```
    typedef double XFLOAT;
    typedef double OTA_FLOAT;

const int kNarrowBand     = 0;
const int kWideBand       = 1;
const int kSuperWideBand  = 2;

namespace POLQAV2
{

void SlidingWinMean(CPOLQAData *PolqaHandle, XFLOAT const *in, int oddWinlen, XFLOAT
*out, int len);
void SlidingWinPowNorm(CPOLQAData *POLQAHandle,
                       XFLOAT const *In, XFLOAT *Out,
                       int len,
                       XFLOAT thresh,
                       int WINLEN,
                       XFLOAT meanOutLevel,
                       bool doAvoidShortBursts,
                       int const hystLenInSamples);
void WarpSpectrum(XFLOAT const *OrigSpec, XFLOAT *WarpedSpec, XFLOAT WarpingFac, int
NumBands);

void CreateArrayFromCSignal(double*** pppArray, CSignal* pSignal)
{
    double** ppArray;
    *pppArray = (double**)matMalloc(pSignal->aNumberOfWindows * sizeof(double*));
    ppArray = *pppArray;

    for (int f=0; f<pSignal->aNumberOfWindows; f++)
    {
        ppArray[f] = (double*)matMalloc(pSignal->aNumberOfBands * sizeof(double));
        for (int b=0; b<pSignal->aNumberOfBands; b++)
            ppArray[f][b] = pSignal->m_pData[f][b];
    }
}

XFLOAT CPairParameters::ComputePitchRatio()
{
    int NumPitchFrames=0;
    XFLOAT PitchRatio=1.0;
    XFLOAT Sum=0, MaxVal=0;

    if (mPitchFreqRef==0.0)
        return PitchRatio;

    {
        PitchRatio = 1.0;
        mNumVoicedFrames = 0;
        mNumVoicedFramesRef = 0;
        mNumVoicedFramesDeg = 0;

        XFLOAT  BinWidth = 0.02 * mPitchFreqRef;

        SmartBufferPolqa SB_PitchHistogram(POLQAHandle, 51);
        XFLOAT* PitchHistogram = SB_PitchHistogram.Buffer;

        matbZero(PitchHistogram, 51);

        for (int frameIndex = 0; frameIndex <= statics->stopFrameIdx; frameIndex++)
        {
            if (mpPitchVec[frameIndex]>0)
                mNumVoicedFramesRef++;
            if (mpPitchVecDeg[frameIndex]>0)
                mNumVoicedFramesDeg++;
            if (mpPitchVec[frameIndex]>0 && mpPitchVecDeg[frameIndex]>0)
            {
                XFLOAT Diff = mpPitchVec[frameIndex] - mpPitchVecDeg[frameIndex];

                int PitchBin = (int)(Diff/BinWidth + 1e-12 + 25.0);

                if (PitchBin<51 && PitchBin>=0)
                {
                    PitchHistogram[PitchBin]++;
```

```
                    NumPitchFrames++;
                }
            }
        }
        ;
        mNumVoicedFrames = NumPitchFrames;

        if (NumPitchFrames>10)

        {
            XFLOAT MaxVal = -1.0;
            int MaxPos = -1;

            MaxVal = matMaxExt(PitchHistogram + 1, 51 - 1, &MaxPos);
            MaxPos += 1;

            if (MaxPos >= 1 && MaxPos<51-1)
            {
                Sum = PitchHistogram[MaxPos];

                {
                    Sum += PitchHistogram[MaxPos-1];
                    Sum += PitchHistogram[MaxPos+1];
                    MaxVal = Sum / 3.0;

                    if (Sum > 0.3*NumPitchFrames)

                    {
                        XFLOAT AvgDiff;
                        AvgDiff =   ((XFLOAT)MaxPos-1)*PitchHistogram[MaxPos-1] +
                                    ((XFLOAT)MaxPos)*PitchHistogram[MaxPos]+
                                    ((XFLOAT)MaxPos+1)*PitchHistogram[MaxPos+1];
                        AvgDiff = (AvgDiff / Sum - 25.0) * BinWidth;
                        PitchRatio = mPitchFreqRef / (mPitchFreqRef + AvgDiff);

                    }
                }

            }
        }

    }

    if ((PitchRatio<1.01)&&(PitchRatio>0.99))
        PitchRatio = 1.0;

    return PitchRatio;
}

void CreateAlignTimeSeries(CPOLQAData *POLQAHandle, const CTimeSeries &InputTimeSeries,
const CIntArray &aStartSampleUtterance, const CIntArray &aStopSampleUtterance, const
CIntArray &aDelayUtterance, const int frameLength, CTimeSeries &AlignedTimeSeries)
{

    int BlendLen=frameLength/32;
    XFLOAT SinTab[(2048/32)];
    XFLOAT CosTab[(2048/32)];
    XFLOAT Step = 3.1415/(2*(BlendLen-1));
    for (int i=0; i<BlendLen; i++)
    {
        SinTab[i] = sin(i*Step);
        CosTab[i] = cos(i*Step);
        SinTab[i] = SinTab[i] * SinTab[i];
        CosTab[i] = CosTab[i] * CosTab[i];
    }

    int LastStartFrameInputSample =  0;
    int LastStartFrameAlignedSample = 0;
    int LastDelay = 0;
    bool FirstFrame = true;
    int AlignedSequenceLength = AlignedTimeSeries.GetLength();
    int LastInputFrame = InputTimeSeries.GetLength() - frameLength/2;
    const int stopFrameIdx = POLQAHandle->statics->stopFrameIdx;
    for(int frameIndex = POLQAHandle->statics->startFrameIdx; frameIndex <=
stopFrameIdx; frameIndex++)
    {
```

```
        int Utt = GetUtteranceForFrame(aStartSampleUtterance, aStopSampleUtterance,
aDelayUtterance, frameIndex, frameLength);
        int Delay = aDelayUtterance.m_pData[Utt];
        int StartFrameInputSample =  frameIndex * (int)(frameLength/2) + Delay;
        int StartFrameAlignedSample = frameIndex * (int)(frameLength/2);

        if (StartFrameAlignedSample>=0 && StartFrameAlignedSample <
AlignedSequenceLength-frameLength/2)
        {
            if (StartFrameInputSample >= 0 && StartFrameInputSample < LastInputFrame)
            {
                matbCopy(InputTimeSeries.m_pData + StartFrameInputSample,
AlignedTimeSeries.m_pData + StartFrameAlignedSample,
(int)frameLength/2);

                if (!FirstFrame && LastDelay!=Delay)
                {
                    int AlignedStart = StartFrameAlignedSample-BlendLen/2;
                    int SrcStart = StartFrameInputSample -BlendLen/2;
                    int LastStart = LastStartFrameInputSample + frameLength/2
-BlendLen/2;
                    for (int i=0; i<BlendLen; i++)
                        *(AlignedTimeSeries.m_pData+AlignedStart+i) = CosTab[i]*
*(InputTimeSeries.m_pData+LastStart+i) + SinTab[i] *
*(InputTimeSeries.m_pData+SrcStart+i);
                }

                LastStartFrameInputSample = StartFrameInputSample;
                LastStartFrameAlignedSample = StartFrameAlignedSample;
                LastDelay = Delay;
                FirstFrame = false;
            }
            else
                for (int i = StartFrameAlignedSample; i < StartFrameAlignedSample +
frameLength/2; i++)
                    AlignedTimeSeries.m_pData[i] = 0.0;
        }

    }
}

void CPairParameters::ShiftPitch(CHzSpectrum const *spectrumRef, CHzSpectrum const
*spectrumDeg, CHzSpectrum *spectrumDegCorrected, bool const *pActiveFrameFlags,
                                 int* bestShiftPerFrame, XFLOAT
*bestWarpingFacPerFrame)
{
    const XFLOAT freqRes = statics->aFrequencyResolutionHz;
    XFLOAT const FRAMES_PER_SEC = statics->sampleRate / (statics->frameLength/2);

    int    const SHIFT_PITCH_SMOOTHING_LEN     =
        (int)((XFLOAT)100.0 / freqRes + 0.5) | 0x1;
    int    const SHIFT_PITCH_MAX_PITCH_BIN     = (int)(1500.0 / freqRes + 0.5);
    int    const SHIFT_PITCH_MAX_SPEC_BIN      = (int)(3000.0 / freqRes + 0.5);
    XFLOAT const SHIFT_PITCH_MIN_CORR          = 0.8;

    int    const SHIFT_PITCH_FRAME_SEARCH_LEN  = (((1) > ((int)((XFLOAT)0.016 *
FRAMES_PER_SEC))) ? (1) : ((int)((XFLOAT)0.016 * FRAMES_PER_SEC)));

    XFLOAT const SHIFT_PITCH_MAX_PITCH_RATIO   = (XFLOAT)1.1;
    XFLOAT const SHIFT_PITCH_CORREL_SPL_THR    =
        20.0 / 10.0;

    XFLOAT const SHIFT_PITCH_CORREL_DAMPING    =
        0.08;
    XFLOAT  const SHIFT_PITCH_MAX_LOGFAC_CHANGE =
        fabs(log10((XFLOAT)0.96)) * (XFLOAT)62.5 / (XFLOAT)FRAMES_PER_SEC;

    const int NumBands = statics->aNumberOfHzBands;

    SmartBufferPolqa SB1(this->POLQAHandle, NumBands);
    SmartBufferPolqa SB2(this->POLQAHandle, NumBands);
    SmartBufferPolqa SB3(this->POLQAHandle, NumBands);
    SmartBufferPolqa SB4(this->POLQAHandle, NumBands);

    XFLOAT *SmoothedRefSpec     = SB1.Buffer;
    XFLOAT *SmoothedDegSpec     = SB2.Buffer;
```

```
    XFLOAT *OrigSmoothedDegSpec = SB3.Buffer;
    XFLOAT *WarpedDegSpec       = SB4.Buffer;

    XFLOAT MaxRefLev, MaxDegLev;
    XFLOAT MaxXCorr, CorrBefore, CorrAfter, BestCorr = -1.0, RefAutoCorr, DegAutoCorr;
    XFLOAT RefPitch, DegPitch, MinPitchRatio, MaxPitchRatio, BestWarpingFac = 1.0;
    int    RefStartBin, RefStopBin, DegStartBin, DegStopBin, StartBin, StopBin, BinLen,
           WarpedStartBin, WarpedStopBin, WarpedBinLen, MaxShiftLen;

    for (int i = statics->startFrameIdx; i <= statics->stopFrameIdx; i++)
    {
        bestShiftPerFrame     [i] = 0;
        bestWarpingFacPerFrame[i] = 1.0;

        MaxRefLev = log10(matMax(spectrumRef->m_pData[i]+1, NumBands-1) + 1e-10);
        MaxDegLev = log10(matMax(spectrumDeg->m_pData[i]+1, NumBands-1) + 1e-10);
        if (!pActiveFrameFlags[i] || MaxRefLev < SHIFT_PITCH_CORREL_SPL_THR ||
MaxDegLev < SHIFT_PITCH_CORREL_SPL_THR)
        {
            continue;
        }

        SlidingWinMean(this->POLQAHandle, spectrumRef->m_pData[i]+1,
SHIFT_PITCH_SMOOTHING_LEN, WarpedDegSpec+1, NumBands-1);
        matbThresh1(WarpedDegSpec+1, NumBands-1, (XFLOAT)0.0, MAT_LT);
        matbLog2(WarpedDegSpec+1, SmoothedRefSpec+1, NumBands-1);
        SlidingWinMean(this->POLQAHandle, spectrumDeg->m_pData[i]+1,
SHIFT_PITCH_SMOOTHING_LEN, WarpedDegSpec+1, NumBands-1);
        matbThresh1(WarpedDegSpec+1, NumBands-1, (XFLOAT)0.0, MAT_LT);
        matbLog2(WarpedDegSpec+1, SmoothedDegSpec+1, NumBands-1);

        CorrBefore = CorrAfter = 0.0;
        matbAdd1(-SHIFT_PITCH_CORREL_SPL_THR, SmoothedRefSpec+1, NumBands-1);
        matbAdd1(-SHIFT_PITCH_CORREL_SPL_THR, SmoothedDegSpec+1, NumBands-1);
        matbThresh1(SmoothedRefSpec+1, NumBands-1, 0.0, MAT_LT);
        matbThresh1(SmoothedDegSpec+1, NumBands-1, 0.0, MAT_LT);

        for (RefStartBin = 1; RefStartBin < NumBands-1 && SmoothedRefSpec[RefStartBin]
== 0.0; RefStartBin++);
        for (DegStartBin = 1; DegStartBin < NumBands-1 && SmoothedDegSpec[DegStartBin]
== 0.0; DegStartBin++);
        for (RefStopBin = NumBands-1; RefStopBin >= 0 && SmoothedRefSpec[RefStopBin] ==
0.0; RefStopBin--);
        for (DegStopBin = NumBands-1; DegStopBin >= 0 && SmoothedDegSpec[DegStopBin] ==
0.0; DegStopBin--);
        StartBin = (((RefStartBin) > (DegStartBin)) ? (RefStartBin) : (DegStartBin));
        StopBin  = (((RefStopBin) < ((((DegStopBin) < (SHIFT_PITCH_MAX_SPEC_BIN)) ?
(DegStopBin) : (SHIFT_PITCH_MAX_SPEC_BIN)))) ? (RefStopBin) : ((((DegStopBin) <
(SHIFT_PITCH_MAX_SPEC_BIN)) ? (DegStopBin) : (SHIFT_PITCH_MAX_SPEC_BIN))));
        BinLen   = StopBin - StartBin + 1;
        if (BinLen < (int)(1000.0 / freqRes + 0.5))
        {
            continue;
        }

        SlidingWinPowNorm(POLQAHandle, SmoothedRefSpec+1, WarpedDegSpec+1, NumBands-1,
0.0,
                          2*SHIFT_PITCH_SMOOTHING_LEN+1,
10.0*SHIFT_PITCH_CORREL_SPL_THR, false, 2);
        matbCopy(WarpedDegSpec+1, SmoothedRefSpec+1, NumBands-1);
        SlidingWinPowNorm(POLQAHandle, SmoothedDegSpec+1, WarpedDegSpec+1, NumBands-1,
0.0,
                          2*SHIFT_PITCH_SMOOTHING_LEN+1,
10.0*SHIFT_PITCH_CORREL_SPL_THR, false, 2);
        matbCopy(WarpedDegSpec+1, SmoothedDegSpec+1, NumBands-1);

        matbCopy(SmoothedDegSpec+1, OrigSmoothedDegSpec+1, NumBands-1);

        MaxShiftLen = ((((int)ceil((SHIFT_PITCH_MAX_PITCH_RATIO-1.0)*BinLen)) <
(NumBands/2 - 1)) ? ((int)ceil((SHIFT_PITCH_MAX_PITCH_RATIO-1.0)*BinLen)) :
(NumBands/2 - 1));
        RefAutoCorr = matbNormL2(SmoothedRefSpec+StartBin, BinLen);
        DegAutoCorr = matbNormL2(SmoothedDegSpec+StartBin, BinLen);
        matCrossCorr(POLQAHandle->mh, SmoothedRefSpec+StartBin, BinLen,
SmoothedDegSpec+StartBin, BinLen,
                     WarpedDegSpec, 2*MaxShiftLen+1, -MaxShiftLen);
```

```
        MaxXCorr    = matMax(WarpedDegSpec, 2*MaxShiftLen+1);
        MaxXCorr   /= RefAutoCorr*DegAutoCorr;

        if (MaxXCorr < SHIFT_PITCH_MIN_CORR)
        {
            continue;
        }

        MinPitchRatio = MaxPitchRatio = 1.0f;
        for (int j = -SHIFT_PITCH_FRAME_SEARCH_LEN; j <= SHIFT_PITCH_FRAME_SEARCH_LEN;
j++)
        {
            if (i+j < statics->startFrameIdx ||
                i+j > statics->stopFrameIdx ||
                (RefPitch = CPairParameters::mpPitchVecDeg[i+j]) <= 0.0 ||
                (DegPitch = CPairParameters::mpPitchVec   [i+j]) <= 0.0)
                continue;

            if (RefPitch / DegPitch >= 2.0)
                RefPitch /= 2.0;
            if (DegPitch / RefPitch >= 2.0)
                DegPitch /= 2.0;

            if (RefPitch>0 && DegPitch>0)

            {

                MinPitchRatio = (((DegPitch / RefPitch) < (MinPitchRatio)) ? (DegPitch
/ RefPitch) : (MinPitchRatio));
                MaxPitchRatio = (((DegPitch / RefPitch) > (MaxPitchRatio)) ? (DegPitch
/ RefPitch) : (MaxPitchRatio));

            }
        }
        if (AlmostEqualUlpsFinal((float)MinPitchRatio, 1.0f, 4) &&
AlmostEqualUlpsFinal((float)MaxPitchRatio, 1.0f, 4))
        {
            continue;
        }

        MinPitchRatio = ((int)(((100*(((1/SHIFT_PITCH_MAX_PITCH_RATIO) >
(MinPitchRatio-0.01)) ? (1/SHIFT_PITCH_MAX_PITCH_RATIO) :
(MinPitchRatio-0.01))) > 0) ? (100*(((1/SHIFT_PITCH_MAX_PITCH_RATIO) >
(MinPitchRatio-0.01)) ? (1/SHIFT_PITCH_MAX_PITCH_RATIO) :
(MinPitchRatio-0.01)))+0.5f : (100*(((1/SHIFT_PITCH_MAX_PITCH_RATIO) >
(MinPitchRatio-0.01)) ? (1/SHIFT_PITCH_MAX_PITCH_RATIO) :
(MinPitchRatio-0.01)))-0.5f)) / 100.0;
        MaxPitchRatio = ((int)(((100*(((SHIFT_PITCH_MAX_PITCH_RATIO) <
(MaxPitchRatio+0.01)) ? (SHIFT_PITCH_MAX_PITCH_RATIO) : (MaxPitchRatio+0.01)))
> 0) ? (100*(((SHIFT_PITCH_MAX_PITCH_RATIO) < (MaxPitchRatio+0.01)) ?
