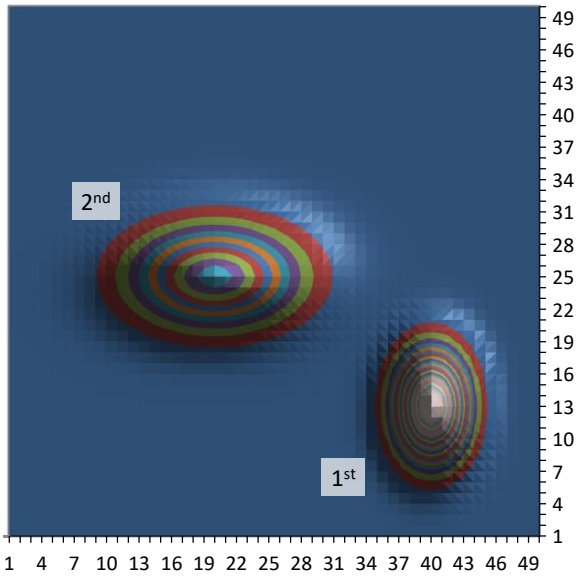
```
createimg 1500 vmi_1D_large.txt
processimg vmi_1D_large.txt 750 750 1 0 1 200 0.0025
```

The radial distributions thus obtained can also be fitted to yield virtually the same coefficient and beta vectors for the underlying kinetic energy and angular anisotropy distributions (see `*_1D_fit_large.txt`).

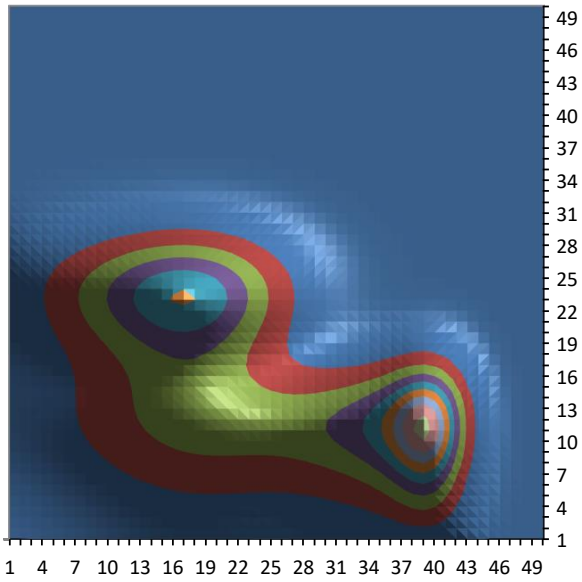
The inversion and reconstruction of single VMI data, as discussed hitherto, is routinely possible with a number of efficiently implemented and versatile approaches, some of which are cited in the main article text. Among these, the `i2Invert.py` program is uniquely capable of reconstructing coincidence double imaging data to obtain the momentum (and, hence, energy) correlation as well as the angular anisotropies of the imaged particles, formed in coincidence. As an example, `coeff_2D.txt` contains a 50 × 50 momentum correlation matrix with two peaks, which allows us to calculate the corresponding radial

correlation matrix. The angular anisotropies can be addressed analogously to the 1D problem, and are not shown here for simplicity.

```
init2 coeff_2D.txt coeff_2D.txt na na na
```
The radial distribution is calculated based on the coefficient matrix:
```
calc raddist_2D.txt
```

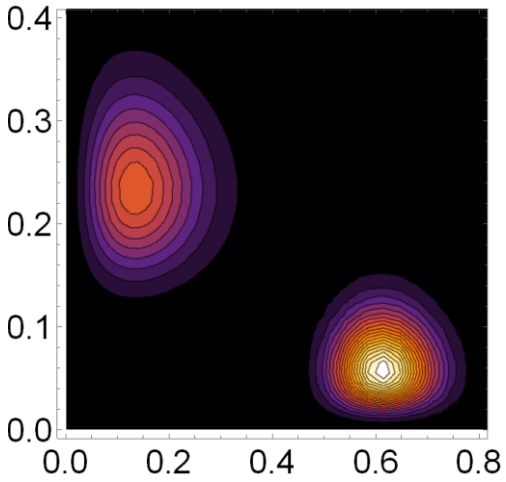


Momentum correlation matrix `coeff_2D.txt` displaying two peaks. Assuming the *x* momentum distribution corresponds to electrons, and the *y* momentum distribution to ions, slow ions are detected in coincidence with a sharp, fast electron distribution in the first peak and fast ions are formed in coincidence with slower, broadly spread electrons in the first peak.



VMI radial distribution correlation matrix resulting from the previous momentum correlation matrix. The projection to the *x*-axis gives the electron radial distribution, as observed in the total electron image, the projection along the *y*-axis yields the total ion radial distributions, as observed in the total ion image. Coincidence detection allows us to choose a set of ring regions of interest in, *e.g.*, the electron image, and by evaluating the coincident ion images, to obtain the radial correlation matrix column by column.

