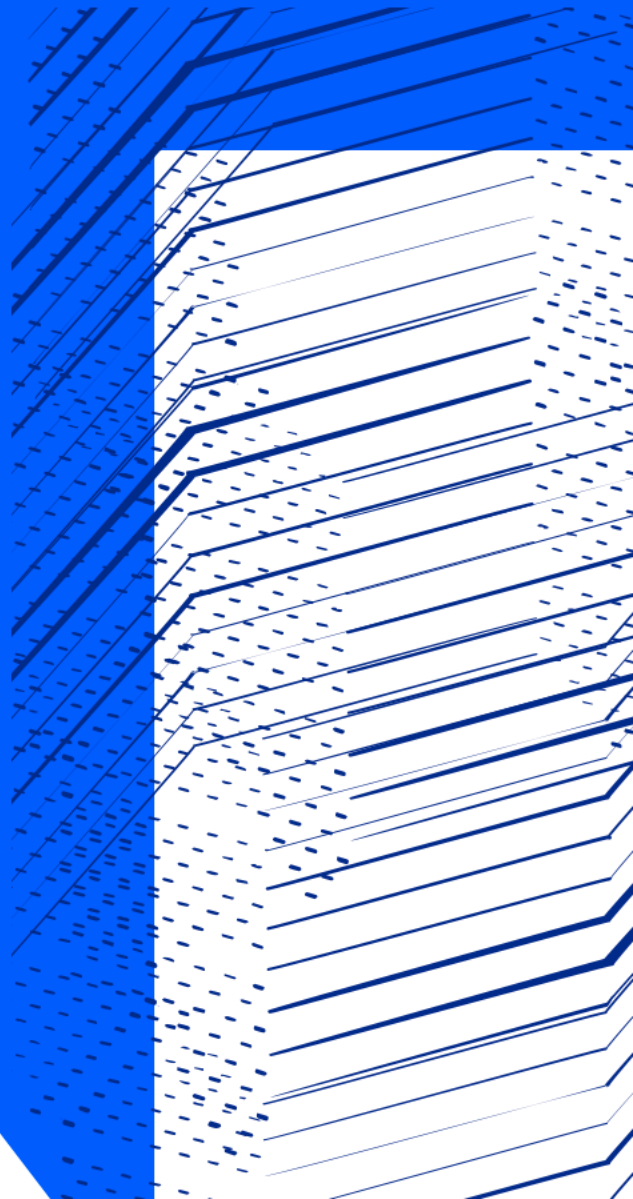
Science and Technology Facilities Council

Legacy of Scientific code

Dr. Anthony Lim,
Dr. Martyn Gigg, Dr Stephen Smith



Contents

- Introduction
- Failing fast
- Iterative improvements
- Useful tools
- Conclusion

Science and Technology Facilities Council

Introduction

- Was asked to work on the quasielasticbayes package
  - The original code is written in Fortran
  - Has in recent years been made available as a Python package
- Has an associated publication from 1992
- The Fortran code has not changed for many years and has Fortran 90 extension

Physica B 182 (1992) 341–348
North-Holland

PHYSICA B

Bayesian analysis of quasielastic neutron scattering data

D.S. Sivia, C.J. Carlile and W.S. Howells
Rutherford Appleton Laboratory, Chilton, Didcot, Oxon, UK

S. König
Physik-Department E22, Technische Universität München, D-8046 Garching, Germany

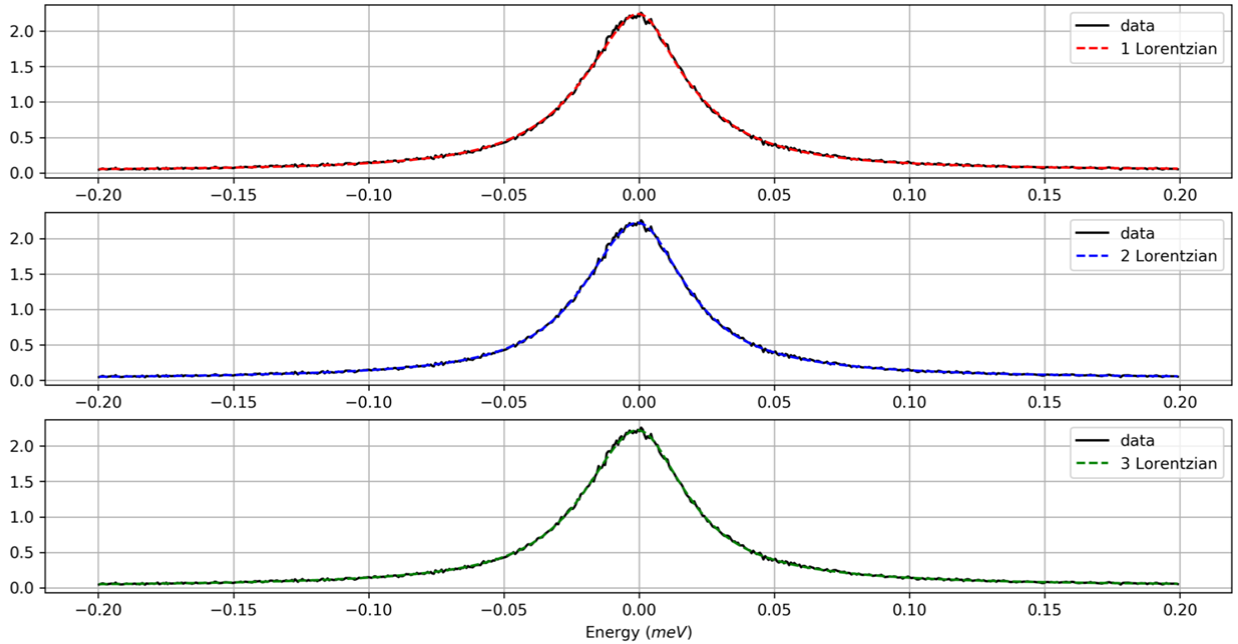
We consider the analysis of quasielastic neutron scattering data from a Bayesian point-of-view. This enables us to use probability theory to assess how many quasielastic components there is most evidence for in the data, as well as providing an optimal estimate of their parameters. We review the theory briefly, describe an efficient algorithm for its implementation and illustrate its use with both simulated and real data.