(SHIFT_PITCH_MAX_PITCH_RATIO) : (MaxPitchRatio+0.01)))+0.5f :
(100*(((SHIFT_PITCH_MAX_PITCH_RATIO) < (MaxPitchRatio+0.01)) ?
(SHIFT_PITCH_MAX_PITCH_RATIO) : (MaxPitchRatio+0.01)))-0.5f)) / 100.0;

        BestCorr = -1.0;
        BestWarpingFac = 1.0;
        for (XFLOAT pitchRatio = MinPitchRatio; pitchRatio <= MaxPitchRatio; pitchRatio
+= 0.01)
        {
            WarpSpectrum(spectrumDeg->m_pData[i], WarpedDegSpec, pitchRatio, NumBands);

            SlidingWinMean(this->POLQAHandle, WarpedDegSpec+1,
SHIFT_PITCH_SMOOTHING_LEN, SmoothedDegSpec+1, NumBands-1);
            matbThresh1(SmoothedDegSpec+1, NumBands-1, (XFLOAT)0.0, MAT_LT);
            matbLog2(SmoothedDegSpec+1, WarpedDegSpec+1, NumBands-1);
            matbAdd1(-SHIFT_PITCH_CORREL_SPL_THR, WarpedDegSpec+1, NumBands-1);
            matbThresh1(WarpedDegSpec+1, NumBands-1, 0.0, MAT_LT);

            SlidingWinPowNorm(POLQAHandle, WarpedDegSpec+1, SmoothedDegSpec+1,
NumBands-1, 0.0,
                              2*SHIFT_PITCH_SMOOTHING_LEN+1,
10.0*SHIFT_PITCH_CORREL_SPL_THR, false, 2);

            WarpedStartBin = ((((int)ceil(StartBin*pitchRatio)) > (StartBin)) ?
((int)ceil(StartBin*pitchRatio)) : (StartBin));
            WarpedStopBin  = ((((int) (StopBin *pitchRatio)) < ((((StopBin) <
```

```
(SHIFT_PITCH_MAX_PITCH_BIN)) ? (StopBin) : (SHIFT_PITCH_MAX_PITCH_BIN)))) ?
((int) (StopBin *pitchRatio)) : ((((StopBin) < (SHIFT_PITCH_MAX_PITCH_BIN))
? (StopBin) : (SHIFT_PITCH_MAX_PITCH_BIN))));
            WarpedBinLen   = WarpedStopBin - WarpedStartBin + 1;

            CorrAfter = matPearsonCorrelation(SmoothedRefSpec+WarpedStartBin,
SmoothedDegSpec+WarpedStartBin, WarpedBinLen);

            if (CorrAfter > BestCorr)
            {
                BestCorr = CorrAfter;
                BestWarpingFac = pitchRatio;
            }
        }

        WarpSpectrum(spectrumDeg->m_pData[i], WarpedDegSpec, BestWarpingFac, NumBands);

        SlidingWinMean(this->POLQAHandle, WarpedDegSpec+1, SHIFT_PITCH_SMOOTHING_LEN,
SmoothedDegSpec+1, NumBands-1);
        matbThresh1(SmoothedDegSpec+1, NumBands-1, (XFLOAT)0.0, MAT_LT);
        matbLog2(SmoothedDegSpec+1, WarpedDegSpec+1, NumBands-1);
        matbAdd1(-SHIFT_PITCH_CORREL_SPL_THR, WarpedDegSpec+1, NumBands-1);
        matbThresh1(WarpedDegSpec+1, NumBands-1, 0.0, MAT_LT);

        SlidingWinPowNorm(POLQAHandle, WarpedDegSpec+1, SmoothedDegSpec+1, NumBands-1,
0.0,
                          2*SHIFT_PITCH_SMOOTHING_LEN+1,
10.0*SHIFT_PITCH_CORREL_SPL_THR, false, 2);

        WarpedStartBin = ((((int)ceil(StartBin*BestWarpingFac)) > (StartBin)) ?
((int)ceil(StartBin*BestWarpingFac)) : (StartBin));
        WarpedStopBin  = ((((int) (StopBin *BestWarpingFac)) < ((((StopBin) <
(SHIFT_PITCH_MAX_SPEC_BIN)) ? (StopBin) : (SHIFT_PITCH_MAX_SPEC_BIN)))) ?
((int) (StopBin *BestWarpingFac)) : ((((StopBin) < (SHIFT_PITCH_MAX_SPEC_BIN))
? (StopBin) : (SHIFT_PITCH_MAX_SPEC_BIN))));
        WarpedBinLen   = WarpedStopBin - WarpedStartBin + 1;

        CorrBefore = matPearsonCorrelation(SmoothedRefSpec+WarpedStartBin,
OrigSmoothedDegSpec+WarpedStartBin, WarpedBinLen);
        CorrAfter  = matPearsonCorrelation(SmoothedRefSpec+WarpedStartBin,
SmoothedDegSpec    +WarpedStartBin, WarpedBinLen);

        if (CorrAfter - (1.0/CorrBefore - 1.0)*SHIFT_PITCH_CORREL_DAMPING <=
CorrBefore)
        {
            continue;
        }
        else
        {
            bestShiftPerFrame     [i] = (((1) >
((int)((int)(((fabs(1.0-BestWarpingFac)*CPairParameters::mPitchFreqRef) >
0) ? (fabs(1.0-BestWarpingFac)*CPairParameters::mPitchFreqRef)+0.5f :
(fabs(1.0-BestWarpingFac)*CPairParameters::mPitchFreqRef)-0.5f)))) ? (1) :
((int)((int)(((fabs(1.0-BestWarpingFac)*CPairParameters::mPitchFreqRef) >
0) ? (fabs(1.0-BestWarpingFac)*CPairParameters::mPitchFreqRef)+0.5f :
(fabs(1.0-BestWarpingFac)*CPairParameters::mPitchFreqRef)-0.5f))));
            bestWarpingFacPerFrame[i] = BestWarpingFac;
        }
    }

    XFLOAT PrevWarpingFacLog10 = (XFLOAT)0.0, WarpingFacLog10;
    for (int i = statics->startFrameIdx; i <= statics->stopFrameIdx; i++)
    {
        WarpingFacLog10 = log10((XFLOAT)bestWarpingFacPerFrame[i]);

        if (WarpingFacLog10 > PrevWarpingFacLog10)
            WarpingFacLog10 = (((WarpingFacLog10) < (PrevWarpingFacLog10 +
SHIFT_PITCH_MAX_LOGFAC_CHANGE)) ? (WarpingFacLog10) : (PrevWarpingFacLog10
+ SHIFT_PITCH_MAX_LOGFAC_CHANGE));
        else
            WarpingFacLog10 = (((WarpingFacLog10) > (PrevWarpingFacLog10 -
SHIFT_PITCH_MAX_LOGFAC_CHANGE)) ? (WarpingFacLog10) : (PrevWarpingFacLog10
- SHIFT_PITCH_MAX_LOGFAC_CHANGE));

        WarpSpectrum(spectrumDeg->m_pData[i], spectrumDegCorrected->m_pData[i],
                     (XFLOAT)pow((XFLOAT)10.0, WarpingFacLog10), NumBands);
```

```
        PrevWarpingFacLog10 = WarpingFacLog10;
    }
}

void WarpSpectrum(XFLOAT const *OrigSpec, XFLOAT *WarpedSpec, XFLOAT WarpingFac, int
NumBands)
{
    if (WarpingFac <= 0.0)
        return;
    if (AlmostEqualUlpsFinal((float)WarpingFac, 1.0f, 4))
    {
        matbCopy(OrigSpec, WarpedSpec, NumBands);
        return;
    }

    WarpedSpec[0] = OrigSpec[0];
    XFLOAT exactIdx, lowerRatio, upperRatio;

    WarpingFac = 1.0/WarpingFac;

    int i;
    for (i = 1; i < NumBands && (int)ceil(WarpingFac*i) < NumBands; i++)
    {
        exactIdx   = (((WarpingFac*i) > (1.0)) ? (WarpingFac*i) : (1.0));
        upperRatio = exactIdx - (int)(exactIdx);
        lowerRatio = 1.0 - upperRatio;

        WarpedSpec[i] = lowerRatio*OrigSpec[(int)exactIdx] +
upperRatio*OrigSpec[(int)ceil(exactIdx)];
    }

    if (i < NumBands && (int)(WarpingFac*i) < NumBands)
    {
        exactIdx   = WarpingFac*i;
        lowerRatio = (int)ceil(exactIdx) - exactIdx;

        WarpedSpec[i] = lowerRatio*OrigSpec[(int)exactIdx];
    }
    for (; i < NumBands; i++)
        WarpedSpec[i] = 1e-10;
}

void SlidingWinMean(CPOLQAData *PolqaHandle, XFLOAT const *in, int oddWinlen, XFLOAT
*out, int len)
{
    if (oddWinlen > len)
        oddWinlen = (len-1) | 0x1;
    int i = 0, j, halfwinlen = oddWinlen/2;
    if (in == NULL || out == NULL || oddWinlen <= 1)
        return;

    SmartBufferPolqa SB1(PolqaHandle, oddWinlen);
    XFLOAT *temp = SB1.Buffer;

    for (i = 0; i <= halfwinlen; i++)
        temp[i%(halfwinlen+1)] = matMean(in, 2*i+1);

    for (; i < len - halfwinlen; i++)
    {
        out[i-(halfwinlen+1)] = temp[i%(halfwinlen+1)];
        temp[i%(halfwinlen+1)] = matMean(in+i-halfwinlen, oddWinlen);
    }

    for (; i < len; i++)
    {
        out [i-(halfwinlen+1)] = temp[i%(halfwinlen+1)];
        temp[i%(halfwinlen+1)] = matMean(in+len-2*(len-i-1)-1, 2*(len-i-1)+1);
    }

    for (j = 0; j < halfwinlen+1; i++, j++)
        out[i-(halfwinlen+1)] = temp[i%(halfwinlen+1)];

    return;
}
```

```
void SlidingWinPowNorm(CPOLQAData *POLQAHandle,
                       XFLOAT const *fIn, XFLOAT *fOut,
                       int len,
                       XFLOAT thresh,
                       int WINLEN,
                       XFLOAT meanOutLevel,
                       bool doAvoidShortBursts,
                       int hystLenInSamples)
{

    const double ROUNDFACTOR = 1e8;

    if ((WINLEN | 0x1) != WINLEN)
        WINLEN--;

    if (fIn == NULL || fOut == NULL || len < WINLEN+3 ||
        fIn == fOut || WINLEN < 2)
        throw std::string("Invalid input parameters. Note: In-place operation is not
supported.");

    double dWinEnergy  = 0.0;
    double dEnergThr   = thresh * WINLEN;
    int    hystCounter = 0;
    XFLOAT fTemp;
    fTemp = matbNormL2(fIn, WINLEN/2);

    dWinEnergy = floor(ROUNDFACTOR*fTemp*fTemp)/ROUNDFACTOR;

    SmartBufferPolqa SB(POLQAHandle, len);
    XFLOAT *fInSquared = SB.Buffer;

    matbSqr2(fIn, fInSquared, len);

    //Process first WINLEN/2 data points
    for (int i = 0; i < WINLEN/2+1; i++)
    {
        if (dWinEnergy > dEnergThr || (hystCounter > 0 && dWinEnergy > 0.0))
        {
            fOut[i]     = (XFLOAT)(fIn[i] / sqrt(dWinEnergy / WINLEN));
            hystCounter = dWinEnergy > dEnergThr ?
                (((hystLenInSamples) < (hystCounter+1)) ? (hystLenInSamples) :
(hystCounter+1)) :
                (((0) > (hystCounter-1)) ? (0) : (hystCounter-1));
        }
        else
        {
            fOut[i]     = (XFLOAT)0.0;
            hystCounter = 0;
        }

        dWinEnergy += floor(ROUNDFACTOR * fIn[i+WINLEN/2]*fIn[i+WINLEN/2])/ROUNDFACTOR;

    }

    //Main loop
    for (int i = WINLEN/2+1; i < len - WINLEN/2; i++)
    {
        if (dWinEnergy > dEnergThr || (hystCounter > 0 && dWinEnergy > 0.0))
        {
            fOut[i]     = (XFLOAT)(fIn[i] / sqrt(dWinEnergy / WINLEN));
            hystCounter = dWinEnergy > dEnergThr ?
                (((hystLenInSamples) < (hystCounter+1)) ? (hystLenInSamples) :
(hystCounter+1)) :
                (((0) > (hystCounter-1)) ? (0) : (hystCounter-1));
        }
        else
        {
            fOut[i]     = (XFLOAT)0.0;
            hystCounter = 0;
        }

        dWinEnergy = floor(matSum(&fInSquared[i - WINLEN/2], WINLEN) *
ROUNDFACTOR)/ROUNDFACTOR;

    }
```

```
    //Process last WINLEN/2 data points
    for (int i = len - WINLEN/2; i < len; i++)
    {
        if (dWinEnergy > dEnergThr || (hystCounter > 0 && dWinEnergy > 0.0))
        {
            fOut[i]     = (XFLOAT)(fIn[i] / sqrt(dWinEnergy / WINLEN));
            hystCounter = dWinEnergy > dEnergThr ?