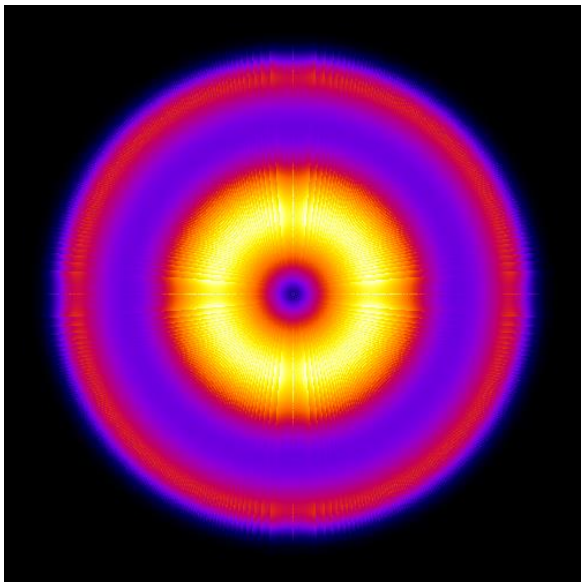
The radial correlation matrix is available experimentally, and can be used to reconstruct the momentum correlation of the particles, which reproduces the initial parameters used to calculate the radial correlation.



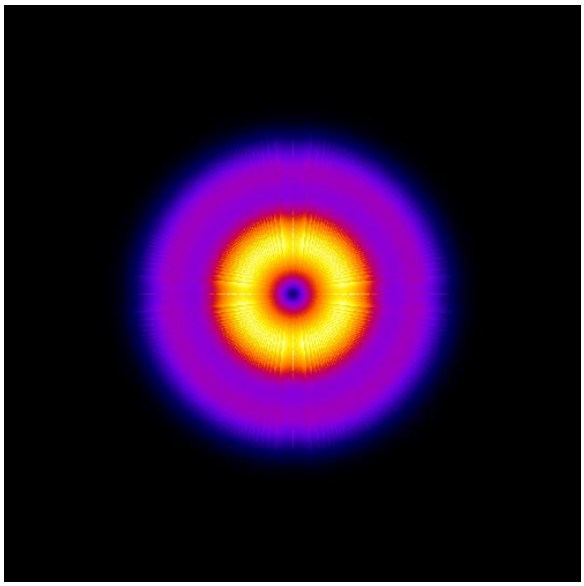
The energy correlation matrix between the two particles can be saved using `savespec spect_2D.txt 1 1`, assuming Newton spheres up to 1 (eV) kinetic energy are projected onto the detectors. Mind the different plot range as well as the non-linear transformation between momenta and energies, and the resulting distortion of the momentum correlation peaks.

```
init2 raddist_2D.txt na na na na
maxiter 40 150000
fit coeff
```

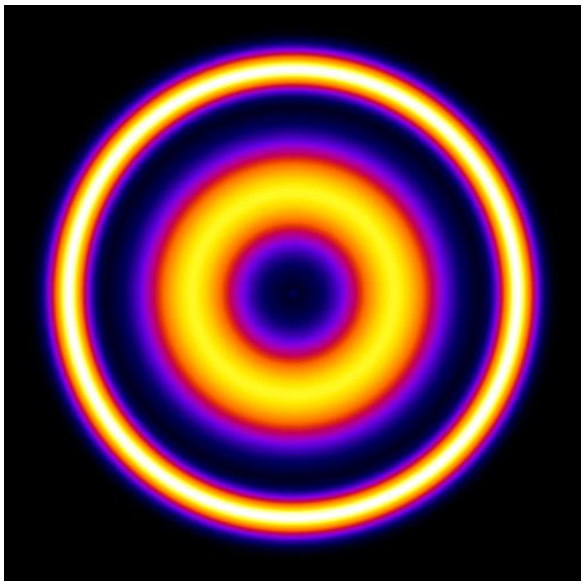
Finally, a few velocity map images and equatorial regions are shown, based on the momentum correlation matrix.



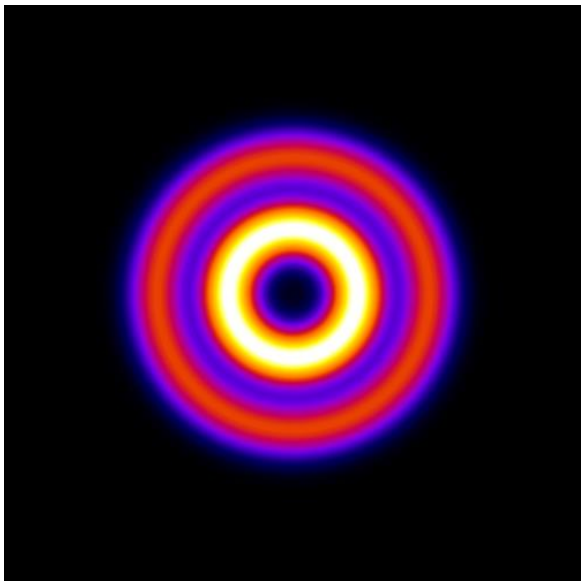
Total electron image (*x* axis in correlation)