Introduction

- The code is designed to give the fit parameters for quasi elastic data
  - Background
  - Offset
  - Elastic peak
  - Inelastic features

- The results are accepted as being correct

- The Bayesian part gives the likelihood for 0, 1, 2, 3 inelastic features (e.g. Lorentzian)



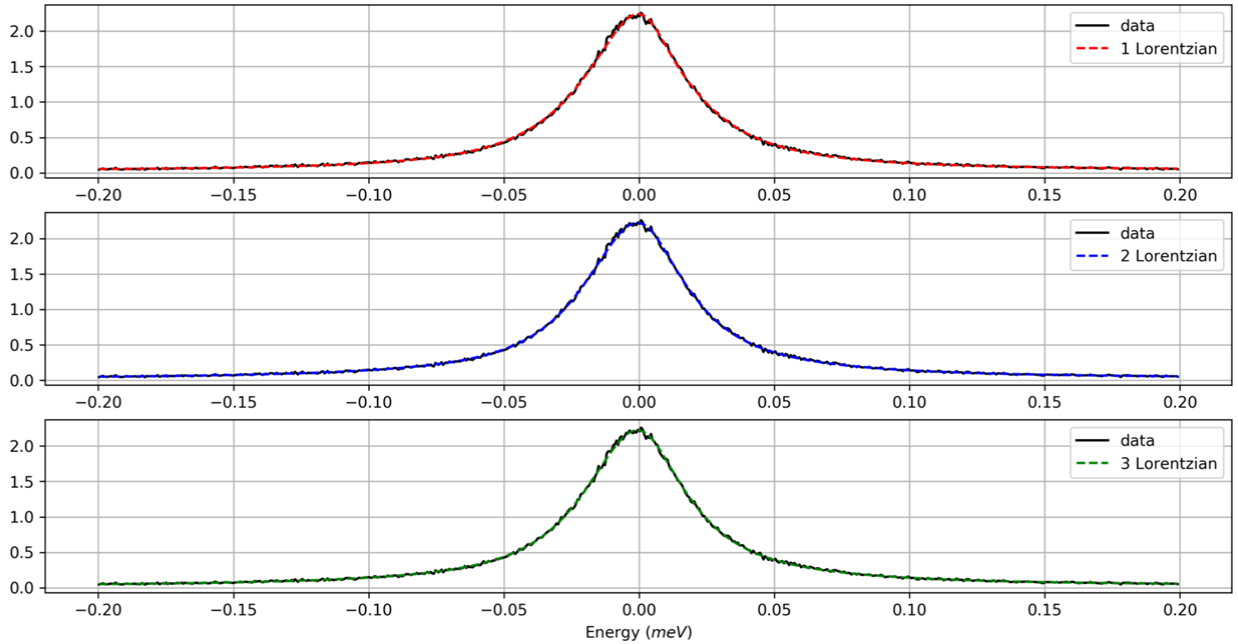
Science and Technology Facilities Council

Introduction

- The code is designed to give the fit parameters for quasi elastic data
  - Background
  - Offset
  - Elastic peak
  - Inelastic features

- The results are accepted as being correct

- The Bayesian part gives the likelihood for 0, 1, 2, 3 inelastic features (e.g. Lorentzian)

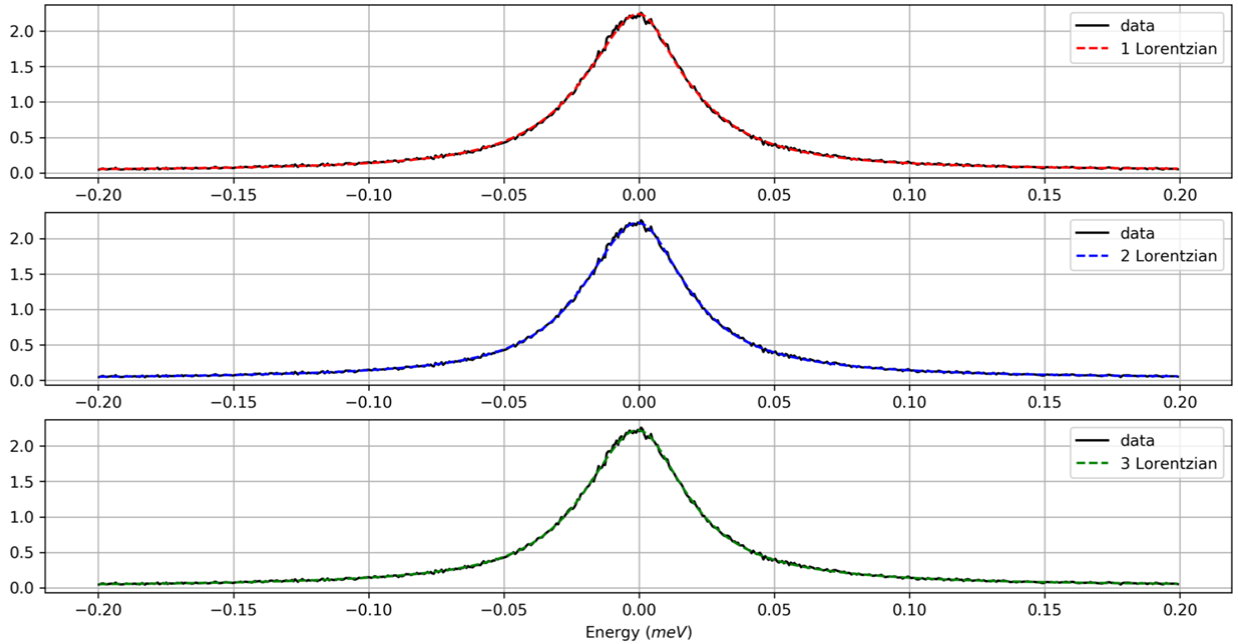


The most likely is 2 Lorentzians

Science and Technology Facilities Council

So, what's the problem?

- No one understands how the results are generated
  - The paper focuses on verifying results
  - The author is no longer available to discuss the method
- Written in the style of Fortran 70
- The code is difficult to compile (modern compilers do not work)
- Gives different answers on Linux and Windows