                (((hystLenInSamples) < (hystCounter+1)) ? (hystLenInSamples) :
(hystCounter+1)) :
                (((0) > (hystCounter-1)) ? (0) : (hystCounter-1));
        }
        else
        {
            fOut[i]     = (XFLOAT)0.0;
            hystCounter = 0;
        }

        dWinEnergy -= floor(ROUNDFACTOR *
fIn[i-WINLEN/2-1]*fIn[i-WINLEN/2-1])/ROUNDFACTOR;

    }

    if (doAvoidShortBursts)
    {
        int actStart = 0;
        for (int i = 0; i < len; i++)
        {
            while (fOut[i] == (XFLOAT)0.0 && i < len) i++;
            actStart = i;
            while (fOut[i] != (XFLOAT)0.0 && i < len) i++;
            if (i - actStart < 2*WINLEN)
                for (int j = actStart; j < i; j++)
                    fOut[j] = (XFLOAT)0.0;
        }
    }

    //Rescale processed data to desired level
    XFLOAT fScalefac;
    fScalefac = matbNormL2(fOut, len) / sqrt((XFLOAT)len);
    if (fScalefac > (XFLOAT)0.0)
    {
        fScalefac = pow((XFLOAT)10.0, (XFLOAT)meanOutLevel/(XFLOAT)20.0) / fScalefac;
        matbMpy1(fScalefac, fOut, len);
    }
}

//START OF THE MAIN DisturbanceProcess ROUTINE

BOOL CPairParameters::DisturbanceProcess (POLQA_RESULT_DATA* pOverviewHolder)
{

    XFLOAT MeasuredSampleRate = pOverviewHolder->m_MeasuredSamplerate;
    CBarkSpectrum   smearedOriginalPitchPowerDensity,
smearedDistortedPitchPowerDensity;
    CBarkSpectrum   originalPitchPowerDensityMainAvg,
distortedPitchPowerDensityMainAvg;
    CBarkSpectrum   originalPitchLoudnessDensityMainAvg,
distortedPitchLoudnessDensityMainAvg;
    CBarkSpectrum   disturbanceDensityAsymAdd;
    CBarkSpectrum   mask;

    CHzSpectrum     originalHzPowerSpectrum, distortedHzPowerSpectrum;
    CBarkSpectrum   originalPitchPowerDensity, distortedPitchPowerDensity;
    CBarkSpectrum   originalPitchPowerDensity_intact,
distortedPitchPowerDensity_intact;
    CBarkSpectrum   originalLoudnessDensity, distortedLoudnessDensity;
    CBarkSpectrum   disturbanceDensity;

    XFLOAT          ratioAvgCorrection, globalScaleCorrection,
globalScaleCorrectionActiveAdded;
    int             numberCVCsoft, numberCVCsoft2, numberCVCsoft3, numberCVCactive;
    int             numberOfSpeechFrames, numberOfFrames, numberOfSilentFrames,
numberOfNotSilentFrames, numberOfActiveFrames = 0, numberOfSuperSilentFrames;
    int             numberOfaActiveFreqresponse, numberOfaSuperLoud,
numberOfPowerRatioFrames, numberOfPowerRatioTimeClipFrames, numberSilentRatioOk;
    int             numberActiveOk, numberActiveRatioOk, numberActiveRatioOkCorrection;
```

```
    int             frameIndex, bandIndex, bandIdxLow, bandIdxHigh,  hulpCount;
    int             SNRloudnessCountTotal, SNRloudnessCountExcellent,
SNRloudnessCountGood,   SNRloudnessCountFair;
    int             SNRloudnessCountPoor, SNRloudnessCountBad;

    int             SNRloudnessRatioCountTotal, SNRloudnessRatioCountOK;

    int THRESHOLD_BAD_FRAMES;

    XFLOAT          powFac;
    XFLOAT          oldScale = 1, oldOldScale = 1;
    XFLOAT          scale, MaxScale, MinScale, MinMinScale, scaleOriginalFactor,
scaleCorrectionQuality, scaleCorrectionQualityPlus,
scaleCorrectionQualityPlusAdded;
    XFLOAT          scaleCorrectionQualityPlusOld, scaleCorrectionQualityPlusAddedOld;
    XFLOAT          scaleCorrectionIntell, scaleCorrectionMusic,
scaleCorrectionIntellOld, scaleCorrectionMusicOld;
    XFLOAT          minimumOriginalFramePower,  maxDisturbance;
    XFLOAT          aPowerRatioaAvg, aPowerRatioaAvgProduct, aOriginalSilencePowerMean,
aDistortedSilencePowerMean;
    XFLOAT          aOriginalCVCsoftPowerMean, aDistortedCVCsoftPowerMean,
aOriginalCVCactivePowerMean, aDistortedCVCactivePowerMean;
    XFLOAT          aOriginalCVCsoftPowerMean2, aDistortedCVCsoftPowerMean2,
aOriginalCVCsoftPowerMean3, aDistortedCVCsoftPowerMean3;
    XFLOAT          aOriginalLoudnessMean, aOriginalLoudnessPureFrqMean,
aDistortedLoudnessMean, aDistortedLoudnessPureFrqMean;
    XFLOAT          LpBandRangeLocal, aLoudnessScalingOriginal,
aLoudnessScalingDistorted;
    XFLOAT          LpBandRangePartial, LpLoudnessMeanPartial, LpBandRangeComplete,
LpLoudnessMeanComplete, aLoudnessPureFrqScaling, LpLoudness;
    XFLOAT          fixedGlobalInternalLevel, fixedGlobalInternalLevelAdded;
    CNewStdString      s;
    int             count, count0, count1,  i;
    XFLOAT          hulp, hulp0, hulp1, hulp2, hulp3, hulp4,  hulpRatio,  hulpMem,
loudnessPureFrqVar;
    XFLOAT          hulpLowOld, hulpLow, hulpHighOld, hulpHigh;
    XFLOAT          hulpLowOldNarrowband, hulpLowNarrowband, hulpHighOldNarrowband,
hulpHighNarrowband;

    XFLOAT          maxDisturbanceOverFile;
    int             maxDisturbanceFrame;

    XFLOAT          loudnessScaleLow;
    XFLOAT          oldOldLoudnessScaleLow, oldLoudnessScaleLow;
    XFLOAT          originalLoudnessHulp, distortedLoudnessHulp;

    XFLOAT          frameCorrelationTimeOriginal, frameCorrelationTimeDistorted,
frameCorrelationTimeDisturbance;
    XFLOAT          frameCorrelationTimeOriginalOld, frameCorrelationTimeDistortedOld;
    XFLOAT          frameCorrelationTimeCompensationOriginal,
frameCorrelationTimeCompensationDistorted,
frameCorrelationTimeCompensationDifference;
    XFLOAT          frameFlatnessTimeOriginal, frameFlatnessTimeDistorted,
frameFlatnessDisturbance, frameFlatnessDisturbanceAvg;
    XFLOAT          frameFlatnessDistortedAvgCompensationSilent,
frameFlatnessDistortedAvgCompensationAddedSilent;
    XFLOAT          frameFlatnessDisturbanceAvgCompensationSilent,
frameFlatnessDisturbanceAvgCompensationAddedSilent;
    XFLOAT          frameFlatnessDisturbanceAvgCompensationActive,
frameFlatnessDisturbanceAvgCompensationAddedActive,
frameFlatnessDisturbanceAvgCompensationActiveFrq;
    XFLOAT          frameFlatnessDisturbanceAvgCompensation,
frameFlatnessDisturbanceAvgCompensationAdded, frameFlatnessDisturbanceAdded;
    XFLOAT          overallDisturbance, overallMovingAvgDisturbance,
overallMovingAvgDisturbanceOld, overallMovingAvgDisturbanceOldOld;
    XFLOAT          overallAvgDisturbance, overallAvgAddedDisturbance;
    XFLOAT          originalLoudnessTimbrePerFrame,
originalLoudnessTimbrePerFrameDifferenceOld,
originalLoudnessTimbrePerFrameDifference,
originalLoudnessTimbrePerFrameDifferenceCompensation,
originalLoudnessTimbrePerFrameDifferenceCompensationAdd;
    XFLOAT          distortedLoudnessTimbrePerFrame,
distortedLoudnessTimbreHighPerFrame;
    XFLOAT          distortedLoudnessTimbrePerFrameNarrowband,
distortedLoudnessTimbrePerFrameNarrowbandAvg,
distortedLoudnessTimbrePerFrameLoudAvg, differenceInLoudnessTimbrePerFrame;
```

```
    XFLOAT          distortedLoudnessTimbreHighPerFrameAvg,
distortedLoudnessTimbreHighPerFrameAvgSilent;
    XFLOAT          distortedLoudnessTimbrePerFrameDifferenceOld,
distortedLoudnessTimbrePerFrameDifference,
distortedLoudnessTimbrePerFrameDifferenceCompensation;
    XFLOAT          noiseContrastParameter, avgPitchLoudFramesCompensation;
    XFLOAT          PitchRatio = 1.0;
    XFLOAT          globalScaleCorrectionIntellLevelCorrectionForMaximumD,
globalScaleCorrectionIntellLevelCorrectionForMaximumA;
    XFLOAT          bandLowBarkPolqaPlus;
    XFLOAT          delayReliabilityPerFrameWeight, delayReliabilityPerFrameWeightOld;
    XFLOAT          number_of_sections_inserted;
    XFLOAT          number_of_sections_critical;
    XFLOAT          number_of_sections_invalid;

    originalHzPowerSpectrum. Initialize ("originalHzPowerSpectrum", POLQAHandle);
    distortedHzPowerSpectrum. Initialize ("distortedHzPowerSpectrum", POLQAHandle);
    originalPitchPowerDensity. Initialize ("originalPitchPowerDensity", POLQAHandle);
    distortedPitchPowerDensity. Initialize ("distortedPitchPowerDensity", POLQAHandle);
    originalPitchPowerDensity_intact. Initialize ("originalPitchPowerDensity_intact",
POLQAHandle);
    distortedPitchPowerDensity_intact. Initialize ("distortedPitchPowerDensity_intact",
POLQAHandle);
    smearedOriginalPitchPowerDensity. Initialize ("smearedOriginalPitchPowerDensity",
POLQAHandle);
    smearedDistortedPitchPowerDensity. Initialize ("smearedDistortedPitchPowerDensity",
POLQAHandle);

    originalLoudnessDensity. Initialize ("originalLoudnessDensity", POLQAHandle);
    distortedLoudnessDensity. Initialize ("distortedLoudnessDensity", POLQAHandle);
    originalPitchPowerDensityMainAvg. Initialize ("originalPitchPowerDensityMainAvg",
POLQAHandle);
    distortedPitchPowerDensityMainAvg. Initialize ("distortedPitchPowerDensityMainAvg",
POLQAHandle);
    originalPitchLoudnessDensityMainAvg. Initialize
("originalPitchLoudnessDensityMainAvg", POLQAHandle);
    distortedPitchLoudnessDensityMainAvg. Initialize
("distortedPitchLoudnessDensityMainAvg", POLQAHandle);
    disturbanceDensity. Initialize ("disturbanceDensity", POLQAHandle);
    disturbanceDensityAsymAdd. Initialize ("disturbanceDensityAsymAdd", POLQAHandle);
    mask. Initialize ("mask", POLQAHandle);

    XFLOAT delayThrMs = 3.0;
    XFLOAT delayThrMsMem = 3.0;
    XFLOAT hulpDelay, hulpDelayMem, hulpDelayMemOld, delayVariationCompensation,
delayVariationCompensationAdded;
    hulpDelay = 9999.0;
    hulpDelayMem = 9999.0;
    hulpDelayMemOld = 9999.0;
    pOverviewHolder->m_ConstantDelayIndicator = 0;
    for(int i = (statics->startFrameIdx+2); i <= statics->stopFrameIdx; i++) {
        hulpDelayMemOld = hulpDelayMem;
        hulpDelayMem = hulpDelay;
        hulpDelay = 1.0*abs(pOverviewHolder->m_DelayPerFrame[i] -
pOverviewHolder->m_DelayPerFrame[i-2]);
        if(( (XFLOAT) (hulpDelay)*1000.0/ (XFLOAT)
pOverviewHolder->m_SampleFrequencyHz) < 10.0)
pOverviewHolder->m_ConstantDelayIndicator += ( (XFLOAT) (hulpDelay)*1000.0/
(XFLOAT) pOverviewHolder->m_SampleFrequencyHz);
    }
    numberOfSpeechFrames = statics->stopFrameIdx - statics->startFrameIdx;
    pOverviewHolder->m_ConstantDelayIndicator /= ((XFLOAT)numberOfSpeechFrames + 0.1);
    constantDelayIndicator = pOverviewHolder->m_ConstantDelayIndicator;
    delayVariationCompensation = 1.0 + constantDelayIndicator/20.0;
    if (aListeningCondition==WIDE_H) {
        delayVariationCompensationAdded = 1.0 + constantDelayIndicator/20.0;
    } else {
        delayVariationCompensationAdded = 1.0 + constantDelayIndicator/10.0;
    }

    delayVariationCompensation = 1.0;
    delayVariationCompensationAdded = 1.0;

    ShowProgress (10, "PSQM Processing");

    int NumDelayChanges = 0;
```

```
    int utt;
    long LastDelay = aDelayUtterance.m_pData[0];
    int numberOfUtterances = aDelayUtterance.GetSize();
    bool* UseThisFrame = new bool[statics->stopFrameIdx+1];

    long* FrameFlags = new long[statics->stopFrameIdx+1];

    int testSum = 0;
    for (frameIndex = 0; frameIndex <= statics->stopFrameIdx; frameIndex++)
    {
        int utt = GetUtteranceForFrame(aStartSampleUtterance, aStopSampleUtterance,
aDelayUtterance, frameIndex, aOriginalTimeSeries.GetFrameLength());
        if(utt>=0)
            testSum++;
        UseThisFrame[frameIndex] = (utt>=0);
        FrameFlags[frameIndex] = 0;
        if (utt >= 0 && aDelayUtterance.m_pData[utt] != LastDelay)
        {
            NumDelayChanges++;
            FrameFlags[frameIndex] |= 0x0000001;
            if (abs(aDelayUtterance.m_pData[utt]-LastDelay)<0.001*statics->sampleRate)
                FrameFlags[frameIndex] |= 0x0000002;

        }
        if (utt>=0) LastDelay = aDelayUtterance.m_pData[utt];
    }
    pOverviewHolder->m_NumDelayChangesPerFrame = (XFLOAT)NumDelayChanges /
(XFLOAT)statics->stopFrameIdx;

    PitchRatio = pOverviewHolder->m_PitchRef / pOverviewHolder->m_PitchDeg;
    PitchRatio=ComputePitchRatio();
    pOverviewHolder->m_PitchRatio=PitchRatio;
    pOverviewHolder->m_VoicedFramesRef = mNumVoicedFramesRef;
    pOverviewHolder->m_VoicedFramesDeg = mNumVoicedFramesDeg;
    pOverviewHolder->m_VoicedFrames = mNumVoicedFrames;

    ShowProgress (5, "Power spectrum computation - original");

    originalHzPowerSpectrum. STFTPowerAndPhaseSpectrumOf (POLQAHandle,
aOriginalTimeSeries, aStartSampleUtterance, aStopSampleUtterance, aDelayUtterance,
false, false);