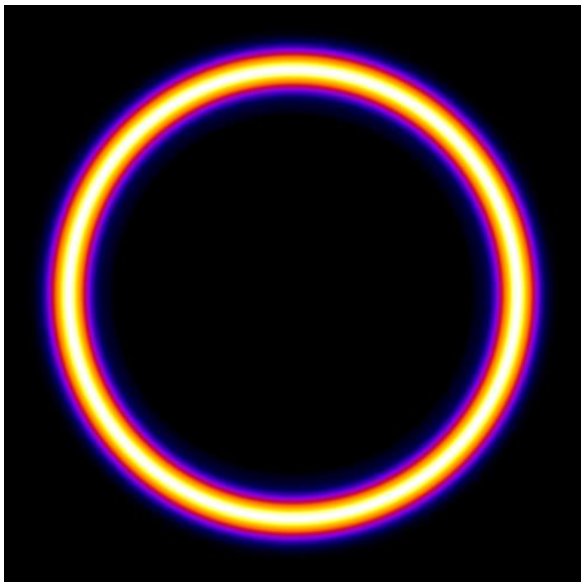
Total ion image (*y* axis in correlation)



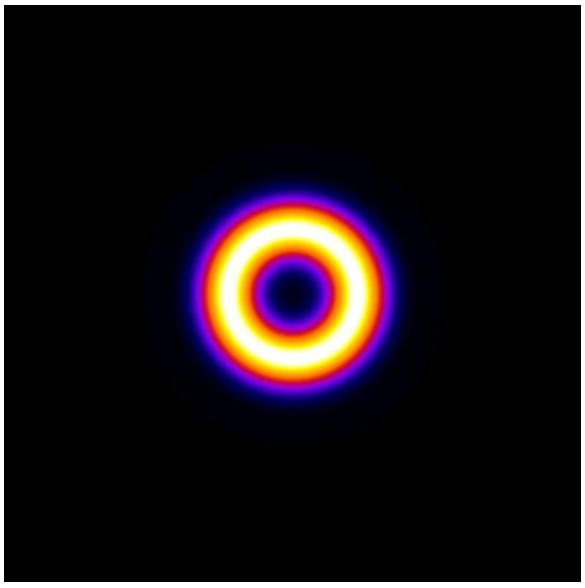
Equatorial slice of electron image



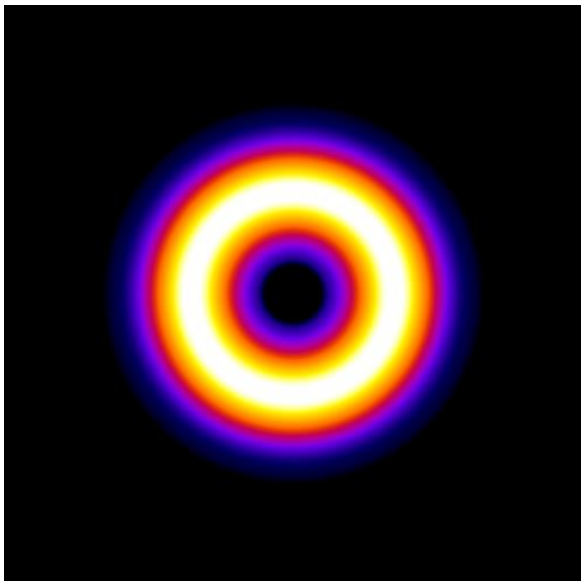
Equatorial slice of ion image



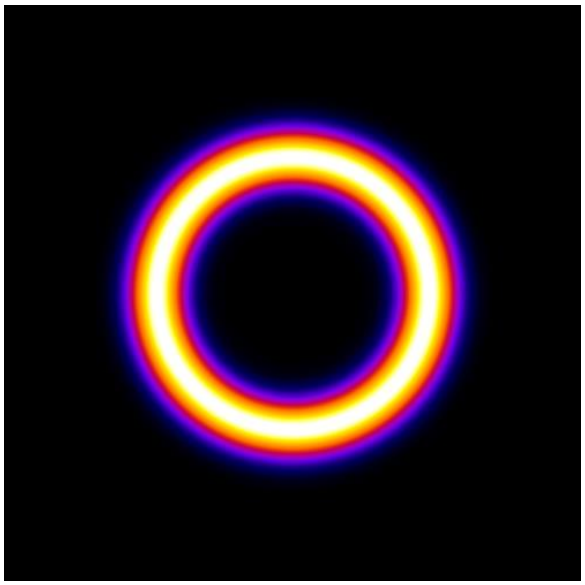
Equatorial slice of the high-energy electron peak



Thanks to the correlation matrix, we can establish that the high-energy electron peak is in coincidence with low-energy ions



Equatorial slice of the low-energy electron peak



Coincident image reconstruction yields high KE ions as coincident signal

The benefits of coincident image reconstruction are qualitatively evident in the last example. The electron signal in the low-radius region has contributions from the high-energy peak, which means that the ion image in coincidence with the central region on the electron detector has both low and high kinetic energy components. Only by way of coincident image reconstruction is it possible to obtain the pure high KE image shown on the right.

Coincident slicing approaches may also yield these equatorial images directly, without the need for inversion. However, electron slicing is challenging, and coincident slicing means that the overwhelming majority of the signal is discarded. Furthermore, slicing is only possible if the mass spectrum is sparse.