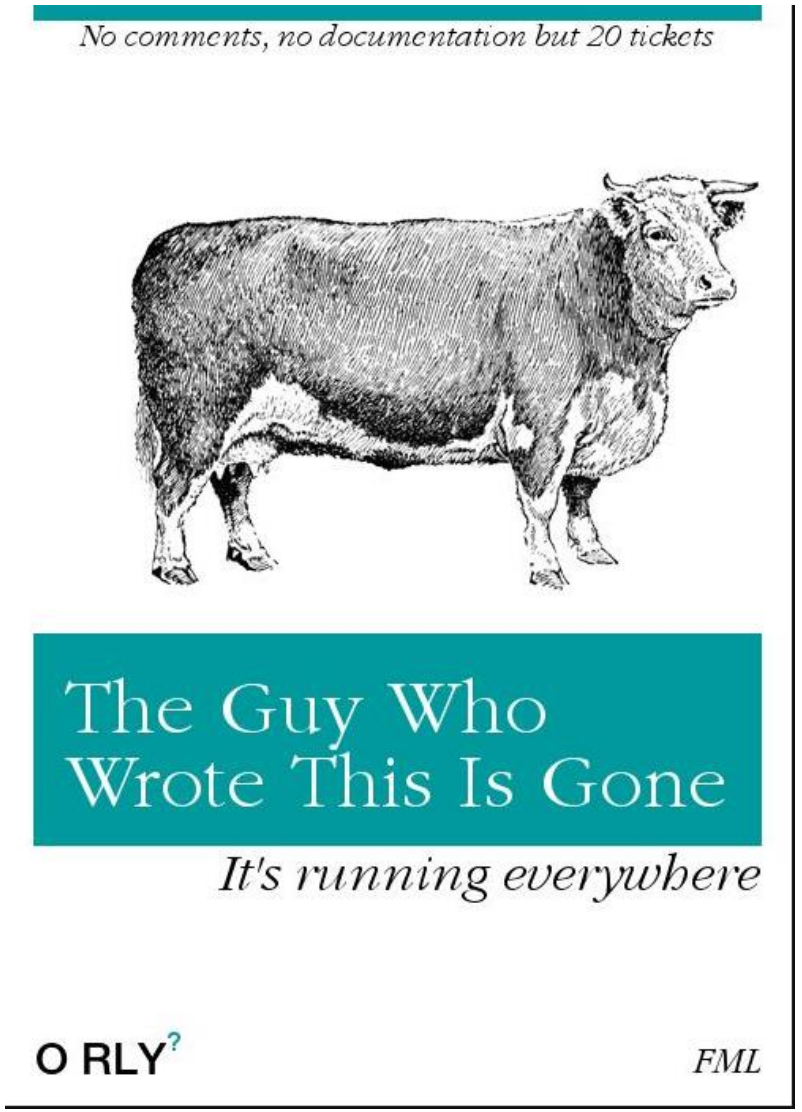


Science and Technology Facilities Council

Legacy code

- Legacy scientific code is normally not:
  - Documented
  - Maintainable
  - Readable
  - Commented

- It is also possible that:
  - The author has left
  - No one knows how it works – just that its right



Science and Technology Facilities Council

Fail fast

- Snippet of some Fortran code
- Whats the function name mean?
- What are the variables (e.g. WX)?
- Why does it close a file that has not been opened?
- What values are changed when exiting the subroutine?

```
166         SUBROUTINE BINBLR(WX,WY,WE,NB,XB,YB,NBIN)
167         INCLUDE 'mod_files.f90'
168         REAL WX(*),WY(*),WE(*),XB(*),YB(*)
169         N=0
170         SMALL=1.0E-20
171         BNORM=1.0/FLOAT(NBIN)
172         do I=1,NB,NBIN
173           N=N+1
174           XXD=0.0
175           DD=0.0
176           K=0
177           do J=0,NBIN-1
178            IJ=I+J
179             IF (IJ.LE.NB) THEN
180               XXD=XXD+WX(IJ)
181               IF (WE(IJ).GT.SMALL) THEN
182                 K=K+1
183                 DD=DD+WY(IJ)
184               ENDIF
185             ENDIF
186           end do
187           XB(N)=BNORM*XXD
188           YB(N)=0.0
189           IF (K.GT.0) YB(N)=BNORM*DD
190         end do
191         NB=N
192         close(unit=53)
193         END
```

Science and Technology Facilities Council

Fail fast

- Kept as much the same as possible
- Converted Fortran arrays to numpy arrays
- Add some comments from understanding of the code

```
def BINBLR(WX,WY,WE,NB,NBIN):
      """
      Original dat is W* and the output is *B
      It seems to just be a rebin alg
      """
      XB = np.zeros(NB)
      YB = np.zeros(NB)

      SMALL=1.0E-20
      BNORM=1.0/float(NBIN)

      for I in get_range(NB,NBIN):
        XXD=0.0
        DD=0.0
        K=0
        for J in range(NBIN-2):
            IJ=I+J
            if IJ<=NB:
                XXD=XXD+WX[IJ]
                if WE[IJ] > SMALL: # only include non-zero errors
                    K=K+1
                    DD=DD+WY[IJ]

        XB[I] = BNORM*XXD
        YB[I] = 0.0
        if K>0:
            YB[I] = BNORM*DD
      return XB, YB
```

Science and Technology Facilities Council

Fail fast

```
166         SUBROUTINE BINBLR(WX,WY,WE,NB,XB,YB,NBIN)
167         INCLUDE 'mod_files.f90'
168         REAL WX(*),WY(*),WE(*),XB(*),YB(*)
169         N=0
170         SMALL=1.0E-20
171         BNORM=1.0/FLOAT(NBIN)
172         do I=1,NB,NBIN
173           N=N+1
174           XXD=0.0
175           DD=0.0
176           K=0
177           do J=0,NBIN-1
178            IJ=I+J
179             IF (IJ.LE.NB) THEN
180               XXD=XXD+WX(IJ)
181               IF (WE(IJ).GT.SMALL) THEN
182                 K=K+1
183                 DD=DD+WY(IJ)
184               ENDIF
185             ENDIF
186           end do
187           XB(N)=BNORM*XXD
188           YB(N)=0.0
189           IF (K.GT.0) YB(N)=BNORM*DD
190         end do
191         NB=N
192         close(unit=53)
193         END
```

```
1  def BINBLR(WX,WY,WE,NB,NBIN):
2        """
3        Original dat is W* and the output is *B
4        It seems to just be a rebin alg
5        """
6        XB = np.zeros(NB)
7        YB = np.zeros(NB)
8
9        SMALL=1.0E-20
10       BNORM=1.0/float(NBIN)
11
12       for I in get_range(NB,NBIN):
13         XXD=0.0
14         DD=0.0
15         K=0
16         for J in range(NBIN-2):
17             IJ=I+J
18             if IJ<=NB:
19                 XXD=XXD+WX[IJ]
20                 if WE[IJ] > SMALL: # only include non-zero errors
21                     K=K+1
22                     DD=DD+WY[IJ]
23
24         XB[I] = BNORM*XXD
25         YB[I] = 0.0
26         if K>0:
27             YB[I] = BNORM*DD
28       return XB, YB
```