    globalScaleDistortedToFixedlevelHulp = globalScaleDistortedToFixedlevel;
    if (globalScaleDistortedToFixedlevelHulp>9.0) globalScaleDistortedToFixedlevelHulp
= 9.0;
    if (globalScaleDistortedToFixedlevelHulp<1.0) globalScaleDistortedToFixedlevelHulp
= 1.0;
    globalScaleDistortedToFixedlevelHulp = pow(globalScaleDistortedToFixedlevelHulp,
0.01);

    if (aListeningCondition==WIDE_H) {
        LpBandRangeLocal = 1.2;
        LpBandRangePartial = 2.1;
        LpLoudnessMeanPartial = 0.65;
        LpBandRangeComplete = 2.0;
        LpLoudnessMeanComplete = 1.45;
        LpLoudness = 2.2;
        fixedGlobalInternalLevel = 20.0;
        fixedGlobalInternalLevelAdded = 16.0;
    } else {
        LpBandRangeLocal = 1.15;
        LpBandRangePartial = 2.3;
        LpLoudnessMeanPartial = 0.7;
        LpBandRangeComplete = 2.05;
        LpLoudnessMeanComplete = 1.4;
        LpLoudness = 2.3;
        fixedGlobalInternalLevel = 20.0;
        fixedGlobalInternalLevelAdded = 17.0;
    }

    ShowProgress (5, "Power spectrum computation - distorted");

    distortedHzPowerSpectrum. STFTPowerAndPhaseSpectrumOf (POLQAHandle,
aDistortedTimeSeries, aStartSampleUtterance, aStopSampleUtterance, aDelayUtterance,
true, true);
```

```
    CHzSpectrum *distortedHzPowerSpectrumCorrected = new CHzSpectrum;
    distortedHzPowerSpectrumCorrected->Initialize("distortedHzPowerSpectrumCorrected",
POLQAHandle);
    bestSpectrumShift = 0;
    bestSpectrumShift = (int*)matMalloc((statics->stopFrameIdx + 1) * sizeof(int));
    XFLOAT *bestWarpingFacPerFrame = (XFLOAT*)matMalloc((statics->stopFrameIdx + 1) *
sizeof(XFLOAT));

    CheckTimeMatInit(POLQAHandle->mh, 3);

    ShowProgress (10, "Spectrum correction");

    for (int i = statics->startFrameIdx; i <= statics->stopFrameIdx; i++)

    {
        bestSpectrumShift     [i] = 0;
        bestWarpingFacPerFrame[i] = 1.0;
        WarpSpectrum(distortedHzPowerSpectrum.m_pData[i],
distortedHzPowerSpectrumCorrected->m_pData[i], 1.0/PitchRatio,
statics->aNumberOfHzBands);
        mpPitchVecDeg[i] *= PitchRatio;

            matbCopy(distortedHzPowerSpectrumCorrected->m_pData[i],
distortedHzPowerSpectrum.m_pData[i], statics->aNumberOfHzBands);

    }

    PitchRatio = 1.0;

        ShiftPitch(&originalHzPowerSpectrum, &distortedHzPowerSpectrum,
distortedHzPowerSpectrumCorrected, pActiveFrameFlags, bestSpectrumShift,
bestWarpingFacPerFrame);

    for(int fr = statics->startFrameIdx; fr <= statics->stopFrameIdx; fr++)
        matbCopy(distortedHzPowerSpectrumCorrected->m_pData[fr],
distortedHzPowerSpectrum.m_pData[fr], statics->aNumberOfHzBands);
    delete distortedHzPowerSpectrumCorrected;
    distortedHzPowerSpectrumCorrected = NULL;

    CheckTimeMatEval(POLQAHandle->mh, 3, &ClockCycles, &TimeDiff);
    AddProcessingTime(pOverviewHolder, "Spectrum correction", TimeDiff, ClockCycles);

    ShowProgress (10, "Disturbance computation");

    XFLOAT SampleRateRatio = pOverviewHolder->m_MeasuredSamplerate/statics->sampleRate;
    XFLOAT SampleRateRatioCompensation, SampleRateRatioCompensation2,
SampleRateRatioCompensation3, SampleRateRatioCompensation4,
SampleRateRatioCompensation5;

    if (SampleRateRatio<0.0) SampleRateRatio = 1.0;

    if (SampleRateRatio<0.9) {
            s. Format ("SampleRateRatio less then 0.9 \n ");
            gLogFile. WriteString (s);
    }
    if (SampleRateRatio>1.1) {
            s. Format ("SampleRateRatio larger then 1.1 \n ");
            gLogFile. WriteString (s);
    }
    if (aListeningCondition==WIDE_H) {
        SampleRateRatioCompensation3 = 1.0;
        if (SampleRateRatio<0.99) SampleRateRatio = 0.99;
        if (SampleRateRatio>1.05) SampleRateRatio = 1.05;
        if (SampleRateRatio<1.0) {
            SampleRateRatioCompensation2 = 1.0/pow(SampleRateRatio, 50.0);
            SampleRateRatioCompensation4 = 1.0/pow(SampleRateRatio, 12.0);
            SampleRateRatioCompensation5 = 1.0/pow(SampleRateRatio, 60.0);
        } else {
            SampleRateRatioCompensation2 = pow(SampleRateRatio, 50.0);
            SampleRateRatioCompensation4 = pow(SampleRateRatio, 12.0);
            SampleRateRatioCompensation5 = pow(SampleRateRatio, 25.0);
        }
        SampleRateRatioCompensation = pow(SampleRateRatio, 12.0);
    } else {
        if (SampleRateRatio<0.98) SampleRateRatio = 0.98;
        if (SampleRateRatio>1.04) SampleRateRatio = 1.04;
```

```
        if (SampleRateRatio<1.0) {
            SampleRateRatioCompensation2 = 1.0/pow(SampleRateRatio, 2500.0);
            SampleRateRatioCompensation3 = 1.0/pow(SampleRateRatio, 2.0);
            SampleRateRatioCompensation4 = 1.0/pow(SampleRateRatio, 12.0);
            SampleRateRatioCompensation5 = 1.0/pow(SampleRateRatio, 50.0);
        } else {
            SampleRateRatioCompensation2 = pow(SampleRateRatio, 2500.0);
            SampleRateRatioCompensation3 = pow(SampleRateRatio, 2.0);
            SampleRateRatioCompensation4 = pow(SampleRateRatio, 12.0);
            SampleRateRatioCompensation5 = pow(SampleRateRatio, 25.0);
        }
        SampleRateRatioCompensation = pow(SampleRateRatio, 12.0);
    }

    SampleRateRatioCompensation  = 1.0;
    SampleRateRatioCompensation2 = 1.0;
    SampleRateRatioCompensation3 = 1.0;
    SampleRateRatioCompensation4 = 1.0;
    SampleRateRatioCompensation5 = 1.0;

    SampleRateRatioCompensation = 1.0;

    originalPitchPowerDensity. FrequencyWarpingOf (POLQAHandle,
originalHzPowerSpectrum, 1.0);
    distortedPitchPowerDensity. FrequencyWarpingOf (POLQAHandle,
distortedHzPowerSpectrum, PitchRatio);

    XFLOAT maxFreqBarkSource;
    if (aSampleFrequencyHzSource<(2650.0*2.0) ) {
        maxFreqBarkSource =
-1.8585e-6*aSampleFrequencyHzSource*aSampleFrequencyHzSource/4.0 +
10.263*aSampleFrequencyHzSource/2.0 + 0.1203;
    } else {
        maxFreqBarkSource = 4.9289*log(aSampleFrequencyHzSource/2.0)/log(2.718281828) -
23.463;
    }
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
        originalPitchPowerDensity. MultiplyWithOverBandRange (frameIndex, 0.0,
maxFreqBarkSource, 99.0);
        distortedPitchPowerDensity. MultiplyWithOverBandRange (frameIndex, 0.0,
maxFreqBarkSource, 99.0);
    }

    aAvgOriginalPower = 0.0;
    aAvgDistortedPower = 0.0;
    numberOfFrames = 0;

    numberCVCsoft = 0;
    numberCVCactive = 0;
    aOriginalCVCsoftPowerMean = 0.0;
    aDistortedCVCsoftPowerMean = 0.0;
    aOriginalCVCactivePowerMean = 0.0;
    aDistortedCVCactivePowerMean = 0.0;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        aOriginalTotalPower.m_pData[frameIndex] = originalPitchPowerDensity. Total
(frameIndex, 300.0, 3500.0);
        aDistortedTotalPower.m_pData[frameIndex] = distortedPitchPowerDensity. Total
(frameIndex, 300.0, 3500.0);
        numberOfFrames++;
        aAvgOriginalPower += aOriginalTotalPower.m_pData[frameIndex];
        aAvgDistortedPower += aDistortedTotalPower.m_pData[frameIndex];
        hulp1 = originalPitchPowerDensity. Total (frameIndex, 300.0, 3500.0);
        hulp2 = distortedPitchPowerDensity. Total (frameIndex, 300.0, 3500.0);
        if ( (hulp1<8.0E7) && (hulp1>2.0E7) ) {
            numberCVCsoft++;
            aOriginalCVCsoftPowerMean += hulp1;
            aDistortedCVCsoftPowerMean += hulp2;
        }
        if ( (hulp1<2.0E8) && (hulp1>2.0E5) ) {
            numberCVCactive++;
            aOriginalCVCactivePowerMean += hulp1;
            aDistortedCVCactivePowerMean += hulp2;
        }
    }
```

```
    aOriginalCVCsoftPowerMean /= (numberCVCsoft+0.01);
    aDistortedCVCsoftPowerMean /= (numberCVCsoft+0.01);
    aOriginalCVCactivePowerMean /= (numberCVCactive+0.01);
    aDistortedCVCactivePowerMean /= (numberCVCactive+0.01);
    if (aListeningCondition==WIDE_H) {
        CVCratioSNRlevelRangecompensation0001 =
((aDistortedCVCsoftPowerMean+1.0)/(aDistortedCVCactivePowerMean+1.0)+2.0) /
((aOriginalCVCsoftPowerMean+1.0)/(aOriginalCVCactivePowerMean+1.0)+2.0);
        if (CVCratioSNRlevelRangecompensation0001<1.0) {
            CVCratioSNRlevelRangecompensation0001 += 0.15;
            if (CVCratioSNRlevelRangecompensation0001>1.0)
CVCratioSNRlevelRangecompensation0001=1.0;
            CVCratioSNRlevelRangecompensation0001 =
pow(CVCratioSNRlevelRangecompensation0001,0.6);
        } else {
            CVCratioSNRlevelRangecompensation0001 = 1.0;
        }
    } else {
        CVCratioSNRlevelRangecompensation0001 =
((aDistortedCVCsoftPowerMean+1.0)/(aDistortedCVCactivePowerMean+1.0)+3.0) /
((aOriginalCVCsoftPowerMean+1.0)/(aOriginalCVCactivePowerMean+1.0)+3.0);
        if (CVCratioSNRlevelRangecompensation0001<1.0) {
            CVCratioSNRlevelRangecompensation0001 += 0.2;
            if (CVCratioSNRlevelRangecompensation0001>1.0)
CVCratioSNRlevelRangecompensation0001=1.0;
            CVCratioSNRlevelRangecompensation0001 =
pow(CVCratioSNRlevelRangecompensation0001,0.4);
        } else {
            CVCratioSNRlevelRangecompensation0001 = 1.0;
        }
    }

    aAvgOriginalPower /= (numberOfFrames+0.01);
    aAvgDistortedPower /= (numberOfFrames+0.01);

    aAvgActiveOriginalPower = 0.0;
    numberOfFrames = 0;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        if (aOriginalTotalPower.m_pData[frameIndex] > aAvgOriginalPower/100.0) {
            aAvgActiveOriginalPower += aOriginalTotalPower.m_pData[frameIndex];
            numberOfFrames++;
        }
    }
    aAvgActiveOriginalPower /= (numberOfFrames+0.01);

    aAvgActiveDistortedPower = 0.0;
    numberOfFrames = 0;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        if (aDistortedTotalPower.m_pData[frameIndex] > aAvgDistortedPower/100.0) {
            aAvgActiveDistortedPower += aDistortedTotalPower.m_pData[frameIndex];
            numberOfFrames++;
        }
    }
    aAvgActiveDistortedPower /= (numberOfFrames+0.01);
    globalScaleCorrection =  (aAvgActiveDistortedPower+0.01) /
(aAvgActiveOriginalPower+0.01);
    globalScaleCorrectionIntellLevelCorrection = (aAvgActiveDistortedPower+1.0) /
(aAvgActiveOriginalPower+1.0);
    if (aListeningCondition==WIDE_H) {
        if (globalScaleCorrectionIntellLevelCorrection<0.1)
globalScaleCorrectionIntellLevelCorrection = 0.1;
        if (globalScaleCorrectionIntellLevelCorrection<1.0) {
            globalScaleCorrectionIntellLevelCorrectionForMaximumD =
pow(globalScaleCorrectionIntellLevelCorrection,0.1);
            globalScaleCorrectionIntellLevelCorrectionForMaximumA =
pow(globalScaleCorrectionIntellLevelCorrection,0.4);
        } else {
            globalScaleCorrectionIntellLevelCorrectionForMaximumD =
pow(globalScaleCorrectionIntellLevelCorrection,0.1);
            globalScaleCorrectionIntellLevelCorrectionForMaximumA =
pow(globalScaleCorrectionIntellLevelCorrection,0.3);
        }
    } else {
        if (globalScaleCorrectionIntellLevelCorrection<0.1)
```

```
globalScaleCorrectionIntellLevelCorrection = 0.1;
        if (globalScaleCorrectionIntellLevelCorrection<1.0) {
            globalScaleCorrectionIntellLevelCorrectionForMaximumD =
pow(globalScaleCorrectionIntellLevelCorrection,0.1);
            globalScaleCorrectionIntellLevelCorrectionForMaximumA =
pow(globalScaleCorrectionIntellLevelCorrection,0.3);
        } else {
            globalScaleCorrectionIntellLevelCorrectionForMaximumD =
pow(globalScaleCorrectionIntellLevelCorrection,0.1);
            globalScaleCorrectionIntellLevelCorrectionForMaximumA =
pow(globalScaleCorrectionIntellLevelCorrection,0.3);
        }
    }

//Determine silent interactive etc.... intervals
    aPowerRatioaAvg = 0.0;
    aPowerRatioaAvgProduct = 1.0;
    aOriginalSilencePowerMean = 0.0;
    aDistortedSilencePowerMean = 0.0;

    numberOfaActiveFreqresponse = 0;
    numberOfaSuperLoud = 0;
    numberOfPowerRatioFrames = 0;
    numberOfPowerRatioTimeClipFrames = 0;
    numberSilentRatioOk = 0;
    numberActiveOk = 0;
    numberActiveRatioOk = 0;
    numberActiveRatioOkCorrection = 0;
    numberOfSilentFrames = 0;
    numberOfNotSilentFrames = 0;
    numberOfSuperSilentFrames = 0;
    ratioAvgCorrection = 0.0;
    envelopeContinuityCompensation000 = 0.0;
    count0 = 0;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {

        if (aListeningCondition==STANDARD_IRS) {
            if (aOriginalTotalPower.m_pData[frameIndex] < 3.0E6 ) {
                numberOfSilentFrames++;
                aSilent.m_pData[frameIndex] = TRUE;
                aOriginalSilencePowerMean += aOriginalTotalPower.m_pData[frameIndex];
                aDistortedSilencePowerMean += aDistortedTotalPower.m_pData[frameIndex];
            } else {
                numberOfNotSilentFrames++;
                aSilent.m_pData[frameIndex] = FALSE;
            }
        }
        if (aListeningCondition==WIDE_H) {
            if (aOriginalTotalPower.m_pData[frameIndex] < 6.0E6 ) {
                numberOfSilentFrames++;
                aSilent.m_pData[frameIndex] = TRUE;
                aOriginalSilencePowerMean += aOriginalTotalPower.m_pData[frameIndex];
                aDistortedSilencePowerMean += aDistortedTotalPower.m_pData[frameIndex];
            } else {
                numberOfNotSilentFrames++;
                aSilent.m_pData[frameIndex] = FALSE;
            }
        }
        if (aListeningCondition==NARROW_H) {
            if (aOriginalTotalPower.m_pData[frameIndex] < 3.0E6 ) {
                numberOfSilentFrames++;
                aSilent.m_pData[frameIndex] = TRUE;
                aOriginalSilencePowerMean += aOriginalTotalPower.m_pData[frameIndex];
                aDistortedSilencePowerMean += aDistortedTotalPower.m_pData[frameIndex];
            } else {
                numberOfNotSilentFrames++;
                aSilent.m_pData[frameIndex] = FALSE;
            }
        }

        aPowerRatio.m_pData[frameIndex] = 1.0;
        if (aOriginalTotalPower.m_pData[frameIndex] > 1.0E7 ) {
            aPowerRatio.m_pData[frameIndex] =
(aOriginalTotalPower.m_pData[frameIndex]+100.0)/(aDistortedTotalPower.m_pDa
ta[frameIndex]+100.0);
```

```
            if (aPowerRatio.m_pData[frameIndex] < 1.0) aPowerRatio.m_pData[frameIndex]
= 1.0;
            aPowerRatioaAvgProduct /= pow(aPowerRatio.m_pData[frameIndex],0.003);
            if (aPowerRatio.m_pData[frameIndex] > 1.0) {
                aPowerRatioaAvg += 1/aPowerRatio.m_pData[frameIndex];
                numberOfPowerRatioFrames++;
            }
            if ( (aOriginalTotalPower.m_pData[frameIndex]>5.0e7) &&
((aOriginalTotalPower.m_pData[frameIndex]+3000.0)/(aDistortedTotalPower.m_p
Data[frameIndex]+3000.0)> 50.0)) numberOfPowerRatioTimeClipFrames++;
        }

        if (aOriginalTotalPower.m_pData[frameIndex] < 2.0E5 ) {
            numberOfSuperSilentFrames++;
            aSuperSilent.m_pData[frameIndex] = TRUE;
        } else {
            aSuperSilent.m_pData[frameIndex] = FALSE;
        }

        if (aOriginalTotalPower.m_pData[frameIndex] > 8.0E6 ) {
            aActiveFreqresponse.m_pData[frameIndex] = TRUE;
            numberOfaActiveFreqresponse++;
        }

        if (aOriginalTotalPower.m_pData[frameIndex] > 7.0E8 ) {
            aSuperLoud.m_pData[frameIndex] = TRUE;
            numberOfaSuperLoud++;
        }

        if (aOriginalTotalPower.m_pData[frameIndex] > 2.0E7 ) {
            aActiveFreqresponseIntell.m_pData[frameIndex] = TRUE;
        }

        aSilentRatioOk.m_pData[frameIndex] = FALSE;
        aActiveRatioOk.m_pData[frameIndex] = FALSE;
        if (aOriginalTotalPower.m_pData[frameIndex] < 3.0E6 ) {
            hulp = (aDistortedTotalPower.m_pData[frameIndex] + 1.0) /
(globalScaleCorrection*aOriginalTotalPower.m_pData[frameIndex]+1.0);
            if (hulp<1.0e4) {
                numberSilentRatioOk++;
                aSilentRatioOk.m_pData[frameIndex] = TRUE;
            }
        } else {
            numberActiveOk++;
            hulp = (aDistortedTotalPower.m_pData[frameIndex] + 10.0) /
(globalScaleCorrection*aOriginalTotalPower.m_pData[frameIndex]+10.0);
            if ( (0.1<hulp) && (hulp<10.0) ) {
                numberActiveRatioOk++;
                aActiveRatioOk.m_pData[frameIndex] = TRUE;
            }
            hulp = (aDistortedTotalPower.m_pData[frameIndex]+10.0) /
(globalScaleCorrection*aOriginalTotalPower.m_pData[frameIndex]+10.0);
            if ( (0.3<hulp) && (hulp<3.3) ) {
                numberActiveRatioOkCorrection++;
            }
        }

        if ( frameIndex>(statics->startFrameIdx + 31) ){
            count1 = 0;
            for (i=0; i<=28; i++) if (aOriginalTotalPower.m_pData[frameIndex-i]>2.0e7)
count1++;
            if (count1>25.0) {
                count0++;
                hulp1 = 0.0;
                hulp2 = 0.0;
                hulp3 = 0.0;
                hulp4 = 0.0;
                for (i=0; i<=12; i++) {
                    hulp1 += aOriginalTotalPower.m_pData[frameIndex-i-12];
                    hulp3 += aDistortedTotalPower.m_pData[frameIndex-i-12];
                    hulp2 += aOriginalTotalPower.m_pData[frameIndex-i];
                    hulp4 += aDistortedTotalPower.m_pData[frameIndex-i];
                }
                envelopeContinuityCompensation000 += fabs( (hulp1+5.0e9)/(hulp2+5.0e9)
- (hulp3+5.0e9)/(hulp4+5.0e9) );
            }
```

```
        }
    }

    aOriginalSilencePowerMean /= (numberOfSilentFrames+0.01);
    aDistortedSilencePowerMean /= (numberOfSilentFrames+0.01);
    hulp = (numberOfNotSilentFrames - 20.0);
    if (hulp<0.0) hulp = 0.0;
    hulp /= 50000.0;
    if (numberOfNotSilentFrames>100) {
        aPowerRatioaAvgTimeClipCompensation000 = pow((
(100+1.0)/(numberOfPowerRatioTimeClipFrames+1.0)),hulp);
    } else {
        aPowerRatioaAvgTimeClipCompensation000 = pow((
(numberOfNotSilentFrames+1.0)/(numberOfPowerRatioTimeClipFrames+1.0)),hulp);
    }
    if (numberOfPowerRatioTimeClipFrames<5) aPowerRatioaAvgTimeClipCompensation000 =
1.0;
    aPowerRatioaAvg /= (numberOfPowerRatioFrames+0.01);
    envelopeContinuityCompensation000 /= (count0+1.0);
    envelopeContinuityCompensation000 /= 5.0;

XFLOAT  aDistortedSilencePowerMeanCompensation =
pow((aDistortedSilencePowerMean+1.0),0.01);
        if (aListeningCondition==STANDARD_IRS) {
            aDistortedSilencePowerMeanCompensation *= 0.8;
        }
        if (aListeningCondition==WIDE_H) {

            aDistortedSilencePowerMeanCompensation *=
(1.0/SampleRateRatioCompensation4);

        }
        if (aListeningCondition==NARROW_H) {
            aDistortedSilencePowerMeanCompensation *=
(0.9/SampleRateRatioCompensation4);
        }

//END Determine silent interactive etc.... intervals

//GLOBAL SCALE original based on all active frames
    hulp1 = 0.0;
    hulp2 = 0.0;
    hulpCount = 0 ;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)     {
        if (!aSilent.m_pData[frameIndex]) {

            hulp1 += originalPitchPowerDensity. Total (frameIndex, 350.0, 3500.0);
            hulp2 += distortedPitchPowerDensity. Total (frameIndex, 350.0, 3500.0);

            hulpCount++;

        }
    }
    hulp1 /= (hulpCount+1.0);
    hulp2 /= (hulpCount+1.0);

    scaleOriginalFactor = (hulp2+1.0e4)/(hulp1+1.0e4);
    if (scaleOriginalFactor<0.03) scaleOriginalFactor = scaleOriginalFactor +
pow((0.03-scaleOriginalFactor),1.5);

    frameFlatnessDistortedAvgCompensationSilent = 0.01;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)     {
        originalPitchPowerDensity. MultiplyWith ( frameIndex, scaleOriginalFactor );
        if (aSuperSilent.m_pData[frameIndex])
frameFlatnessDistortedAvgCompensationSilent += distortedPitchPowerDensity.
SpectralFlatnessPower (frameIndex);
    }
    frameFlatnessDistortedAvgCompensationSilent /= (numberOfSuperSilentFrames+0.01);
    frameFlatnessDistortedAvgCompensationSilent000 =
frameFlatnessDistortedAvgCompensationSilent;

    if (frameFlatnessDistortedAvgCompensationSilent000<0.0003)
frameFlatnessDistortedAvgCompensationSilent000 = 0.0003;
    if (frameFlatnessDistortedAvgCompensationSilent000>0.007)
```

```
frameFlatnessDistortedAvgCompensationSilent000 = 0.007;
    frameFlatnessDistortedAvgCompensationSilent000 =
1/frameFlatnessDistortedAvgCompensationSilent000 - 1.0/0.007;
    frameFlatnessDistortedAvgCompensationSilent000 =
frameFlatnessDistortedAvgCompensationSilent000/1700.0;
    frameFlatnessDistortedAvgCompensationAddedSilent =
pow((frameFlatnessDistortedAvgCompensationSilent+0.3),0.01);
    frameFlatnessDistortedAvgCompensationSilent =
pow((frameFlatnessDistortedAvgCompensationSilent+0.3),0.01);

    if (aListeningCondition==WIDE_H) {
        globalScaleCorrectionActive = (hulp2+1.0e4)/(hulp1+1.0e4);
        globalScaleCorrectionActiveAdded = globalScaleCorrectionActive;
        if (globalScaleCorrectionActive>1.0) {
            globalScaleCorrectionActive = pow(globalScaleCorrectionActive,0.05);
            globalScaleCorrectionActiveAdded =
pow(globalScaleCorrectionActiveAdded,0.07);
        } else {

            globalScaleCorrectionActive = pow(globalScaleCorrectionActive,0.17);
            globalScaleCorrectionActiveAdded =
pow(globalScaleCorrectionActiveAdded,0.07);

        }
    } else {
        globalScaleCorrectionActive = (hulp2+1.0e4)/(hulp1+1.0e4);
        globalScaleCorrectionActiveAdded = globalScaleCorrectionActive;
        if (globalScaleCorrectionActive>1.0) {
            globalScaleCorrectionActive = pow(globalScaleCorrectionActive,0.04);
            globalScaleCorrectionActiveAdded =
pow(globalScaleCorrectionActiveAdded,0.07);
        } else {
            globalScaleCorrectionActive = pow(globalScaleCorrectionActive,0.16);
            globalScaleCorrectionActiveAdded =
pow(globalScaleCorrectionActiveAdded,0.07);
        }
    }

//END GLOBAL SCALE original

//FREQ Indicator

    ShowProgress (5, "Calculating Frequency Indicator");

    CBarkSpectrum   originalPitchPowerDensityPureFrq,
distortedPitchPowerDensityPureFrq, originalLoudnessDensityPureFrq,
distortedLoudnessDensityPureFrq;
    CBarkSpectrum   originalPitchPowerDensityAvgPureFrq,
distortedPitchPowerDensityAvgPureFrq;
    CBarkSpectrum   originalPitchLoudnessDensityAvgPureFrq,
distortedPitchLoudnessDensityAvgPureFrq;

    originalPitchPowerDensityPureFrq. Initialize ("originalPitchPowerDensityPureFrq",
POLQAHandle);
    distortedPitchPowerDensityPureFrq. Initialize ("distortedPitchPowerDensityPureFrq",
POLQAHandle);
    originalLoudnessDensityPureFrq. Initialize ("originalLoudnessDensityPureFrq",
POLQAHandle);
    originalPitchPowerDensityAvgPureFrq. Initialize
("originalPitchPowerDensityAvgPureFrq", POLQAHandle);
    distortedPitchPowerDensityAvgPureFrq. Initialize
("distortedPitchPowerDensityAvgPureFrq", POLQAHandle);
    originalPitchLoudnessDensityAvgPureFrq. Initialize
("originalPitchLoudnessDensityAvgPureFrq", POLQAHandle);
    distortedPitchLoudnessDensityAvgPureFrq. Initialize
("distortedPitchLoudnessDensityAvgPureFrq", POLQAHandle);
    distortedLoudnessDensityPureFrq. Initialize ("distortedLoudnessDensityPureFrq",
POLQAHandle);

    originalPitchPowerDensityPureFrq. FrequencyWarpingOf (POLQAHandle,
originalHzPowerSpectrum, 1.0);
    distortedPitchPowerDensityPureFrq. FrequencyWarpingOf (POLQAHandle,
distortedHzPowerSpectrum, PitchRatio);

    hulp1 = 0.0;
    hulp2 = 0.0;
```

```
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        if (aActiveRatioOk.m_pData[frameIndex]) {
            hulp1 += originalPitchPowerDensityPureFrq. Total (frameIndex, 300.0,
5000.0);
            hulp2 += distortedPitchPowerDensityPureFrq. Total (frameIndex, 300.0,
5000.0);
        }
    }
    hulp1 /= (numberOfSpeechFrames + 0.01);
    hulp2 /= (numberOfSpeechFrames + 0.01);
    hulp1 = 3.0e8/(hulp1*aGlobalCompensation1 + 1.0);
    hulp2 = 3.0e8/(hulp2*aGlobalCompensation1 + 1.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
    {
        originalPitchPowerDensityPureFrq. MultiplyWith (frameIndex, hulp1);
        distortedPitchPowerDensityPureFrq. MultiplyWith (frameIndex, hulp2);
    }
    distortedLoudnessDensityPureFrq. IntensityWarpingOf (POLQAHandle,
distortedPitchPowerDensityPureFrq);

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
    {
        aOriginalTotalPower.m_pData[frameIndex] = originalPitchPowerDensityPureFrq.
TotalAudible (POLQAHandle, frameIndex, 1.0e2);
        aDistortedTotalPower.m_pData[frameIndex] = distortedPitchPowerDensityPureFrq.