Python code for coincident VMI reconstruction. For updates and download please check https://www.psi.ch/sls/vuv/pepico.

i2Invert.py

```
#!/usr/bin/python
"""i2Invert.py Double VMI reconstruction written for Python 2.7 by Andras Bodi (c) 2017
Anaconda (https://www.continuum.io/anaconda-overview) is recommended under Windows
for mathematics libraries
A simple command line interpreter.
Input data:
 - A radial distribution vector for single VMI experiments
 - A square radial distribution matrix for double VMI experiments
 - Radial distribution vector/matrix of the raw image multiplied by P2(cos(a))
   where P2 is the 2nd-degree Legendre polynomial and a is the angle with the
   E polarization vector. Only the first of the two VMI images, i.e. the radial distribution of which
   corresponds to the column indeces in the coefficient matrix, is multiplied by P2,
   and the result is the angular anisotropy (beta) parameter matrix for that image.
Output data:
 - Momentum distributions in the original VMI space (1D vectors or 2D correlation matrices)
 - Beta parameters (1-image experiments -- 1 vector or 2-image experiments -- 2 matrices)
 - Momentum distribution can be transformed to energy-dependent spectra
 - VMI images as well as equatorial slices can be calculated
 - VMI images can be analysed to obtain raw and P2-multiplied radial distributions
Commands are documented individually below.
"""

from math import sqrt, pow, pi
from numpy import zeros, fromiter, sum, outer, power, ones, loadtxt, savetxt, array, pad
import scipy.optimize
try:
    import readline
except:
    pass
from time import strftime
from os import chdir
from scipy.interpolate import interp1d
from sys import stdout

def comp1Dbasis(j, R, E, beta):
    """Calculate 1D basis vector based on dimensionality of the problem and
    gridsize along a single coordinate.
    """
    if beta == 0:
# Eq1
        return (sqrt(E - j*j) - sqrt(E - (j+1)*(j+1))) / R
    else:
        f2 = (j+0.5)/R
        f2 *= f2
# Eq5
        return (sqrt(E - j*j) - sqrt(E - (j+1)*(j+1))) / R * (1+beta*(0.75*f2-0.5))


def comp1DP2basis(j, R, E, beta):
    """Calculate 1D basis vector for P2 integral based on dimensionality of the problem and
    gridsize along a single coordinate.
    """
    f2 = (j+0.5)/R
    f2 *= f2
# Eq6
    return (sqrt(E - j*j) - sqrt(E - (j+1)*(j+1))) / R * (1+beta*(0.75*f2-0.5)) * (8+beta*(15*f2-
4))/(32+beta*(24*f2-16))


def comp1ibeta(beta, coeff, expt):
    return comp1i(coeff, beta, expt, True)


def comp1i(coeff, beta, expt=None, calcP2=False, returncalc=False):
    """1-image calculation
    Compare radial distribution calculated based on coefficients with measured data.
    Whether the calculated radial distribution is returned depends on returncalc.
    """
    dim = len(coeff)
    computed = zeros(dim)
    if calcP2:
        fbasis = comp1DP2basis
    else:
        fbasis = comp1Dbasis
```

```
    for i in range(dim):
        mybeta = beta[i]
        R = i+1
        actE = R*R
        base = fromiter( (fbasis(j, R, actE, mybeta) for j in range(R)), float)
        base *= coeff[i]
        computed[0:R] += base
    if expt is not None:
        error = sum(power(dim*(computed-expt),2))
    else:
        error = 0
    if returncalc:
        return error, computed
    else:
        return error


def comp2ibeta(beta1, coeff, beta2, expt):
    return comp2i(coeff, beta1, beta2, expt, True)

def comp2i(coeff, beta1, beta2, expt=None, calcP2=False, returncalc=False):
    """2-image calculation
    Compare radial distribution calculated based on coefficients with measured data.
    Whether the calculated radial distribution is returned depends on returncalc.
    """
    size = len(coeff)
    dim = int(sqrt(size))
    computed = zeros([dim,dim])
    if calcP2:
        fbasis1 = comp1DP2basis
    else:
        fbasis1 = comp1Dbasis
    fbasis2 = comp1Dbasis
    for i in range(size):
        k = i % dim
        l = i / dim
        mybeta1 = beta1[i]
        mybeta2 = beta2[i]
        R1 = k + 1
        actE1 = R1 * R1
        R2 = l + 1
        actE2 = R2 * R2
        base1 = fromiter((fbasis1(j, R1, actE1, mybeta1) for j in range(R1)), float)
        base2 = fromiter((fbasis2(j, R2, actE2, mybeta2) for j in range(R2)), float)
# 2DBasis
        computed[0:R2,0:R1] += coeff[i] * outer(base2, base1)
    if expt is not None:
        error = sum(power(dim*dim*(computed-expt),2))
    else:
        error = 0
    if returncalc:
        return error, computed
    else:
        return error


def main():
    reqexit = False
    commandcount = 0
    mydim = 0
    maxiter = 10
    maxfun = 15000
    while not reqexit :                 # command interpreter loop
        commandcount += 1
        inp = raw_input( strftime('%Y-%m-%d %H:%M:%S') + ' [' + str(commandcount) +']: ').split()
        stdout.flush()
        if len(inp) > 0 :
            command = inp[0].upper()
        else:
            command = '#'
        if command == 'EXIT' or command == 'BYE':
            reqexit = True              # empty line or exit finishes program
        elif command == 'INIT1':
            #1-image job initialization
            if len(inp) != 5:
                print ' Init1 needs 4 parameters.'
                print '     param 1: radial distribution vector (initialized to 1 if gridsize given)'
                print '     param 2: coefficient vector (initialized to rad dist if "na")'