Science and Technology Facilities Council

Fail Fast

- Sometimes had –1's in the for loops or inputs that assumed counting started at 1
- This became very confusing to know when the indices needed shifting

```
C     ------------------------------------------------------------
      SUBROUTINE DATIN1
      INCLUDE 'res_par.f90'
      INCLUDE 'mod_files.f90'
      INCLUDE 'options.f90'
      COMMON /DATCOM/ XDAT(m_d),DAT(m_d),SIG(m_d),NDAT
      COMMON/ModPars/NBIN,IMIN,IMAX,RSCL,BNORM
      SMALL=1.0E-20
      if (ABS(RSCL-1.0).GT.0.01)then
       OPEN(UNIT=53,FILE=lptfile,STATUS='old',FORM='formatted',
     1 access='append')
       WRITE(53,*) ' DATIN1; Data error-bars multiplied by: ',RSCL
       close(unit=53)
      endif
      RSCL=RSCL*RSCL
      N=0
      do I=IMIN,IMAX,NBIN
       N=N+1
       XXD=0.0
       DD=0.0
       EE=0.0
       K=0
       do J=0,NBIN-1
        XXD=XXD+XDAT(I+J)
        IF (SIG(I+J).GT.SMALL) THEN
         K=K+1
         DD=DD+DAT(I+J)
         EE=EE+SIG(I+J)
        ENDIF
       end do
       XDAT(N)=BNORM*XXD
       IF (K.GT.0) THEN
        DAT(N)=BNORM*DD
        SIG(N)=2.0*FLOAT(K*K)/(EE*RSCL)
       ELSE
        DAT(N)=0.0
        SIG(N)=0.0
       ENDIF
      end do
      NDAT=N
      END
```

Science and Technology Facilities Council

Fail Fast

- Sometimes had –1's in the for loops or inputs that assumed counting started at 1
- This became very confusing to know when the indices needed shifting
- Best to "cut losses" and use a method that prevents the problem

```
def DATIN1(XDAT, DAT, SIG, NDAT, NBIN, IMIN, IMAX, RSCL, BNORM, lptfile):
    SMALL = 1.0E-20

    if abs(RSCL - 1.0) > 0.01:
        path = os.path.realpath(__file__)
        path = os.path.join(path, lptfile)
        new_file = open(path, "a")
        new_file.write(f' DATIN1; Data error-bars multiplied by: RSCL')
        new_file.close()

    RSCL = pow(RSCL, 2)
    N = 0
    for II in range(IMIN - 1, IMAX - 1, NBIN):

        N += 1
        XXD = 0.0
        DD = 0.0
        EE = 0.0
        K = 0
        for J in range(0-1, NBIN - 2):
            XXD += XDAT[II + J]
            if SIG[II + J] > SMALL:
                K = K + 1
                DD += DAT[II + J]
                EE += SIG[II + J]

        if K > 0:
            DAT[N-1] = BNORM * DD
            SIG[N-1] = (2.0 * float(K * K)) /
                 (EE * RSCL))

        else:
            DAT[N-1] = 0.0
            SIG[N-1] = 0.0

        XDAT[N-1] = BNORM * XXD

    NDAT = N
    return RSCL, DAT, SIG, XDAT, NDAT
```

Science and Technology Facilities Council

Iterative improvements

- Kept as much the same as possible compared to Fortran
- Created a set of classes to make Python look like Fortran (hide the –1's):
  - Vector class (vec)
  - 2D matrix
  - A wrapper for ranges get_range
- Added more comments
- Compared outputs with Fortran

```python
def DATIN1(XDAT, DAT, SIG, NDAT, NBIN, IMIN, IMAX, RSCL, BNORM, lptfile):
    SMALL = 1.0E-20

    if abs(RSCL - 1.0) > 0.01:
        path = os.path.realpath(__file__)
        path = os.path.join(path, lptfile)
        new_file = open(path, "a")
        new_file.write(f' DATIN1; Data error-bars multiplied by: RSCL')
        new_file.close()

    RSCL = pow(RSCL, 2)
    N = 0
    # loop over bins
    for II in get_range(IMIN, IMAX, NBIN):
        N += 1
        XXD = 0.0
        DD = 0.0
        EE = 0.0
        K = 0
        for J in get_range(0, NBIN - 1):
            XXD += XDAT(II + J)
            if SIG(II + J) > SMALL:
                K = K + 1
                DD += DAT(II + J)
                EE += SIG(II + J)

        # update values
        if K > 0:
            DAT(N) = BNORM * DD
            SIG(N) = (2.0 * float(K * K)) /
                 (EE * RSCL))

        else:
            DAT(N) = 0.0
            SIG(N) = 0.0

        XDAT(N) = BNORM * XXD

    NDAT = N
    return RSCL, DAT, SIG, XDAT, NDAT
```

Science and Technology Facilities Council

Iterative improvements

- Globals led to very long argument list

```
def DATIN1(XDAT, DAT, SIG, NDAT, NBIN, IMIN, IMAX, RSCL, BNORM, lptfile):
    SMALL = 1.0E-20

    if abs(RSCL - 1.0) > 0.01:
        path = os.path.realpath(__file__)
        path = os.path.join(path, lptfile)
        new_file = open(path, "a")
        new_file.write(f' DATIN1; Data error-bars multiplied by: RSCL')
        new_file.close()

    RSCL = pow(RSCL, 2)
    N = 0
    # loop over bins
    for II in get_range(IMIN, IMAX, NBIN):
        N += 1
        XXD = 0.0
        DD = 0.0
        EE = 0.0
        K = 0
        for J in get_range(0, NBIN - 1):
            XXD += XDAT(II + J)
            if SIG(II + J) > SMALL:
                K = K + 1
                DD += DAT(II + J)
                EE += SIG(II + J)

        # update values
        if K > 0:
            DAT(N) = BNORM * DD
            SIG(N) = (2.0 * float(K * K)) /
                 (EE * RSCL))

        else:
            DAT(N) = 0.0
            SIG(N) = 0.0

        XDAT(N) = BNORM * XXD

    NDAT = N
    return RSCL, DAT, SIG, XDAT, NDAT
```

Science and Technology Facilities Council

Iterative improvements

- Globals led to very long argument list
  - Created a class to hold the globals

```
def DATIN1(COMS, lptfile):
    SMALL = 1.0E-20

    if abs(COMS["Params"].RSCL - 1.0) > 0.01:
        path = os.path.realpath(__file__)
        path = os.path.join(path, lptfile)
        new_file = open(path, "a")
        new_file.write(f' DATIN1; Data error-bars multiplied by: COMS["Params"].RSCL')
        new_file.close()

    COMS["Params"].RSCL = pow(COMS["Params"].RSCL, 2)
    N = 0  # here N is a counter and not the total number
    # loop over original bins
    for II in get_range(
            COMS["Params"].IMIN,
            COMS["Params"].IMAX,
            COMS["Params"].NBIN):

        N += 1
        XXD = 0.0
        DD = 0.0
        EE = 0.0
        K = 0
        # get values from rebinned data
        for J in get_range(0, COMS["Params"].NBIN - 1):
            XXD += COMS["DATA"].XDAT(II + J)
            if COMS["DATA"].SIG(II + J) > SMALL:
                K = K + 1
                DD += COMS["DATA"].DAT(II + J)
                EE += COMS["DATA"].SIG(II + J)

        if K > 0:

            DD = COMS["Params"].BNORM * DD
            EE = round_sig(
                round_sig(2.0 * float(K * K)) /
                round_sig(EE *
                          COMS["Params"].RSCL))
        else:
            DD = 0.0
            EE = 0.0

        COMS["DATA"].DAT.set(N, DD)
        COMS["DATA"].SIG.set(N, EE)
        COMS["DATA"].XDAT.set(N, COMS["Params"].BNORM * XXD)

    COMS["DATA"].NDAT = N
```