TotalAudible (POLQAHandle, frameIndex, 1.0e2);
        hulp = (aDistortedTotalPower.m_pData[frameIndex] + 100.0) /
(aOriginalTotalPower.m_pData[frameIndex]+100.0);
        if (hulp<1.0) hulp = 1/hulp;
        if  ( (aOriginalTotalPower.m_pData[frameIndex] > 1.0E5) &&
(aDistortedTotalPower.m_pData[frameIndex] > 1.0E5) && (hulp < 24.0) )
            aBothActive.m_pData[frameIndex] = TRUE;
        else
            aBothActive.m_pData[frameIndex] = FALSE;
    }

    for (frameIndex = (statics->startFrameIdx +2); frameIndex <=
(statics->stopFrameIdx); frameIndex++) {
        if (!aBothActive.m_pData[frameIndex]) {
            aBothActive.m_pData[frameIndex-1] = FALSE;
            aBothActive.m_pData[frameIndex-2] = FALSE;
        }
    }
    for (frameIndex = (statics->stopFrameIdx -2); frameIndex >=
(statics->startFrameIdx); frameIndex--) {
        if (!aBothActive.m_pData[frameIndex]) {
            aBothActive.m_pData[frameIndex+1] = FALSE;
            aBothActive.m_pData[frameIndex+2] = FALSE;
        }
    }

    originalPitchPowerDensityAvgPureFrq.  TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensityPureFrq,  aSilent, 1.0);
    distortedPitchPowerDensityAvgPureFrq. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensityPureFrq, aSilent, 1.0);
    PrintFrequencyResponse (aResultsFile, originalPitchPowerDensityAvgPureFrq,
distortedPitchPowerDensityAvgPureFrq);

    originalLoudnessDensityPureFrq. IntensityWarpingOf (POLQAHandle,
originalPitchPowerDensityPureFrq);

    originalPitchLoudnessDensityMainAvg.  TimeLpAudibleOfSilent
(originalLoudnessDensityPureFrq,  aSilent, 3.0, numberOfSilentFrames);
    distortedPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(distortedLoudnessDensityPureFrq, aSilent, 3.0, numberOfSilentFrames);

    originalLoudnessDensityPureFrq.  AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg,  0.3);
    distortedLoudnessDensityPureFrq. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
distortedPitchLoudnessDensityMainAvg, 0.3);

    originalPitchLoudnessDensityAvgPureFrq. TimeLpOf (POLQAHandle,
originalLoudnessDensityPureFrq,  aBothActive, 2.0);
```

```
    distortedPitchLoudnessDensityAvgPureFrq. TimeLpOf(POLQAHandle,
distortedLoudnessDensityPureFrq, aBothActive, 2.0);

    {
        SmartBufferPolqa SB1(POLQAHandle, statics->aNumberOfBarkBands);
        XFLOAT *temp1 = SB1.Buffer;
        SmartBufferPolqa SB2(POLQAHandle, statics->aNumberOfBarkBands);
        XFLOAT *temp2 = SB2.Buffer;

        matbSqrt2(originalPitchLoudnessDensityAvgPureFrq.m_pData[0], temp1,
statics->aNumberOfBarkBands);
        matbSqrt2(temp1, temp2, statics->aNumberOfBarkBands);
        aOriginalLoudnessPureFrqMean = pow( matSum(temp2, statics->aNumberOfBarkBands),
4);

        matbSqrt2(distortedPitchLoudnessDensityAvgPureFrq.m_pData[0], temp1,
statics->aNumberOfBarkBands);
        matbSqrt2(temp1, temp2, statics->aNumberOfBarkBands);
        aDistortedLoudnessPureFrqMean = pow( matSum(temp2,
statics->aNumberOfBarkBands), 4);

        aLoudnessPureFrqScaling = (aOriginalLoudnessPureFrqMean+1e-10) /
(aDistortedLoudnessPureFrqMean+1e-10);
    }

    loudnessPureFrqVar = 0.0;
    hulpMem = 0.0;

    for (bandIndex = 0; bandIndex < statics->aNumberOfBarkBands; bandIndex++)
    {
            hulp = ( (originalPitchLoudnessDensityAvgPureFrq.m_pData[0][bandIndex] -
aLoudnessPureFrqScaling*distortedPitchLoudnessDensityAvgPureFrq.m_pData[0][
bandIndex]) );
            if (hulp<0) hulp1 = - ((-hulp) * sqrt(-hulp));
            else if (hulp>=0) hulp1 = hulp * sqrt(hulp);
            if (hulpMem<0) hulp2 = - ((-hulpMem) * sqrt(-hulpMem));
            else if (hulpMem>=0) hulp2 = hulpMem * sqrt(hulpMem);

            loudnessPureFrqVar += fabs(hulp1-hulp2) * pow((bandIndex+1.0),0.4);

            hulpMem = hulp;
    }

    aPureFrqLoudnessMean = 0.0;
    for (bandIndex = 0; bandIndex < statics->aNumberOfBarkBands; bandIndex++)
    {
        if ( (originalPitchLoudnessDensityAvgPureFrq.m_pData[0][bandIndex]) >
(aLoudnessPureFrqScaling*distortedPitchLoudnessDensityAvgPureFrq.m_pData[0][ban
dIndex]) )
        {
            hulp = (originalPitchLoudnessDensityAvgPureFrq.m_pData[0][bandIndex] -
aLoudnessPureFrqScaling*distortedPitchLoudnessDensityAvgPureFrq.m_pData[0][
bandIndex]);
            aPureFrqLoudnessMean += sqrt(hulp);
        }
        else
        {
            hulp =
(aLoudnessPureFrqScaling*distortedPitchLoudnessDensityAvgPureFrq.m_pData[0]
[bandIndex] -
originalPitchLoudnessDensityAvgPureFrq.m_pData[0][bandIndex]);

            if (aListeningCondition==WIDE_H) {
                aPureFrqLoudnessMean += (0.2*sqrt(hulp));
            } else {
                aPureFrqLoudnessMean += (0.1*sqrt(hulp));
            }

        }
    }
    aPureFrqLoudnessMean = aPureFrqLoudnessMean * loudnessPureFrqVar *
aPowerRatioaAvgProduct;
    loudnessPureFrqVar1 = loudnessPureFrqVar;
    loudnessPureFrqVar2 = log10(loudnessPureFrqVar+0.0001);
    aPowerRatioaAvgProduct1 = aPowerRatioaAvgProduct;
    aPowerRatioaAvgProduct2 = log10(aPowerRatioaAvgProduct+0.0001);
```

```
    XFLOAT aPureFrqLoudnessMeanCompensation;
    aPureFrqLoudnessMeanCompensation = pow((aPureFrqLoudnessMean+1.0),0.01);
//END FREQ Indicator

    CheckTimeMatInit(POLQAHandle->mh, 3);
//REVERBERATION indicator
    reverbIndicator = aOriginalTimeSeriesReverb.ReverberationIndicator(POLQAHandle,
statics->sampleRate, aOriginalTimeSeries, aDistortedTimeSeries);
    CheckTimeMatEval(POLQAHandle->mh, 3, &ClockCycles, &TimeDiff);
    AddProcessingTime(pOverviewHolder, "Reverberation indicator", TimeDiff,
ClockCycles);

    CheckTimeMatInit(POLQAHandle->mh, 3);
//POLQAMAIN PART 0
    aScale.Initialize("aScale", mMaxModelFrames);

int numberOfUsedFrames = 0;
    count = 0;
    scaleDistortion = 0.0;
    oldOldScale = 1.0;
    oldScale = 1.0;
    minimumOriginalFramePower = 10000000.0;
    MaxScale = 1.0;
    MinScale = 1.0;
    MinMinScale = 0.4;
    for (frameIndex = (statics->startFrameIdx); frameIndex <= (statics->stopFrameIdx);
frameIndex++)     {
        if (UseThisFrame[frameIndex]) {
            aOriginalTotalPower.m_pData[frameIndex] = originalPitchPowerDensity. Total
(frameIndex, 0.0, 3.0e4);
            aDistortedTotalPower.m_pData[frameIndex] = distortedPitchPowerDensity.
Total (frameIndex, 0.0, 3.0e4);
            if (aOriginalTotalPower.m_pData[frameIndex] < minimumOriginalFramePower)
{minimumOriginalFramePower = aOriginalTotalPower.m_pData[frameIndex];}
            scale = (aDistortedTotalPower.m_pData[frameIndex] + (XFLOAT) 6.0e3) /
(aOriginalTotalPower.m_pData[frameIndex] + (XFLOAT) 6.0e3) ;
            if ( frameIndex>10 && aActiveFreqresponse.m_pData[frameIndex] &&
oldScale>scale ) {
                scaleDistortion += fabs(oldScale - scale);
                count++;
            }

            if (scale > MaxScale) scale = MaxScale;
            if (scale < MinMinScale) scale = MinMinScale;
            aScale.m_pData[frameIndex] = scale;
            if (aListeningCondition==WIDE_H) {
                scale = (XFLOAT) 0.2 * oldOldScale + (XFLOAT) 0.3 * oldScale + (XFLOAT)
0.5 * scale;
                oldOldScale = oldScale;
                oldScale = scale;
                originalPitchPowerDensity. MultiplyWith (frameIndex, pow(scale,
0.7*globalScaleDistortedToFixedlevelHulp));
            } else {
                scale = (XFLOAT) 0.35 * oldOldScale + (XFLOAT) 0.35 * oldScale +
(XFLOAT) 0.3 * scale;
                oldOldScale = oldScale;
                oldScale = scale;
                originalPitchPowerDensity. MultiplyWith (frameIndex, pow(scale,
0.45*globalScaleDistortedToFixedlevelHulp));
            }

            numberOfUsedFrames++;
        } else {
            originalPitchPowerDensity. MultiplyWith (frameIndex, 0.0);
            distortedPitchPowerDensity. MultiplyWith (frameIndex, 0.0);
        }
    }
    scaleDistortion /= (count+0.1);
    XFLOAT fractionOfUsedFrames;
    if ( numberOfUsedFrames > statics->startFrameIdx ) {
        fractionOfUsedFrames = numberOfUsedFrames/(numberOfSpeechFrames+1.0);
    } else {
        fractionOfUsedFrames = 1.0;
    }
    XFLOAT fractionOfSilentFrames =
(numberOfSilentFrames+1.0)/(numberOfUsedFrames+1.0);
```

```
    if (fractionOfSilentFrames>0.5) fractionOfSilentFrames= 0.5;

    if (aListeningCondition==STANDARD_IRS) {

        originalPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensity, aActiveFreqresponse, 0.8);
        distortedPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensity, aActiveFreqresponse, 0.8);
        originalPitchPowerDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchPowerDensityMainAvg,
                                                                  distortedPitchPowerDensit
yMainAvg, 8.0E5, 0.8,
                                                                  statics->listeningConditi
on);

    }
    if (aListeningCondition==WIDE_H) {

        if (maxFreqBarkSource<22.0) {
            originalPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensity, aActiveFreqresponse, 0.7);
            distortedPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensity, aActiveFreqresponse, 0.7);
            originalPitchPowerDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchPowerDensityMainAvg,
                                                                  distortedPitchPowerDensit
yMainAvg, 3.0E4, 0.4,
                                                                  statics->listeningConditi
on);
        } else {
            originalPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensity, aActiveFreqresponse, 0.6);
            distortedPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensity, aActiveFreqresponse, 0.6);
            originalPitchPowerDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchPowerDensityMainAvg,
                                                                  distortedPitchPowerDensit
yMainAvg, 1.0E4, 0.7,
                                                                  statics->listeningConditi
on);
        }

    }
    if (aListeningCondition==NARROW_H) {
        originalPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensity, aActiveFreqresponse, 0.5);
        distortedPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensity, aActiveFreqresponse, 0.5);
        originalPitchPowerDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchPowerDensityMainAvg,
                                                                  distortedPitchPowerDensit
yMainAvg, 1.0E5, 0.6,
                                                                  statics->listeningConditi
on);
    }

    smearedOriginalPitchPowerDensity. ExcitationOf (POLQAHandle,
originalPitchPowerDensity, UseThisFrame, statics->listeningCondition);
    smearedDistortedPitchPowerDensity. ExcitationOf (POLQAHandle,
distortedPitchPowerDensity, UseThisFrame, statics->listeningCondition);

    originalLoudnessDensity. IntensityWarpingOf (POLQAHandle,
smearedOriginalPitchPowerDensity);
    distortedLoudnessDensity. IntensityWarpingOf (POLQAHandle,
smearedDistortedPitchPowerDensity);

//END POLQAMAIN PART 0

//NOISE Indicator

    ShowProgress (3, "Calculating Noise Indicator");

    CBarkSpectrum     originalPitchPowerDensitySilent,
distortedPitchPowerDensitySilent;
    CBarkSpectrum     originalLoudnessDensitySilent, distortedLoudnessDensitySilent;
    CBarkSpectrum     disturbanceDensityAddSilent;
```

```
    CTimeSeries       aNoise;

    originalPitchPowerDensitySilent. Initialize ("originalPitchPowerDensitySilent",
POLQAHandle);
    distortedPitchPowerDensitySilent. Initialize ("distortedPitchPowerDensitySilent",
POLQAHandle);
    originalLoudnessDensitySilent. Initialize ("originalLoudnessDensitySilent",
POLQAHandle);
    distortedLoudnessDensitySilent. Initialize ("distortedLoudnessDensitySilent",
POLQAHandle);
    disturbanceDensityAddSilent. Initialize ("disturbanceDensityAddSilent",
POLQAHandle);

    originalPitchPowerDensitySilent. FrequencyWarpingOf (POLQAHandle,
originalHzPowerSpectrum, 1.0);
    distortedPitchPowerDensitySilent. FrequencyWarpingOf (POLQAHandle,
distortedHzPowerSpectrum, PitchRatio);

    originalPitchPowerDensityPureFrq. FrequencyWarpingOf (POLQAHandle,
originalHzPowerSpectrum, 1.0);
    distortedPitchPowerDensityPureFrq. FrequencyWarpingOf (POLQAHandle,
distortedHzPowerSpectrum, PitchRatio);

    hulp1 = 0.0;
    hulp2 = 0.0;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        if (aActiveRatioOk.m_pData[frameIndex]) {
            hulp1 += originalPitchPowerDensitySilent. TotalAudible (POLQAHandle,
frameIndex, 1.0);
            hulp2 += distortedPitchPowerDensitySilent. TotalAudible (POLQAHandle,
frameIndex, 1.0);
        }
    }
    hulp1 /= (numberOfSpeechFrames + 0.01);
    hulp2 /= (numberOfSpeechFrames + 0.01);
    hulp1 = 1.0e6/(hulp1+1.0);
    hulp2 = 1.0e6/(hulp2+1.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
    {
        originalPitchPowerDensitySilent. MultiplyWith (frameIndex, hulp1);
        distortedPitchPowerDensitySilent. MultiplyWith (frameIndex, hulp2);
    }

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        if ( originalPitchPowerDensitySilent. TotalAudible (POLQAHandle, frameIndex,
1.0) < 3.0E6)
            aActive.m_pData[frameIndex] = FALSE; else
            aActive.m_pData[frameIndex] = TRUE;
    }

    originalLoudnessDensitySilent. IntensityWarpingOf (POLQAHandle,
originalPitchPowerDensitySilent);
    distortedLoudnessDensitySilent. IntensityWarpingOf (POLQAHandle,
distortedPitchPowerDensitySilent);

    originalPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
originalLoudnessDensitySilent, aActiveRatioOk, 1.0);
    distortedPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
distortedLoudnessDensitySilent, aActiveRatioOk, 1.0);
    originalLoudnessDensitySilent. AudibleFreqRespCompensationExact
(originalPitchLoudnessDensityMainAvg,
                                                                distortedPitchLoudnessDen
sityMainAvg, 0.1);

    disturbanceDensityAddSilent. DifferenceOfBandlimited
(distortedLoudnessDensitySilent, originalLoudnessDensitySilent);

    disturbanceDensityAddSilent. Orthogonalize (aActive);

    disturbanceDensityAddSilent. ComputeLpWeights (POLQAHandle, MINIMUM_POWER_FREQ,
STEP_POWER_FREQ, NUMBER_OF_POWERS_OVER_FREQ, aAddedSilentDisturbance);

    XFLOAT noiseIndicatorAlignJumps, noiseIndicatorAlignJumpsMax, noiseIndicatorTimbre,
```

```
noiseIndicatorTimbreAdd;
    XFLOAT noiseIndicator, noiseIndicatorHighBands, noiseIndicatorPulsImpact,
noiseIndicatorFreqImpact, noiseIndicatorScalingImpact, signalLoudness,
signalLoudnessDistHighBands;
    XFLOAT delayJumpCompNB, delayJumpCompWB;
    int noiseIndicatorAlignJumpsIntNB, noiseIndicatorAlignJumpsIntWB;
    noiseIndicator = 0.0;
    signalLoudness = 0.0;
    noiseIndicatorHighBands = 0.0;
    signalLoudnessDistHighBands = 0.0;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        if (aSilent[frameIndex]) {
            noiseIndicator += disturbanceDensityAddSilent. Total (frameIndex, 200.0,
3500.0);
            noiseIndicatorHighBands += disturbanceDensityAddSilent. Total (frameIndex,
3000.0, 3.0e4);
        } else {
            signalLoudness += originalLoudnessDensity. Total (frameIndex, 0.0, 3.0e4);
            signalLoudnessDistHighBands += distortedLoudnessDensity. Total (frameIndex,
3000.0, 3.0e4);
        }
    }
    if (noiseIndicator<0.0) noiseIndicator = 0.0;
    noiseIndicator /= (numberOfSilentFrames+0.01);

    noiseIndicatorPulsImpact = noiseIndicator;
    if (noiseIndicatorPulsImpact<1.0) noiseIndicatorPulsImpact = 1.0;
    if (noiseIndicatorPulsImpact>10.0) noiseIndicatorPulsImpact = 10.0;
    noiseIndicatorPulsImpact = pow(noiseIndicatorPulsImpact,0.1);

    noiseIndicatorFreqImpact = noiseIndicator;
    if (noiseIndicatorFreqImpact<1.0) noiseIndicatorFreqImpact = 1.0;