                print '     param 3: P2 radial distribution vector (initialized to rad dist if "na")'
                print '     param 4: beta vector (initialized to 0 if "na")'
                continue
```

```
            if inp[1].isdigit():
                gridsize = int(inp[1])
                expt = ones(gridsize)
            else:
                expt = loadtxt(inp[1])
                gridsize = len(expt)
            exptnorm = sum(expt)
            expt /= exptnorm
# 1DCoeffInit
            if inp[2].upper() != 'NA':
                coeffs = loadtxt(inp[2])
                coeffs /= sum(coeffs)
            else:
                coeffs = expt.copy()
            if inp[3].upper() != 'NA':
                exptp2 = loadtxt(inp[3])
                exptp2 /= exptnorm
            else:
                exptp2 = expt.copy()
            if inp[4].upper() != 'NA':
                beta = loadtxt(inp[4])
            else:
                beta = zeros(gridsize)
            mydim = 1
            print 'Done initializing 1-image job.'
        elif command == 'INIT2':
            # 2-image job initialization
            if len(inp) != 6:
                print ' Init2 needs 5 parameters:'
                print '     param 1: radial distribution matrix (initialized to 1 if gridsize given)'
                print '     param 2: coefficient matrix (initialized to rad dist if "na")'
                print '     param 3: P2 radial distribution matrix (initialized to rad dist if "na")'
                print '     param 4: beta matrix for image 1 (initialized to 0 if "na") (can be fitted)'
                print '     param 5: beta matrix for image 2 (initialized to 0 if "na") (constant)'
                continue
            if inp[1].isdigit():
                gridsize = int(inp[1])
                expt = ones([gridsize, gridsize])
            else:
                expt = loadtxt(inp[1])
                gridsize = expt.shape[0]
            exptnorm = sum(expt)
            expt /= exptnorm
            if inp[2].upper() != 'NA':
                coeffs = loadtxt(inp[2]).flatten()
                coeffs /= sum(coeffs)
            else:
                coeffs = expt.copy().flatten()
            if inp[3].upper() != 'NA':
                exptp2 = loadtxt(inp[3])
                exptp2 /= exptnorm
            else:
                exptp2 = expt.copy()
            if inp[4].upper() != 'NA':
                beta1 = loadtxt(inp[4]).flatten()
            else:
                beta1 = zeros(gridsize*gridsize)
            if inp[5].upper() != 'NA':
                beta2 = loadtxt(inp[4]).flatten()
            else:
                beta2 = zeros(gridsize*gridsize)
            mydim = 2
            print 'Done initializing 2-image job.'
        elif command == 'MAXITER':
            # maximum number of iterations in optimization
            # if given, the second parameter is the maximum number of function calls allowed
            maxiter = int(inp[1])
            if len(inp) > 2:
                maxfun = int(inp[2])
        elif command == 'FIT':
            # fit 1D radial distribution coefficients to reproduce measurement
            # or do anisotropy fit as determined by parameter 1
            if len(inp) != 2:
                print 'Fit needs 1 parameter: coeff or beta to be fitted.'
                continue
            myfit = None
            if mydim == 1:
                if inp[1].upper() == 'COEFF':
                    coeffbounds = [(0,1)] * gridsize
# Fit1DCoeff
                    myfit = scipy.optimize.fmin_l_bfgs_b(comp1i, coeffs, args=(beta,expt,), bounds=coeffbounds,
```

```
                                                        approx_grad=True, maxiter=maxiter, maxfun=maxfun)
                    coeffs = myfit[0].copy()
                    lasterror = comp1i(coeffs,beta,expt)
                elif inp[1].upper() == 'BETA':
                    betabounds = [(-1,2)] * gridsize
                    myfit = scipy.optimize.fmin_l_bfgs_b(comp1ibeta, beta, args=(coeffs,exptp2,),
bounds=betabounds,
                                                        approx_grad=True, maxiter=maxiter, maxfun=maxfun)
                    beta = myfit[0].copy()
                    lasterror = comp1i(coeffs,beta,expt,True)
            elif mydim == 2:
                if inp[1].upper() == 'COEFF':
                    coeffbounds = [(0,1)] * gridsize * gridsize
# Fit2DCoeff
                    myfit = scipy.optimize.fmin_l_bfgs_b(comp2i, coeffs, args=(beta1,beta2,expt,),
bounds=coeffbounds,
                                                        approx_grad=True, maxiter=maxiter, maxfun=maxfun)
                    coeffs = myfit[0].copy()
                    lasterror = comp2i(coeffs,beta1,beta2,expt)
                elif inp[1].upper() == 'BETA':
                    betabounds = [(-1,2)] * gridsize * gridsize
                    myfit = scipy.optimize.fmin_l_bfgs_b(comp2ibeta, beta1, args=(coeffs,beta2,exptp2,),
bounds=betabounds,
                                                        approx_grad=True, maxiter=maxiter, maxfun=maxfun)
                    beta1 = myfit[0].copy()
                    lasterror = comp2i(coeffs,beta1,beta2,exptp2,True)
            if myfit is not None:
                print 'Done fit.'
                print 'Error function:', myfit[1]
                print 'Warning flag:', myfit[2]['warnflag']