Science and Technology Facilities Council

Iterative improvements

- Globals led to very long argument list
  - Created a class to hold the globals
- Fortran IO is very different to Python's

```python
def DATIN1(COMS, lptfile):
    SMALL = 1.0E-20

    if abs(COMS["Params"].RSCL - 1.0) > 0.01:
        path = os.path.realpath(__file__)
        path = os.path.join(path, lptfile)
        new_file = open(path, "a")
        new_file.write(f' DATIN1; Data error-bars multiplied by: COMS["Params"].RSCL')
        new_file.close()

    COMS["Params"].RSCL = pow(COMS["Params"].RSCL, 2)
    N = 0  # here N is a counter and not the total number
    # loop over original bins
    for II in get_range(
            COMS["Params"].IMIN,
            COMS["Params"].IMAX,
            COMS["Params"].NBIN):

        N += 1
        XXD = 0.0
        DD = 0.0
        EE = 0.0
        K = 0
        # get values from rebinned data
        for J in get_range(0, COMS["Params"].NBIN - 1):
            XXD += COMS["DATA"].XDAT(II + J)
            if COMS["DATA"].SIG(II + J) > SMALL:
                K = K + 1
                DD += COMS["DATA"].DAT(II + J)
                EE += COMS["DATA"].SIG(II + J)

        if K > 0:

            DD = COMS["Params"].BNORM * DD
            EE = round_sig(
                round_sig(2.0 * float(K * K)) /
                round_sig(EE *
                          COMS["Params"].RSCL))
        else:
            DD = 0.0
            EE = 0.0

        COMS["DATA"].DAT.set(N, DD)
        COMS["DATA"].SIG.set(N, EE)
        COMS["DATA"].XDAT.set(N, COMS["Params"].BNORM * XXD)

    COMS["DATA"].NDAT = N
```

Science and Technology Facilities Council

Iterative improvements

- Globals led to very long argument list
  - Created a class to hold the globals

- Fortran IO is very different to Python's
  - Wrote a class to behave like Fortran IO

```python
def DATIN1(COMS, store, lptfile):
    SMALL = 1.0E-20

    if abs(COMS["Params"].RSCL - 1.0) > 0.01:
        store.open(53, lptfile)
        store.write(
            53,
            f' DATIN1; Data error-bars multiplied by: {COMS["Params"].RSCL}')
        store.close(unit=53)

    COMS["Params"].RSCL = pow(COMS["Params"].RSCL, 2)
    N = 0  # here N is a counter and not the total number
    # loop over original bins
    for II in get_range(
            COMS["Params"].IMIN,
            COMS["Params"].IMAX,
            COMS["Params"].NBIN):

        N += 1
        XXD = 0.0
        DD = 0.0
        EE = 0.0
        K = 0
        # get values from rebinned data
        for J in get_range(0, COMS["Params"].NBIN - 1):
            XXD += COMS["DATA"].XDAT(II + J)
            if COMS["DATA"].SIG(II + J) > SMALL:
                K = K + 1
                DD += COMS["DATA"].DAT(II + J)
                EE += COMS["DATA"].SIG(II + J)

        if K > 0:

            DD = COMS["Params"].BNORM * DD
            EE = round_sig(
                round_sig(2.0 * float(K * K)) /
                round_sig(EE *
                          COMS["Params"].RSCL))
        else:
            DD = 0.0
            EE = 0.0

        COMS["DATA"].DAT.set(N, DD)
        COMS["DATA"].SIG.set(N, EE)
        COMS["DATA"].XDAT.set(N, COMS["Params"].BNORM * XXD)

    COMS["DATA"].NDAT = N
```

Science and Technology Facilities Council

Iterative improvements

- Globals led to very long argument list
  - Created a class to hold the globals
- Fortran IO is very different to Python's
  - Wrote a class to behave like Fortran IO
- Fortran can silently convert matrices into vectors

```
def DATIN1(COMS, store, lptfile):
    SMALL = 1.0E-20

    if abs(COMS["Params"].RSCL - 1.0) > 0.01:
        store.open(53, lptfile)
        store.write(
            53,
            f' DATIN1; Data error-bars multiplied by: {COMS["Params"].RSCL}')
        store.close(unit=53)

    COMS["Params"].RSCL = pow(COMS["Params"].RSCL, 2)
    N = 0  # here N is a counter and not the total number
    # loop over original bins
    for II in get_range(
            COMS["Params"].IMIN,
            COMS["Params"].IMAX,
            COMS["Params"].NBIN):

        N += 1
        XXD = 0.0
        DD = 0.0
        EE = 0.0
        K = 0
        # get values from rebinned data
        for J in get_range(0, COMS["Params"].NBIN - 1):
            XXD += COMS["DATA"].XDAT(II + J)
            if COMS["DATA"].SIG(II + J) > SMALL:
                K = K + 1
                DD += COMS["DATA"].DAT(II + J)
                EE += COMS["DATA"].SIG(II + J)

        if K > 0:

            DD = COMS["Params"].BNORM * DD
            EE = round_sig(
                round_sig(2.0 * float(K * K)) /
                round_sig(EE *
                          COMS["Params"].RSCL))
        else:
            DD = 0.0
            EE = 0.0

        COMS["DATA"].DAT.set(N, DD)
        COMS["DATA"].SIG.set(N, EE)
        COMS["DATA"].XDAT.set(N, COMS["Params"].BNORM * XXD)

    COMS["DATA"].NDAT = N
```

Science and Technology Facilities Council

Iterative improvements

- Globals led to very long argument list
  - Created a class to hold the globals
- Fortran IO is very different to Python's
  - Wrote a class to behave like Fortran IO
- Fortran can silently convert matrices into vectors
  - Re-worked matrix class to be a vector under the hood

```python
def DATIN1(COMS, store, lptfile):
    SMALL = 1.0E-20

    if abs(COMS["Params"].RSCL - 1.0) > 0.01:
        store.open(53, lptfile)
        store.write(
            53,
            f' DATIN1; Data error-bars multiplied by: {COMS["Params"].RSCL}')
        store.close(unit=53)

    COMS["Params"].RSCL = pow(COMS["Params"].RSCL, 2)
    N = 0  # here N is a counter and not the total number
    # loop over original bins
    for II in get_range(
            COMS["Params"].IMIN,
            COMS["Params"].IMAX,
            COMS["Params"].NBIN):

        N += 1
        XXD = 0.0 #  x bins?
        DD = 0.0 #  y values in bin?
        EE = 0.0 # errors?
        K = 0
        # get values from rebinned data
        for J in get_range(0, COMS["Params"].NBIN - 1):
            XXD += COMS["DATA"].XDAT(II + J)
            if COMS["DATA"].SIG(II + J) > SMALL:
                K = K + 1
                DD += COMS["DATA"].DAT(II + J)
                EE += COMS["DATA"].SIG(II + J)

        if K > 0:
            # round sig needed for stability
            DD = COMS["Params"].BNORM * DD
            EE = round_sig(
                round_sig(2.0 * float(K * K)) /
                round_sig(EE *
                          COMS["Params"].RSCL))
        else:
            DD = 0.0
            EE = 0.0

        COMS["DATA"].DAT.set(N, DD)
        COMS["DATA"].SIG.set(N, EE)
        COMS["DATA"].XDAT.set(N, COMS["Params"].BNORM * XXD)

    COMS["DATA"].NDAT = N
```