                print 'Number of function calls:', myfit[2]['funcalls']
                print 'Number of iterations:', myfit[2]['nit']
            else:
                print 'No fit done for some reason.'
        elif command == 'CALC':
            # calculate the radial distribution vector or matrix and save it to param1
            myres = None
            if len(inp) == 2:
                calcP2 = False
            elif len(inp) == 3:
                calcP2 = (inp[2].upper() == 'BETA')
            else:
                print 'No file name given'
                continue
            if not calcP2:
                myexpt = expt
            else:
                myexpt = exptp2
            if mydim == 1:
                myres = comp1i(coeffs, beta, myexpt, calcP2=calcP2, returncalc=True)
            elif mydim == 2:
                myres = comp2i(coeffs, beta1, beta2, myexpt, calcP2=calcP2, returncalc=True)
            if myres is not None:
                savetxt(inp[1], exptnorm * myres[1])
                print 'Done calculating, error function:', myres[0], 'data saved to:', inp[1]
            else:
                print 'Not done calculating.'
        elif command == 'LOADCOEFF':
            # load coefficients from file (parameter 1)
            coeffs = loadtxt(inp[1]).flatten()
            coeffs /= sum(coeffs)
            print 'Done reading coefficients from file.'
        elif command == 'SAVECOEFF':
            # save coefficients to file (parameter 1)
            savecoeffs = coeffs.flatten() * exptnorm
            if mydim == 1:
                savetxt(inp[1], savecoeffs)
                print 'Coefficients saved to', inp[1]
            elif mydim == 2:
                savetxt(inp[1], array([savecoeffs[i:i+gridsize] for i in xrange(0, len(coeffs), gridsize)]))
                print 'Coefficients saved to', inp[1]
        elif command == 'SAVERDIST':
            # save experimental radial distribution to file (parameter 1)
            saverdist = expt.flatten() * exptnorm
            if mydim == 1:
                savetxt(inp[1], saverdist)
                print 'Radial distribution saved to', inp[1]
            elif mydim == 2:
                savetxt(inp[1], array([saverdist[i:i+gridsize] for i in xrange(0, len(coeffs), gridsize)]))
                print 'Radial distribution saved to', inp[1]
        elif command == 'SAVERDISTP2':
```

```
            # save experimental radial distribution to file (parameter 1)
            saverdist = exptp2 * exptnorm
            if mydim == 1:
                savetxt(inp[1], saverdist)
                print 'P2-multiplied radial distribution saved to', inp[1]
            elif mydim == 2:
                savetxt(inp[1], array([saverdist[i:i + gridsize] for i in xrange(0, len(coeffs), gridsize)]))
                print 'P2-multiplied radial distribution saved to', inp[1]
        elif command == 'SAVESPEC':
            # transform coefficients from momentum to energy space
            if len(inp) != 2 + mydim:
                print ' SaveSpec usage:'
                print '    parameter 1: file name to which the spectrum is saved'
                print '    parameter 2: energy range along 1st coordinate'
                print '    parameter 3: energy range along 2nd coordinate (for 2D correlation matrices)'
                continue
            if mydim == 2:
                conversionx = float(inp[2]) / pow(gridsize + 1,2)
                conversiony = float(inp[3]) / pow(gridsize + 1,2)
                converted = []
                for i in range(gridsize):
                    for j in range(gridsize):
# 2DSpect

converted.append([conversionx*(i+1)*(i+1),conversiony*(j+1)*(j+1),coeffs[i+j*gridsize]/(4*conversionx*conversio
ny*(i+1)*(j+1))])
                savetxt(inp[1], array(converted))
                print 'Spectrum saved to', inp[1]
            elif mydim == 1:
                conversionx = float(inp[2]) / pow(gridsize + 1,2)
                converted = []
# 1DSpect
                for i in range(gridsize):
                        converted.append([conversionx*(i+1)*(i+1),coeffs[i]/(2*conversionx*(i+1))])
                savetxt(inp[1], array(converted))
                print 'Spectrum saved to', inp[1]
        elif command == 'SAVEBETASPEC':
            # same as savespec but as beta values are saved, they are not transformed
            if len(inp) != 3 + mydim:
                print ' SaveBetaSpec usage:'
                print '    parameter 1: file name to which the spectrum is saved'
                print '    parameter 2: energy range along 1st coordinate'
                print '    parameter 3: energy range along 2nd coordinate (for 2D correlation matrices)'
                print '    parameter 4: whether beta should be saved for first (1) or second (2) image'
                continue
            if mydim == 2:
                conversionx = float(inp[2]) / pow(gridsize + 1,2)
                conversiony = float(inp[3]) / pow(gridsize + 1,2)
                if int(inp[4]) == 1:
                    savebeta = beta1
                elif int(inp[4]) == 2:
                    savebeta = beta2
                else:
                    print 'No idea which beta I should save.'
                    continue
                converted = []
                for i in range(gridsize):
                    for j in range(gridsize):