Science and Technology Facilities Council

Iterative improvements

- Globals led to very long argument list
  - Created a class to hold the globals
- Fortran IO is very different to Python's
  - Wrote a class to behave like Fortran IO
- Fortran can silently convert matrices into vectors
  - Re-worked matrix class to be a vector under the hood
- Fortran will silently convert between complex and real arrays

```python
def DATIN1(COMS, store, lptfile):
    SMALL = 1.0E-20

    if abs(COMS["Params"].RSCL - 1.0) > 0.01:
        store.open(53, lptfile)
        store.write(
            53,
            f' DATIN1; Data error-bars multiplied by: {COMS["Params"].RSCL}')
        store.close(unit=53)

    COMS["Params"].RSCL = pow(COMS["Params"].RSCL, 2)
    N = 0  # here N is a counter and not the total number
    # loop over original bins
    for II in get_range(
            COMS["Params"].IMIN,
            COMS["Params"].IMAX,
            COMS["Params"].NBIN):

        N += 1
        XXD = 0.0 #  x bins?
        DD = 0.0 #  y values in bin?
        EE = 0.0 # errors?
        K = 0
        # get values from rebinned data
        for J in get_range(0, COMS["Params"].NBIN - 1):
            XXD += COMS["DATA"].XDAT(II + J)
            if COMS["DATA"].SIG(II + J) > SMALL:
                K = K + 1
                DD += COMS["DATA"].DAT(II + J)
                EE += COMS["DATA"].SIG(II + J)

        if K > 0:
            # round sig needed for stability
            DD = COMS["Params"].BNORM * DD
            EE = round_sig(
                round_sig(2.0 * float(K * K)) /
                round_sig(EE *
                          COMS["Params"].RSCL))
        else:
            DD = 0.0
            EE = 0.0

        COMS["DATA"].DAT.set(N, DD)
        COMS["DATA"].SIG.set(N, EE)
        COMS["DATA"].XDAT.set(N, COMS["Params"].BNORM * XXD)

    COMS["DATA"].NDAT = N
```

Science and Technology Facilities Council

Iterative improvements

- Globals led to very long argument list
  - Created a class to hold the globals
- Fortran IO is very different to Python's
  - Wrote a class to behave like Fortran IO
- Fortran can silently convert matrices into vectors
  - Re-worked matrix class to be a vector under the hood
- Fortran will silently convert between complex and real arrays
  - Wrote flatten and compress methods to replicate behaviour

```python
def DATIN1(COMS, store, lptfile):
    SMALL = 1.0E-20

    if abs(COMS["Params"].RSCL - 1.0) > 0.01:
        store.open(53, lptfile)
        store.write(
            53,
            f' DATIN1; Data error-bars multiplied by: {COMS["Params"].RSCL}')
        store.close(unit=53)

    COMS["Params"].RSCL = pow(COMS["Params"].RSCL, 2)
    N = 0  # here N is a counter and not the total number
    # loop over original bins
    for II in get_range(
            COMS["Params"].IMIN,
            COMS["Params"].IMAX,
            COMS["Params"].NBIN):

        N += 1
        XXD = 0.0 #  x bins?
        DD = 0.0 #  y values in bin?
        EE = 0.0 # errors?
        K = 0
        # get values from rebinned data
        for J in get_range(0, COMS["Params"].NBIN - 1):
            XXD += COMS["DATA"].XDAT(II + J)
            if COMS["DATA"].SIG(II + J) > SMALL:
                K = K + 1
                DD += COMS["DATA"].DAT(II + J)
                EE += COMS["DATA"].SIG(II + J)

        if K > 0:
            # round sig needed for stability
            DD = COMS["Params"].BNORM * DD
            EE = round_sig(
                round_sig(2.0 * float(K * K)) /
                round_sig(EE *
                          COMS["Params"].RSCL))
        else:
            DD = 0.0
            EE = 0.0

        COMS["DATA"].DAT.set(N, DD)
        COMS["DATA"].SIG.set(N, EE)
        COMS["DATA"].XDAT.set(N, COMS["Params"].BNORM * XXD)

    COMS["DATA"].NDAT = N
```

Science and Technology Facilities Council

Iterative improvements

- Once I had a good understanding, could use better names
- Didn't delete old functions
  - Added a deprecation warning decorator
  - Call the new function
- Code is much easier to read and follow

```python
def bin_and_filter_sample_data(COMS, store, lptfile):
    SMALL = 1.0E-20

    if abs(COMS["Params"].RSCL - 1.0) > 0.01:
        store.open(53, lptfile)
        store.write(
            53,
            f' DATIN1; Data error-bars multiplied by: {COMS["Params"].RSCL}')
        store.close(unit=53)

    COMS["Params"].RSCL = pow(COMS["Params"].RSCL, 2)
    N = 0  # here N is a counter and not the total number
    # lopp over original bins
    for II in get_range(
            COMS["Params"].IMIN,
            COMS["Params"].IMAX,
            COMS["Params"].NBIN):

        N += 1
        bin_width = 0.0
        y_data_in_bin = 0.0
        sigma_data_in_bin = 0.0
        K = 0
        # get values from rebinned data
        for J in get_range(0, COMS["Params"].NBIN - 1):
            bin_width += COMS["DATA"].XDAT(II + J)
            if COMS["DATA"].SIG(II + J) > SMALL:
                K = K + 1
                y_data_in_bin += COMS["DATA"].DAT(II + J)
                sigma_data_in_bin += COMS["DATA"].SIG(II + J)

        if K > 0:
            # if large sigma(s) exist in bin assume it dominates -> throw away
            # everyting else
            y_data_in_bin = COMS["Params"].BNORM * y_data_in_bin
            sigma_data_in_bin = round_sig(
                round_sig(2.0 * float(K * K)) /
                round_sig(sigma_data_in_bin *
                          COMS["Params"].RSCL))
        else:
            y_data_in_bin = 0.0
            sigma_data_in_bin = 0.0
        # store the scaled value for the new bin
        COMS["DATA"].DAT.set(N, y_data_in_bin)
        COMS["DATA"].SIG.set(N, sigma_data_in_bin)
        COMS["DATA"].XDAT.set(N, COMS["Params"].BNORM * bin_width)

    COMS["DATA"].NDAT = N


@deprecated
def DATIN1(COMS, store, lptfile):
    bin_and_filter_sample_data(COMS, store, lptfile)
```