converted.append([conversionx*(i+1)*(i+1),conversiony*(j+1)*(j+1),savebeta[i+j*gridsize]])
                savetxt(inp[1], array(converted))
                print 'Spectrum saved to', inp[1]
            elif mydim == 1:
                conversionx = float(inp[2]) / pow(gridsize + 1,2)
                converted = []
                for i in range(gridsize):
                        converted.append([conversionx*(i+1)*(i+1),beta[i]])
                savetxt(inp[1], array(converted))
                print 'Spectrum saved to', inp[1]
        elif command == 'LOADBETA':     # load beta vector/matrix from file
            if len(inp) != 1 + mydim:
                print ' LoadBeta usage:'
                print '    parameter 1: file name from which beta parameters are loaded'
                print '    parameter 2: if beta refers to the first (1) or second (2) image in 2-image jobs'
                continue
            if mydim == 1:
                beta = loadtxt(inp[1])
                print 'Beta parameters loaded from', inp[1]
            elif int(inp[2]) == 1:
                beta1 = loadtxt(inp[1]).flatten()
                print 'Beta_1 parameters loaded from', inp[1]
```

```
            elif int(inp[2]) == 2:
                beta2 = loadtxt(inp[1]).flatten()
                print 'Beta_2 parameters loaded from', inp[1]
            else:
                print 'No idea what I should do.'
        elif command == 'SAVEBETA':     # save beta vector/matrix to file
            if len(inp) == 1:
                print ' SaveBeta usage:'
                print '    parameter 1: file name to which beta parameters are saved'
                print '    [parameter 2]: if beta refers to the first (1 default) or second (2) image in 2-
image jobs'
                continue
            if mydim == 1:
                savetxt(inp[1], beta)
                print 'Beta parameters saved to', inp[1]
            elif (len(inp) == 2) or (int(inp[2]) == 1):
                savetxt(inp[1], array([beta1[i:i+gridsize] for i in xrange(0, len(beta1), gridsize)]))
                print 'Beta_1 parameters saved to', inp[1]
            elif int(inp[2]) == 2:
                savetxt(inp[1], array([beta2[i:i+gridsize] for i in xrange(0, len(beta2), gridsize)]))
                print 'Beta_2 parameters saved to', inp[1]
            else:
                print 'No idea what I should do.'
        elif command == "SAVEEXPT":
            savetxt(inp[1], exptnorm * expt)
        elif command == "SAVEEXPTP2":
            savetxt(inp[1], exptnorm * exptp2)
        elif command == "EXPTNORM":
            print('Experimental norm is:', exptnorm)
        elif command == 'CREATEIMG':
            # take 1d coefficient and beta vectors and compute a full projected VMI image
            # param 1 image dimensions along each of the axesin
            # param 2 file to save image to
            if mydim == 2:
                print "Only 1-image calculations are supported."
                continue
            imgsize = int(inp[1])
            imgp2 = imgsize/2.0
            img = zeros((imgsize,imgsize))
            coeffsize = imgsize/2
            interpol_coeff = interp1d(range(gridsize), coeffs, kind='linear')
            interpol_betac = interp1d(range(gridsize), beta*coeffs, kind='linear')
            # beta * coeff needed so that coeff aren't multiplied by invalid beta in interpolation
            pixdif = 1/imgp2
            # to get rid of Dirac delta at r = R
            for k in xrange(coeffsize):
                rcoeff = float(k)*(gridsize-1)/coeffsize
                mycoeff = interpol_coeff(rcoeff)
                if mycoeff > 0:
                    mybetac = interpol_betac(rcoeff) / mycoeff
                    rcoeff /= (gridsize-1)
                    imgbuf = zeros((imgsize, imgsize))
                    for i in xrange(imgsize):
                        for j in xrange(imgsize):
                            r = sqrt(pow(i - imgp2, 2) + pow(j - imgp2, 2)) / imgp2
                            if rcoeff >= (r + pixdif):
                                try:
                                    cosphi = (i - imgp2) / (imgp2 * r)
                                    imgbuf[i,j] = mycoeff * (sqrt(pow(rcoeff,2)-pow(r,2)) - sqrt(pow(rcoeff,2)-
pow(r+pixdif,2))) / (pi * pow(rcoeff,2)) * (1 + mybetac * 0.5 * (3*pow(r/rcoeff * cosphi,2) - 1))
                                except ZeroDivisionError:
                                    pass
                    imgbufsum = imgbuf.sum()
                    if imgbufsum > 0:
                        imgbuf *= mycoeff / imgbufsum
                        img += imgbuf
            img *= exptnorm * coeffs.sum() / img.sum()
            savetxt(inp[2],img)
            print 'Text image saved to', inp[2]
        elif command == 'CREATEIMGSLICE':
            # take 1d coefficient and beta vectors and compute a VMI image slice
            # param 1 image dimensions along each of the axes (image will fill the area)
            # param 2 file to save image to
            if mydim == 2:
                print "Only 1-image calculations are supported."
                continue
            imgsize = int(inp[1])
            img = zeros((imgsize,imgsize))
            interpol_coeff = interp1d(range(gridsize), coeffs, kind='linear')
            interpol_betac = interp1d(range(gridsize), beta*coeffs, kind='linear')
            # beta * coeff needed so that coeff aren't multiplied by invalid beta in interpolation
```

```
            for i in xrange(imgsize):
                for j in xrange(imgsize):
                    r = sqrt(pow(i - imgsize/2, 2) + pow(j - imgsize/2, 2))
                    rcoeff = r / (imgsize/2.0) * gridsize
                    if rcoeff < (gridsize-1):
                        if interpol_coeff(rcoeff) > 0:
                            try:
                                cosphi = (i - imgsize/2.0) / r
                                mycoeff = interpol_coeff(rcoeff)
                                img[i,j] = mycoeff/r * (1 + interpol_betac(rcoeff)/mycoeff * 0.5 *
(3*pow(cosphi,2) - 1))
                            except ZeroDivisionError:
                                pass
            img *= exptnorm / img.sum()
            savetxt(inp[2],img)
            print 'Text image of the equatorial region saved to', inp[2]
        elif command == 'PROCESSIMG':
            # take VMI image and calculate radial distribution and P2-radial distribution
            if len(inp) != 9:
                print ' ProcessImg usage:'
                print '    parameter 1: file to load image from'
                print '    parameter 2: center x coordinate'
                print '    parameter 3: center y coordinate'
                print '    parameter 4: epsilon vector x coordinate'
                print '    parameter 5: epsilon vector y coordinate'
                print '    parameter 6: ellipsoid anisotropy of image (same definition as i2PEPICO measurement
ROI anisotropy)'
                print '    parameter 7: number of radial distributuion vector elements'
                print '    parameter 8: Delta r of radial distribution as a ratio of image size'
                continue
            print 'Initializing new 1-dim job...'
            mydim = 1
            exptvmi = loadtxt(inp[1])
            vmisize = exptvmi.shape[0]
            centerx = float(inp[2])
            centery = float(inp[3])
            evecx = float(inp[4])
            evecy = float(inp[5])
            elen = sqrt(pow(evecx,2) + pow(evecy,2))
            evecx /= elen
            evecy /= elen
            aniso = float(inp[6])
            gridsize = int(inp[7])
            deltar = float(inp[8]) * vmisize

            expt = zeros(gridsize)
            exptp2 = zeros(gridsize)
            beta = zeros(gridsize)

            for i in range(vmisize):
                for j in range (vmisize):
                    dx = (i - centerx) / aniso
                    dy = (j - centery)
                    r = sqrt(pow(dx, 2) + pow(dy, 2))
                    bin = int(r/deltar)
                    if bin < gridsize:
                        expt[bin] += exptvmi[i, j]
                        if r > 0:
                            dx /= r
                            dy /= r
                            p2val = 0.5 * (3 * pow(dx * evecx + dy * evecy, 2) - 1)
                            exptp2[bin] += exptvmi[i, j] * p2val

            exptnorm = expt.sum()
            if exptnorm > 0:
                expt /= exptnorm
                exptp2 /= exptnorm
            coeffs = expt.copy()
            print 'Done initializing new 1-dim job based on loaded VMI.'
        elif command == 'CD':
            # change active directory
            chdir(inp[1])
        elif command == 'SMOOTHCOEFF':
            # smooth coefficients based on neighbouring values
            # parameter 1 -- what to smooth (coeff or "beta" for beta1)
            # parameter 2 -- the weight of the original values in the averaging,
            # which is 50% at 1.0
            if mydim == 2:
                if inp[1].upper() == 'BETA':
                    smoothing = array([beta1[i:i+gridsize] for i in xrange(0, len(beta1), gridsize)])
                else:
```

```
                    smoothing = array([coeffs[i:i+gridsize] for i in xrange(0, len(coeffs), gridsize)])
                smoothingbuf = zeros((gridsize,gridsize))
                smoothingbuf += pad(smoothing, ((0,0),(1,0)), mode='constant')[:,:-1]
                smoothingbuf += pad(smoothing, ((0,0),(0,1)), mode='constant')[:,1:]
                smoothingbuf += pad(smoothing, ((1,0),(0,0)), mode='constant')[:-1,:]
                smoothingbuf += pad(smoothing, ((1,0),(1,0)), mode='constant')[:-1,:-1]
                smoothingbuf += pad(smoothing, ((1,0),(0,1)), mode='constant')[:-1,1:]
                smoothingbuf += pad(smoothing, ((0,1),(0,0)), mode='constant')[1:,:]
                smoothingbuf += pad(smoothing, ((0,1),(1,0)), mode='constant')[1:,:-1]
                smoothingbuf += pad(smoothing, ((0,1),(0,1)), mode='constant')[1:,1:]
                if inp[1].upper() == 'BETA':
                    beta1 = ((8 * float(inp[2]) * smoothing + smoothingbuf)/(8 * float(inp[2])+8)).flatten()
                    print 'beta1 parameters are now smooth.'
                else:
                    coeffs = ((8 * float(inp[2]) * smoothing + smoothingbuf)/(8 * float(inp[2])+8)).flatten()
                    coeffs /= sum(coeffs)
                    print 'i2 coefficients are now smooth.'
            else:
                print 'You can smooth a vector yourself.'
        else:                           # command not supported or comment
            if command != '#':
                print 'I do not understand your input,', inp
    print 'Bye.'


if __name__ == '__main__' :
    main()
```