UKRI Science and Technology Facilities Council

Iterative improvements

- A more complex example
- What are some of the variables (e.g. XBMIN)
- Lots of globals
- What are the subroutines XGINIT and BINBLR
- Continues and go to's

```
      SUBROUTINE BLRINT(XB,YB,NB,IREAD,IDUF)
      INCLUDE 'res_par.f90'
      INCLUDE 'mod_files.f90'
      COMMON /FFTCOM/ FRES(m_d2),FWRK(m_d2),XJ(m_d),TWOPIK(m_d1),NFFT
      COMMON /FITCOM/ FIT(m_d),RESID(m_d),NFEW,FITP(m_p),EXPF(m_d1,6)
      COMMON /DATCOM/ XDAT(m_d),DAT(m_d),SIG(m_d),NDAT
      COMMON/ModRes/ntr,xres,yres,eres,nrbin,ermin,ermax
      REAL XB(m_d),YB(m_d),DER2(m_d),xr(m_d),yr(m_d),er(m_d)
      real xres(m_d),yres(m_d),eres(m_d)
      LOGICAL  LSTART
      DATA     LSTART /.FALSE./
      if(IREAD.eq.0)then
       LSTART=.FALSE.
      else
       LSTART=.TRUE.
      endif
      SMALL=1.0E-20
      NFFT=m_d
      DO I=1,NB
       xr(I)=xres(I)
       yr(I)=yres(I)
       er(I)=eres(I)
      END DO
      CALL BINBLR(xr,yr,er,NB,XB,YB,NRBIN)
      XDMIN=XDAT(1)
      XDMAX=XDAT(NDAT)
      YMAX=0.0
      YSUM=0.0
      DO 10 I=1,NB
        IF (XB(I).LT.XDMIN .OR. XB(I).GT.XDMAX) GOTO 10
        IF (YB(I).GT.YMAX) YMAX=YB(I)
        YSUM=YSUM+YB(I)
   10 CONTINUE
      IF (YSUM.LT.SMALL) THEN
       OPEN(UNIT=53,FILE=lptfile,STATUS='old',FORM='formatted',
     1 access='append')
       write(53,1000)
 1000  format(' Ysum is too small')
       IDUF=1
       close(unit=53)
       RETURN
      ENDIF
      CALL XGINIT(XB,YB,NB,YMAX,XBMIN,XBMAX,LSTART)
```

Science and Technology Facilities Council

Iterative improvements

- A more complex example
- What are some of the variables (e.g. XBMIN)
- Lots of globals
- What are the subroutines XGINIT and BINBLR
- Continues and go to's

```
def bin_resolution(N_bin, IREAD, IDUF, COMS, store, lptfile):
    LSTART = True
    if IREAD == 0:
        LSTART = False

    SMALL = 1.0E-20
    COMS["FFT"].NFFT = m_d
    COMS["res_data"].N_FT = m_d // 2
    # copy resolution data
    xr = Vec(m_d)
    yr = Vec(m_d)
    er = Vec(m_d)
    xr.copy(COMS["Res"].xres.output_range(1, N_bin))
    yr.copy(COMS["Res"].yres.output_range(1, N_bin))
    er.copy(COMS["Res"].eres.output_range(1, N_bin))

    # rebin the resolution data as it is not on an evenly spaced grid
    x_bin, y_bin = rebin(xr, yr, er, N_bin, COMS["Res"].nrbin)

    # sample data x range
    XDMIN = COMS["DATA"].XDAT(1)
    XDMAX = COMS["DATA"].XDAT(COMS["DATA"].NDAT)

    # get indicies for valid x range
    first_index = np.argmax(x_bin.output() >= XDMIN) + 1
    last_index = np.argmin(x_bin.output() <= XDMAX) + 1

    # get total and max Y binned values within valid x range
    YSUM = np.sum(y_bin.output_range(first_index, last_index))
    YMAX = np.max(y_bin.output_range(first_index, last_index))

    if YSUM < SMALL:
        store.open(53, lptfile)
        store.write(53, ' Ysum is too small')
        IDUF = 1
        store.close(unit=53)
        return x_bin, y_bin, IDUF

    XBMIN, XBMAX, y_bin = rm_BG(
        x_bin, y_bin, N_bin, YMAX, LSTART, COMS, store, lptfile)
```

Science and Technology Facilities Council

Useful tools: Cython

- Good profiling tools
- Easy to use

```
1 import pstats, cProfile
2 from ql_data import _QLdata
3
4 # cython: profile=True
5 def QLdata(numb,x_in,y_in,e_in,reals,opft,XD_in,X_r,Y_r,E_r,Wy_in,We_in,sfile,rfile,l_fn, overwrite=True):
6
7     cProfile.runctx("_QLdata(numb,x_in,y_in,e_in,reals,opft,XD_in,X_r,Y_r,E_r,Wy_in,We_in,sfile,rfile,l_fn, overwrite)",
8                     globals(), locals(), "Profile.prof")
9
10    s = pstats.Stats("Profile.prof")
11    s.strip_dirs().sort_stats("time").print_stats()
12
```

Useful tools: Cython

- Good profiling tools
- tottime is the total time, excluding function calls
- Number of calls

```
(quasielasticbayes-dev) PS C:\Users\BTR75544\work\quasi_2\quasielasticbayes> python ..\moo.py
3.14168616 314168616.0
Mon Sep 19 09:18:39 2022    Profile.prof

         678542160 function calls in 106.893 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1   30.453   30.453  105.439  105.439 moo.py:25(_QLdata)
        1   26.115   26.115   44.999   44.999 moo.py:11(gen_data)
100000000   15.153    0.000   23.186    0.000 moo.py:6(radius)
200000000    9.634    0.000    9.634    0.000 {method 'append' of 'list' objects}
200000000    9.250    0.000    9.250    0.000 {method 'random' of '_random.Random' objects}
100000000    8.033    0.000    8.033    0.000 {built-in method math.sqrt}
 78542154    6.802    0.000    6.802    0.000 moo.py:21(add_to_pi)
        1    1.454    1.454  106.893  106.893 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000  106.893  106.893 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Useful tools: Cython

- Good profiling tools
- Python library that allows pre-compiled pseudo-c code
- Can give feedback about where in the code is slow (Python)
- Relatively easy to learn

Generated by Cython 0.29.30

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: bayes_C.c

```
+01: # Mantid Repository : https://github.com/mantidproject/mantid
 02: #
 03: # Copyright &copy; 2022 ISIS Rutherford Appleton Laboratory UKRI,
 04: #   NScD Oak Ridge National Laboratory, European Spallation Source,
 05: #   Institut Laue - Langevin & CSNS, Institute of High Energy Physics, CAS
 06: # SPDX - License - Identifier: GPL - 3.0 +
+07: from quasielasticbayes.python.fortran_python import round_sig
 08:
+09: from numpy import zeros
 10: cimport numpy as np
 11: # It's necessary to call "import_array" if you use any part of the
 12: # numpy PyArray_* API. From Cython 3, accessing attributes like
 13: # ".shape" on a typed Numpy array use this API. Therefore we recommend
 14: # always calling "import_array" whenever you "cimport numpy"
+15: np.import_array()
 16:
 17: # shift bin values onto new grid
+18: def bin_shift_vecs(np.ndarray[np.float_t] y_grid,int N, np.ndarray[np.float_t] index_vec, np.ndarray[np.float_t] XPDAT): # is this
+19:       cdef np.ndarray[np.float_t] y_shifted = zeros(N)
 20:       cdef int I, J
 21:       cdef float fractional_x_shift
 22:
+23:       for I in range(N):
+24:         J=int(index_vec[I]) # get the index that says where the shift value is
+25:         fractional_x_shift = XPDAT[I]
 26:         # get fractions of original bins in the new shifted bin add sum (e.g fractional_x_shift = 0.2)
+27:         y_shifted[I] = y_grid[J-1]*(1-fractional_x_shift) + fractional_x_shift*y_grid[J]
 28:
+29:       return y_shifted
 30:
+31: def HESS0_calc(np.ndarray[np.float_t] RESID, np.ndarray[np.float_t] DDDPAR, int N):
+32:       cdef float SM = 0.0
+33:       cdef int magic = 6
 34:       cdef int kk
+35:       for kk in range(N):
+36:           SM = round_sig(SM,magic) + round_sig(round_sig(RESID[kk],magic)*round_sig(DDDPAR[kk],magic),magic)
+37:       return SM
```

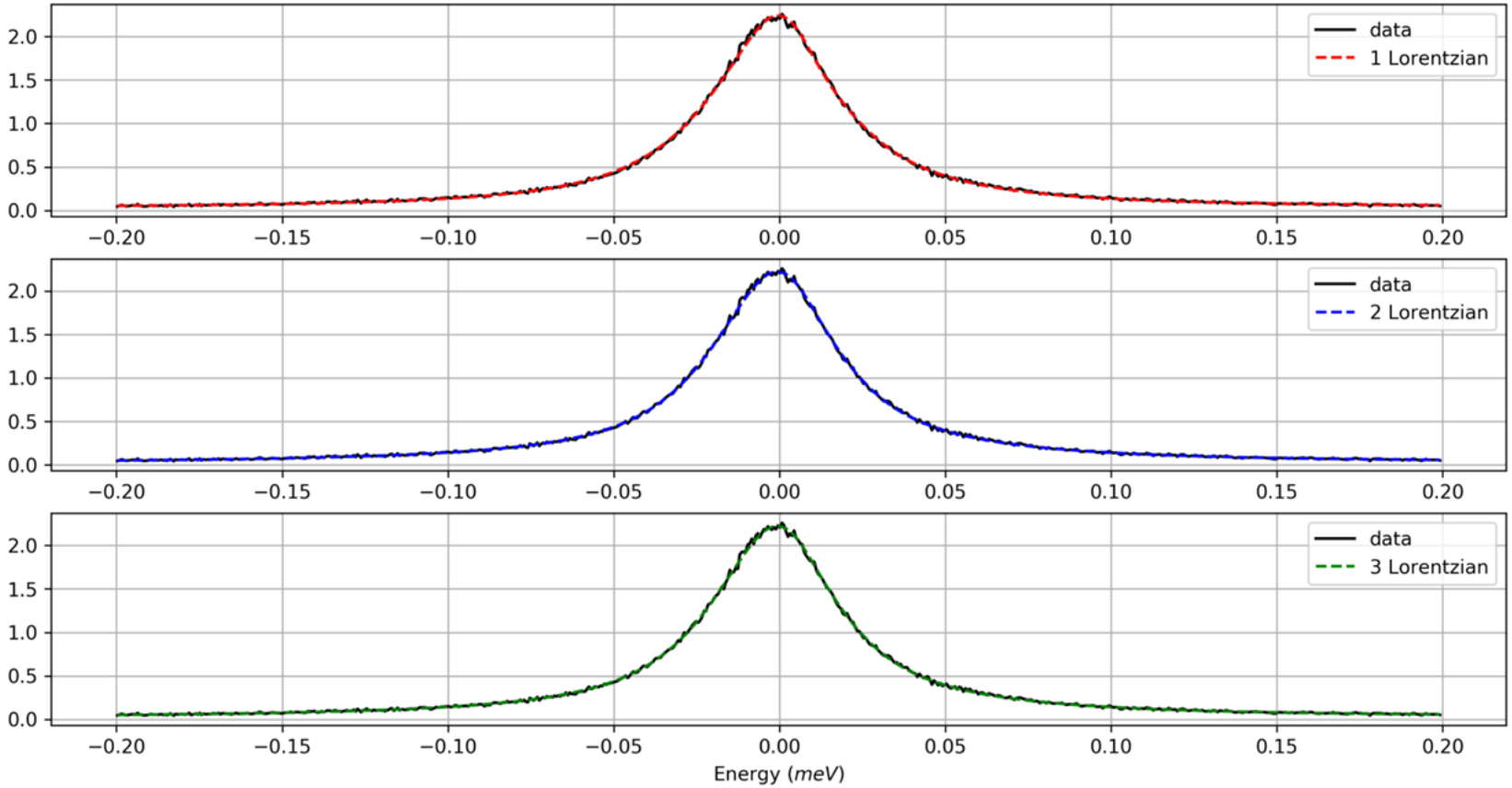
Science and Technology Facilities Council

Conclusion

- Can be difficult and time consuming working on legacy code
- When working with legacy code its important to be open to failing fast
- Iterative improvements are a great way to translate the code
  - Easier if minimal things change
  - Gain more understanding from context
  - Know whats coming
- Needed understanding of this code, solve the compiler problems and understand the OS dependence
- Cython can be used to help profile and speed up Python code

Links

- https://github.com/mantidproject/quasielasticbayes



Science and Technology Facilities Council

UKRI Science and Technology Facilities Council

Thank you

Science and Technology Facilities Council @STFC_matters Science and Technology Facilities Council