    noiseIndicatorFreqImpact = pow(noiseIndicatorFreqImpact,0.1);

    noiseIndicatorScalingImpact = noiseIndicator - 0.3;
    if (noiseIndicatorScalingImpact<0.0) noiseIndicatorScalingImpact = 0.0;
    if (noiseIndicatorScalingImpact>20.0) noiseIndicatorScalingImpact = 20.0;
    noiseIndicatorScalingImpact = pow(noiseIndicatorScalingImpact,0.5);

    if (noiseIndicatorHighBands<0.0) noiseIndicatorHighBands = 0.0;
    noiseIndicatorHighBands /= (numberOfSilentFrames+0.01);
    if (noiseIndicatorHighBands>2.0) noiseIndicatorHighBands = 2.0;
    signalLoudnessDistHighBands /= (numberOfNotSilentFrames+0.01);
    hulp = signalLoudnessDistHighBands - noiseIndicatorHighBands;
    if (hulp<11.0) hulp = 11.0;
    noiseIndicatorHighBandsCompensation000 = 1.2*noiseIndicatorHighBands/hulp;
    s. Format ("noiseIndicatorHighBandsCompensation000=%f  noiseIndicatorHighBands=%f
signalLoudnessDistHighBands=%f  \n ", noiseIndicatorHighBandsCompensation000,
noiseIndicatorHighBands, signalLoudnessDistHighBands);
    gLogFile. WriteString (s);

    noiseIndicatorAlignJumpsMax = 300.0;
    noiseIndicatorAlignJumps = (noiseIndicator-77.0);
    if (noiseIndicatorAlignJumps<1.0) noiseIndicatorAlignJumps = 1.0;

    if (noiseIndicatorAlignJumps>noiseIndicatorAlignJumpsMax) noiseIndicatorAlignJumps
= noiseIndicatorAlignJumpsMax;
    noiseIndicatorAlignJumpsIntNB = 4;
    noiseIndicatorAlignJumpsIntWB = 5;
    delayJumpCompWB = 19.0*pow(noiseIndicatorAlignJumps,0.04);
    delayJumpCompNB = 80.0;

    noiseIndicatorTimbre = (noiseIndicator-73.0);
    if (noiseIndicatorTimbre<1.0) noiseIndicatorTimbre = 1.0;
    noiseIndicatorTimbreAdd = (noiseIndicator-8.0);
    if (noiseIndicatorTimbreAdd<1.0) noiseIndicatorTimbreAdd = 1.0;
    if (noiseIndicatorTimbre>noiseIndicatorAlignJumpsMax) noiseIndicatorTimbre =
noiseIndicatorAlignJumpsMax;
    if (noiseIndicatorTimbreAdd>noiseIndicatorAlignJumpsMax) noiseIndicatorTimbreAdd =
noiseIndicatorAlignJumpsMax;
    noiseIndicatorTimbre = pow(noiseIndicatorTimbre,0.2);
    noiseIndicatorTimbreAdd = pow(noiseIndicatorTimbreAdd,0.3)-0.3;

    signalLoudness /= ((numberOfSpeechFrames-numberOfSilentFrames)+0.01);
    noiseIndicator /= (signalLoudness+0.01);
```

```
    if (noiseIndicator<0.2) noiseIndicator = 0.2;
    noiseIndicator = pow(noiseIndicator,0.1);

//END NOISE Indicator

CheckTimeMatInit(POLQAHandle->mh, 3);
//POLQAMAIN PART 1
    aDistortedLoudnessMeanIndicator1 = 0.0;
    distortedLoudnessTimbrePerFrameLoudAvg = 0.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){

        aOriginalLoudness.m_pData[frameIndex] = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle,  frameIndex, 2.0, bandIdxLow,
bandIdxHigh);
        aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle,  frameIndex, 2.0, bandIdxLow,
bandIdxHigh);

        aDistortedLoudnessMeanIndicator1 +=
pow(aDistortedLoudness.m_pData[frameIndex],0.3);
        if (aSuperLoud.m_pData[frameIndex]) {
            hulp1  = (distortedLoudnessDensity.IntegralLowFrameLoud (frameIndex,
statics->listeningCondition)-distortedLoudnessDensity.IntegralHighFrameLoud
(frameIndex, statics->listeningCondition));
            distortedLoudnessTimbrePerFrameLoudAvg += hulp1;
        }

    }
    aDistortedLoudnessMeanIndicator1 /= ( numberOfSpeechFrames + 0.01);
    aDistortedLoudnessMeanIndicator1 = pow(aDistortedLoudnessMeanIndicator1,(1.0/0.3));
    distortedLoudnessTimbrePerFrameLoudAvg /= (numberOfaSuperLoud+0.1);
    distortedLoudnessTimbrePerFrameLoudAvg000 =
distortedLoudnessTimbrePerFrameLoudAvg/3000.0;

    originalPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(originalLoudnessDensity, aSuperSilent, 4.0, numberOfSuperSilentFrames);
    distortedPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(distortedLoudnessDensity, aSuperSilent, 4.0, numberOfSuperSilentFrames);

    if (aListeningCondition==STANDARD_IRS) {
        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg, 0.3);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2
(distortedPitchLoudnessDensityMainAvg, 0.35, 0.8);
    } else {
        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg, 0.28/noiseIndicatorFreqImpact);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2
(distortedPitchLoudnessDensityMainAvg, 0.32*delayVariationCompensation, 1.0);
    }

//Local loudness scaling original, predominantly for modelling the impact of time clip
effects during active intervals

    oldOldLoudnessScaleLow = 1.0;
    oldLoudnessScaleLow = 1.0;
    if (aListeningCondition==WIDE_H) {
        bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
        bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    } else {
        bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 2.0);
        bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(18.0);
    }
    for (frameIndex = (statics->startFrameIdx); frameIndex <= (statics->stopFrameIdx);
frameIndex++)     {
        if (UseThisFrame[frameIndex]) {
            if (aListeningCondition==WIDE_H) {
                distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.2, bandIdxLow,
bandIdxHigh);
                originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, 1.25, bandIdxLow, bandIdxHigh);
                loudnessScaleLow = (distortedLoudnessHulp + 8.0)/(originalLoudnessHulp
```

```
+ 8.0);
                loudnessScaleLow =
0.05*noiseIndicatorScalingImpact*oldOldLoudnessScaleLow +
0.08*noiseIndicatorScalingImpact*oldLoudnessScaleLow +
(1.0-0.13*noiseIndicatorScalingImpact)*loudnessScaleLow;
                if (loudnessScaleLow<0.02) loudnessScaleLow = 0.02;
                if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;
                if ( (oldOldLoudnessScaleLow < oldLoudnessScaleLow) &&
(oldLoudnessScaleLow < loudnessScaleLow) ) {
                    originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.55), 0.0, 99.0);
                } else {
                    if ( (oldOldLoudnessScaleLow > oldLoudnessScaleLow) &&
(oldLoudnessScaleLow > loudnessScaleLow) ) {
                        originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.55), 0.0, 99.0);
                    } else {
                        originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.75), 0.0, 99.0);
                    }
                }
                oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                oldLoudnessScaleLow = loudnessScaleLow;
            } else {
                distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.1, bandIdxLow,
bandIdxHigh);
                originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, 1.2, bandIdxLow, bandIdxHigh);
                loudnessScaleLow = (distortedLoudnessHulp + 10.0)/(originalLoudnessHulp
+ 10.0);
                loudnessScaleLow = 0.06*oldOldLoudnessScaleLow +
0.14*oldLoudnessScaleLow + 0.8*loudnessScaleLow;
                oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                oldLoudnessScaleLow = loudnessScaleLow;
                if (loudnessScaleLow<0.02) loudnessScaleLow = 0.02;
                if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;
                originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.65), 0.0, 99.0);
            }
        } else {
            originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 0.0,
99.0);
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 0.0,
99.0);
        }
    }

//Local loudness scaling distorted, predominantly for modelling the local impact of
additive noise and pulses during silent intervals

    oldOldLoudnessScaleLow = 1.0;
    oldLoudnessScaleLow = 1.0;
    if (aListeningCondition==WIDE_H) {
        if (maxFreqBarkSource<22.0) {
            if (maxFreqBarkSource<18.0) {
                bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 1.5);
                bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(15.0);
                for (frameIndex = (statics->startFrameIdx); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
                    if (UseThisFrame[frameIndex]) {
                        distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.3,
bandIdxLow, bandIdxHigh);
                        originalLoudnessHulp = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.3,
bandIdxLow, bandIdxHigh);
                        hulp = originalLoudnessHulp;
                        if (hulp>1.0) hulp = 1.0;
                        loudnessScaleLow = (originalLoudnessHulp + 2.0 -
hulp)/(distortedLoudnessHulp + 2.0 - hulp);
                        loudnessScaleLow = 0.02*oldOldLoudnessScaleLow +
0.03*oldLoudnessScaleLow + 0.95*loudnessScaleLow;
                        oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                        oldLoudnessScaleLow = loudnessScaleLow;
                        if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;
```

```
                        distortedLoudnessDensity. MultiplyWithOverBandRange
(frameIndex, pow(loudnessScaleLow, 0.7), 0.0, 99.0);
                    } else {
                        originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
0.0, 0.0, 99.0);
                        distortedLoudnessDensity. MultiplyWithOverBandRange
(frameIndex, 0.0, 0.0, 99.0);
                    }
                }
            } else {
                bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 3.0);
                bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(16.0);
                for (frameIndex = (statics->startFrameIdx); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
                    if (UseThisFrame[frameIndex]) {
                        distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.3,
bandIdxLow, bandIdxHigh);
                        originalLoudnessHulp = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.3,
bandIdxLow, bandIdxHigh);
                        hulp = originalLoudnessHulp;
                        if (hulp>1.0) hulp = 1.0;
                        loudnessScaleLow = (originalLoudnessHulp + 2.0 -
hulp)/(distortedLoudnessHulp + 2.0 - hulp);
                        loudnessScaleLow = 0.02*oldOldLoudnessScaleLow +
0.03*oldLoudnessScaleLow + 0.95*loudnessScaleLow;
                        if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;
                        if ( (oldOldLoudnessScaleLow < oldLoudnessScaleLow) &&
(oldLoudnessScaleLow < loudnessScaleLow) ) {
                            distortedLoudnessDensity. MultiplyWithOverBandRange
(frameIndex, pow(loudnessScaleLow, 0.7), 0.0, 99.0);
                        } else {
                            if ( (oldOldLoudnessScaleLow > oldLoudnessScaleLow) &&
(oldLoudnessScaleLow > loudnessScaleLow) ) {
                                distortedLoudnessDensity. MultiplyWithOverBandRange
(frameIndex, pow(loudnessScaleLow, 0.75), 0.0, 99.0);
                            } else {
                                distortedLoudnessDensity. MultiplyWithOverBandRange
(frameIndex, pow(loudnessScaleLow, 0.65), 0.0, 99.0);
                            }
                        }
                        oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                        oldLoudnessScaleLow = loudnessScaleLow;
                    } else {
                        originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
0.0, 0.0, 99.0);
                        distortedLoudnessDensity. MultiplyWithOverBandRange
(frameIndex, 0.0, 0.0, 99.0);
                    }
                }
            }
        } else {
            bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 3.0);
            bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(16.0);
            for (frameIndex = (statics->startFrameIdx); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
                if (UseThisFrame[frameIndex]) {
                    distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.3, bandIdxLow,
bandIdxHigh);
                    originalLoudnessHulp = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.3, bandIdxLow,
bandIdxHigh);
                    hulp = originalLoudnessHulp;
                    if (hulp>1.0) hulp = 1.0;
                    loudnessScaleLow = (originalLoudnessHulp + 2.0 -
hulp)/(distortedLoudnessHulp + 2.0 - hulp);
                    loudnessScaleLow = 0.02*oldOldLoudnessScaleLow +
0.03*oldLoudnessScaleLow + 0.95*loudnessScaleLow;
                    if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;
                    if ( (oldOldLoudnessScaleLow < oldLoudnessScaleLow) &&
(oldLoudnessScaleLow < loudnessScaleLow) ) {
                        distortedLoudnessDensity. MultiplyWithOverBandRange
(frameIndex, pow(loudnessScaleLow, 0.6), 0.0, 99.0);
                    } else {
```

```
                        if ( (oldOldLoudnessScaleLow > oldLoudnessScaleLow) &&
(oldLoudnessScaleLow > loudnessScaleLow) ) {
                            distortedLoudnessDensity. MultiplyWithOverBandRange
(frameIndex, pow(loudnessScaleLow, 0.65), 0.0, 99.0);
                        } else {
                            distortedLoudnessDensity. MultiplyWithOverBandRange
(frameIndex, pow(loudnessScaleLow, 0.55), 0.0, 99.0);
                        }
                    }
                    oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                    oldLoudnessScaleLow = loudnessScaleLow;
                } else {
                    originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
0.0, 0.0, 99.0);
                    distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
0.0, 0.0, 99.0);
                }
            }
        }
    } else {
        bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 3.5);
        bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(16.0);
        for (frameIndex = (statics->startFrameIdx); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
            if (UseThisFrame[frameIndex]) {
                distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.4, bandIdxLow,
bandIdxHigh);
                originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, 1.4, bandIdxLow, bandIdxHigh);
                hulp = originalLoudnessHulp;
                if (hulp>1.0) hulp = 1.0;
                loudnessScaleLow = (originalLoudnessHulp + 2.0 -
hulp)/(distortedLoudnessHulp + 2.0 - hulp);
                oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                oldLoudnessScaleLow = loudnessScaleLow;
                if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;
                distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.5), 0.0, 99.0);
            } else {
                originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0,
0.0, 99.0);
                distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0,
0.0, 99.0);
            }
        }
    }

//Frequency response compensation in loudness domain

    if (aListeningCondition==STANDARD_IRS) {
        originalPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
originalLoudnessDensity, aActiveFreqresponse, 3.0);
        distortedPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
distortedLoudnessDensity, aActiveFreqresponse, 3.0);
        originalLoudnessDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchLoudnessDensityMainAvg,
                                                                distortedPitchLoudnessDen
sityMainAvg, 20.0, 0.5,
                                                                statics->listeningConditi
on);
    }
    if (aListeningCondition==WIDE_H) {
        originalPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
originalLoudnessDensity, aActiveFreqresponse, 3.0);
        distortedPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
distortedLoudnessDensity, aActiveFreqresponse, 3.0);
        originalLoudnessDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchLoudnessDensityMainAvg,
                                                                distortedPitchLoudnessDen
sityMainAvg, 20.0, 0.6,
                                                                statics->listeningConditi
on);
    }
    if (aListeningCondition==NARROW_H) {
        originalPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
```

```
originalLoudnessDensity, aActiveFreqresponse, 3.0);
        distortedPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
distortedLoudnessDensity, aActiveFreqresponse, 3.0);
        originalLoudnessDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchLoudnessDensityMainAvg,
                                                                distortedPitchLoudnessDen
sityMainAvg, 20.0, 0.5,
                                                                statics->listeningConditi
on);
    }

//set highest bands to zero

    if (maxFreqBarkSource<18.0) {
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 17.0,
99.0);
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 18.0,
99.0);
        }
    } else {
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 21.5,
99.0);
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 22.0,
99.0);
        }
    }

//Local loudness scaling in lowest bands

    oldLoudnessScaleLow = 1.0;
    if (aListeningCondition==WIDE_H) {
        bandIdxLow  = originalLoudnessDensity.GetBandLowIdx (0.0);
        bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(3.0);
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {

            distortedLoudnessHulp = distortedLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 10.0, bandIdxLow, bandIdxHigh);
            originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 10.0, bandIdxLow, bandIdxHigh);

            loudnessScaleLow = (originalLoudnessHulp + 0.1)/(distortedLoudnessHulp +
0.1);
            loudnessScaleLow = 0.05*oldLoudnessScaleLow + 0.95*loudnessScaleLow;
            oldLoudnessScaleLow = loudnessScaleLow;

        distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.2), 0.0, 3.0);
        }
    } else {
        bandIdxLow  = originalLoudnessDensity.GetBandLowIdx (0.0);
        bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(5.0);
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {

            distortedLoudnessHulp = distortedLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 10.0, bandIdxLow, bandIdxHigh);
            originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 10.0, bandIdxLow, bandIdxHigh);

            loudnessScaleLow = (originalLoudnessHulp + 0.1)/(distortedLoudnessHulp +
0.1);
            oldLoudnessScaleLow = loudnessScaleLow;
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.3), 0.0, 5.0);
        }
    }

//Partial Global loudness scaling distorted towards 20 sone
    aDistortedLoudnessMean = 0.0;

    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
```

```
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(17.0);
    if (aListeningCondition==WIDE_H) {

        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangePartial,
bandIdxLow, bandIdxHigh);
            aDistortedLoudnessMean += pow(aDistortedLoudness.m_pData[frameIndex], 0.9);
        }
        aDistortedLoudnessMean /= (numberOfSpeechFrames + 0.5);
        aLoudnessScalingDistorted =
fixedGlobalInternalLevel/(pow(aDistortedLoudnessMean,(1.0/0.9))+0.9);
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
            distortedLoudnessDensity. MultiplyWith (frameIndex,
pow(aLoudnessScalingDistorted,0.4));

    } else {
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangePartial,
bandIdxLow, bandIdxHigh);
            aDistortedLoudnessMean += pow(aDistortedLoudness.m_pData[frameIndex], 0.7);
        }
        aDistortedLoudnessMean /= (numberOfSpeechFrames + 0.5);
        aLoudnessScalingDistorted =
fixedGlobalInternalLevel/(pow(aDistortedLoudnessMean,(1.0/0.7))+0.9);
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
            distortedLoudnessDensity. MultiplyWith (frameIndex,
pow(aLoudnessScalingDistorted,0.4));
    }

// noise contrast calculation in the super silent frames
    noiseContrastParameter = 1.0;
    XFLOAT noiseContrastMax1 = 1.0;
    XFLOAT noiseContrastMax2 = 1.0;
    XFLOAT noiseContrastMax3 = 1.0;
    int frameIndexMax1 = 0;
    int frameIndexMax2 = 0;
    int frameIndexMax3 = 0;
    for (frameIndex = statics->startFrameIdx; (frameIndex <= statics->stopFrameIdx-7);
frameIndex++) {
        if ( aSuperSilent.m_pData[frameIndex] && aSuperSilent.m_pData[frameIndex+1] &&
aSuperSilent.m_pData[frameIndex+2] && aSuperSilent.m_pData[frameIndex+3] &&
aSuperSilent.m_pData[frameIndex+4] && aSuperSilent.m_pData[frameIndex+5] &&
aSuperSilent.m_pData[frameIndex+6] && aSuperSilent.m_pData[frameIndex+7]) {
            hulp1 = aDistortedLoudness.m_pData[frameIndex]+0.2;
            hulp2 = aDistortedLoudness.m_pData[frameIndex+3]+0.2;
            if (hulp1>1.5) hulp1 = 1.5;
            if (hulp2>1.5) hulp2 = 1.5;
            hulpRatio = (hulp2)/(hulp1);
            if (hulpRatio>9.0) hulpRatio = 9.0;\

            if (aListeningCondition==WIDE_H) {
                hulp1 = pow(hulpRatio,0.3);
            } else {
                hulp1 = pow(hulpRatio,0.4);
            }

            if ( (hulp1>noiseContrastMax3) ) {
                if ( (hulp1>noiseContrastMax2) ) {
                    if ( (hulp1>noiseContrastMax1) ) {
                        noiseContrastMax3 = noiseContrastMax2;
                        noiseContrastMax2 = noiseContrastMax1;
                        noiseContrastMax1 = hulp1;
                        frameIndexMax1 = frameIndex;
                    } else {
                        noiseContrastMax3 = noiseContrastMax2;
                        noiseContrastMax2 = hulp1;
                        frameIndexMax2 = frameIndex;
                    }
                } else {
                    noiseContrastMax3 = hulp1;
```

```
                    frameIndexMax3 = frameIndex;
                }

            }
            if (hulp1>1.0) noiseContrastParameter *= pow(hulp1,0.1);
        }
    }

//Complete global loudness scaling original towards distorted
    aOriginalLoudnessMean = 0.0;
    aDistortedLoudnessMean = 0.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            aOriginalLoudness.m_pData[frameIndex] = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangeComplete ,
bandIdxLow, bandIdxHigh);
            aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangeComplete,
bandIdxLow, bandIdxHigh);
            aOriginalLoudnessMean += pow(aOriginalLoudness.m_pData[frameIndex],
LpLoudnessMeanComplete);
            aDistortedLoudnessMean += pow(aDistortedLoudness.m_pData[frameIndex],
LpLoudnessMeanComplete);
    }

    //What you should do here:
    //Normalize aOriginalLoudnessMean and aDistortedLoudnessMean to the number of
speech frames plus 0.5

    aLoudnessScalingOriginal  =
(pow(aDistortedLoudnessMean,(1.0/LpLoudnessMeanComplete))+3.8) /
(pow(aOriginalLoudnessMean, (1.0/LpLoudnessMeanComplete))+3.8);

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
        originalLoudnessDensity. MultiplyWith (frameIndex, aLoudnessScalingOriginal);
    }

//Noise suppression in loudness densities using super silent frames for modelling the
global impact of additive noise during silent intervals

    originalPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(originalLoudnessDensity,  aSuperSilent, 1.0, numberOfSuperSilentFrames);
    distortedPitchLoudnessDensityMainAvg.TimeLpAudibleOfSilent
(distortedLoudnessDensity, aSuperSilent, 1.0, numberOfSuperSilentFrames);

    if (aListeningCondition==STANDARD_IRS) {

        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg,
1.2/(delayVariationCompensation*noiseContrastMax1));
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2
(distortedPitchLoudnessDensityMainAvg,
1.2*delayVariationCompensation/noiseContrastMax1, 0.4);

    }
    if (aListeningCondition==WIDE_H) {

        if (maxFreqBarkSource<22.0) {
            originalLoudnessDensity. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg, 1.0);
            distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2
(distortedPitchLoudnessDensityMainAvg, 0.6, 0.2);
        } else {

            originalLoudnessDensity. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg, 0.9/noiseContrastMax1);
            distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2
(distortedPitchLoudnessDensityMainAvg, 0.7/noiseContrastMax1, 0.2);

        }

    }
    if (aListeningCondition==NARROW_H) {
```

```
        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg, 1.2/noiseContrastMax1);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2
(distortedPitchLoudnessDensityMainAvg, 1.2/noiseContrastMax1, 0.3);
    }

    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
        aOriginalLoudness.m_pData[frameIndex] = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpLoudness, bandIdxLow,
bandIdxHigh);
        aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpLoudness, bandIdxLow,
bandIdxHigh);
    }

//Compute disturbance density for POLQA
    disturbanceDensity. DifferenceOf (distortedLoudnessDensity,
originalLoudnessDensity);

    mask. MinimumOf (distortedLoudnessDensity, originalLoudnessDensity);

    mask *= (XFLOAT) 0.25;

    disturbanceDensity. MaskWith (POLQAHandle, mask);

    CIntArray frameWasSkipped;
    frameWasSkipped. SetSize (statics->nrFrames);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)     {
        frameWasSkipped.m_pData[frameIndex] = FALSE;
    }

   for (utt = 1; utt < aStartSampleUtterance. GetSize (); utt++) {
        int startFrame = (int) floor ((aStartSampleUtterance.m_pData[utt] +
aDelayUtterance.m_pData[utt]) / (0.5 * aTransformLength));
        if (startFrame > (int) floor ((aStopSampleUtterance.m_pData[utt-1]+
aDelayUtterance.m_pData[utt-1]) / (0.5 * aTransformLength)))     {
            startFrame = (int) floor ((aStopSampleUtterance.m_pData[utt-1] +
aDelayUtterance.m_pData[utt-1])/ (0.5 * aTransformLength));
        }

        if (startFrame < 0) {
            startFrame = 0;
        }

        int delayJumpInSamples = aDelayUtterance.m_pData[utt] -
aDelayUtterance.m_pData[utt-1];

        if (delayJumpInSamples < -(int) (aTransformLength * 0.5)) {
            int stopFrame = (int) ((aStartSampleUtterance.m_pData[utt] + (((0) > (fabs
((XFLOAT)delayJumpInSamples))) ? (0) : (fabs
((XFLOAT)delayJumpInSamples)))) / ((XFLOAT)0.5 * aTransformLength)) + 1;
        }
    }

    frameWasSkipped. SetSize (statics->nrFrames);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)     {
        frameWasSkipped.m_pData[frameIndex] = FALSE;
    }

    for (utt=1; utt < aStartSampleUtterance. GetSize (); utt++) {
        int startFrame = (int) floor ((aStartSampleUtterance.m_pData[utt] +
aDelayUtterance.m_pData[utt]) / (0.5 * aTransformLength));
        if (startFrame > (int) floor ((aStopSampleUtterance.m_pData[utt-1]+
aDelayUtterance.m_pData[utt-1]) / (0.5 * aTransformLength)))     {
            startFrame = (int) floor ((aStopSampleUtterance.m_pData[utt-1] +
aDelayUtterance.m_pData[utt-1])/ (0.5 * aTransformLength));
        }

        if (startFrame < 0) {
            startFrame = 0;
        }
```

```
        int delayJumpInSamples = aDelayUtterance.m_pData[utt] -
aDelayUtterance.m_pData[utt-1];

        if (delayJumpInSamples < -(int) (aTransformLength * 0.5)) {
            int stopFrame = (int) ((aStartSampleUtterance.m_pData[utt] + (((0) > (fabs
((XFLOAT)delayJumpInSamples))) ? (0) : (fabs
((XFLOAT)delayJumpInSamples)))) / ((XFLOAT)0.5 * aTransformLength)) + 1;
        }
    }

    disturbanceDensity. ComputeLpWeights (POLQAHandle, MINIMUM_POWER_FREQ,
STEP_POWER_FREQ, NUMBER_OF_POWERS_OVER_FREQ, aDisturbance);

    {
        const int length = statics->stopFrameIdx - statics->startFrameIdx + 1;
        SmartBufferPolqa tempBuffer(POLQAHandle, length);
        XFLOAT *temp = tempBuffer.Buffer;

        matbPow2(aDisturbance[2].m_pData + statics->startFrameIdx, 3.0, temp, length);
        overallAvgDisturbance = matSum(temp, length);

        matbPow2(aAddedDisturbance[2].m_pData + statics->startFrameIdx, 3.0, temp,
length);
        overallAvgAddedDisturbance = matSum(temp, length);
    }
    overallAvgDisturbance /= ( numberOfSpeechFrames + 0.01);
    overallAvgAddedDisturbance /= ( numberOfSpeechFrames + 0.01);
    overallAvgDisturbance = pow(overallAvgDisturbance,0.33);
    overallAvgAddedDisturbance = pow(overallAvgAddedDisturbance,0.33);

    overallDisturbance = 0.0;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
        if (aSilent.m_pData[frameIndex])
            overallDisturbance += aDisturbance[2].m_pData[frameIndex];
    overallDisturbance /= ( numberOfSpeechFrames + 0.01);
//END POLQAMAIN PART 1s
CheckTimeMatEval(POLQAHandle->mh, 3, &ClockCycles, &TimeDiff);
AddProcessingTime(pOverviewHolder, "PESQMAIN PART 2", TimeDiff, ClockCycles);

    frameFlatnessDisturbanceAvg = 0.0;
    frameFlatnessDisturbanceAvgCompensationSilent = 0.0;
    frameFlatnessDisturbanceAvgCompensationActive = 0.0;

    numberOfActiveFrames = 0;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
        if (aSilent.m_pData[frameIndex]) {
            frameFlatnessDisturbanceAvgCompensationSilent += disturbanceDensity.
SpectralFlatness (frameIndex);
        } else {
            numberOfActiveFrames++;
            frameFlatnessDisturbanceAvgCompensationActive += disturbanceDensity.
SpectralFlatness (frameIndex);
        }
    }

    frameFlatnessDisturbanceAvgCompensationSilent /= (numberOfSilentFrames+0.1);
    frameFlatnessDisturbanceAvgCompensation000silent =
frameFlatnessDisturbanceAvgCompensationSilent;
    if (aListeningCondition==WIDE_H) {
        frameFlatnessDisturbanceAvgCompensationAddedSilent =
pow((frameFlatnessDisturbanceAvgCompensationSilent+0.8), 0.12);
        frameFlatnessDisturbanceAvgCompensationSilent =
pow((frameFlatnessDisturbanceAvgCompensationSilent+1.0),0.08);
        if (frameFlatnessDisturbanceAvgCompensation000silent > 0.55)
frameFlatnessDisturbanceAvgCompensation000silent = 0.55;
        if (frameFlatnessDisturbanceAvgCompensation000silent < 0.15)
frameFlatnessDisturbanceAvgCompensation000silent = 0.15;
        frameFlatnessDisturbanceAvgCompensation000silent /= 0.55;
        frameFlatnessDisturbanceAvgCompensation000silent =
pow((frameFlatnessDisturbanceAvgCompensation000silent),0.2)/SampleRateRatioComp
ensation3;
    } else {
        frameFlatnessDisturbanceAvgCompensationAddedSilent =
```

```
pow((frameFlatnessDisturbanceAvgCompensationSilent+0.8), 0.1);
        frameFlatnessDisturbanceAvgCompensationSilent =
pow((frameFlatnessDisturbanceAvgCompensationSilent+1.0),0.08);
        if (frameFlatnessDisturbanceAvgCompensation000silent > 0.85)
frameFlatnessDisturbanceAvgCompensation000silent = 0.85;
        if (frameFlatnessDisturbanceAvgCompensation000silent < 0.5)
frameFlatnessDisturbanceAvgCompensation000silent = 0.5;
        frameFlatnessDisturbanceAvgCompensation000silent /= 0.85;
        frameFlatnessDisturbanceAvgCompensation000silent =
1.02*pow((frameFlatnessDisturbanceAvgCompensation000silent),0.2);
    }

    frameFlatnessDisturbanceAvgCompensationActive /= (numberOfActiveFrames + 0.1);
    frameFlatnessDisturbanceAvgCompensationActiveFrq =
frameFlatnessDisturbanceAvgCompensationActive;
    frameFlatnessDisturbanceAvgCompensationActiveFrq =
0.3*(1.0+frameFlatnessDisturbanceAvgCompensationActiveFrq);
    if (frameFlatnessDisturbanceAvgCompensationActiveFrq > 0.5)
frameFlatnessDisturbanceAvgCompensationActiveFrq = 0.5;
    frameFlatnessDisturbanceAvgCompensation000active =
frameFlatnessDisturbanceAvgCompensationActive;
    if (aListeningCondition==WIDE_H) {
        frameFlatnessDisturbanceAvgCompensationAddedActive =
pow((frameFlatnessDisturbanceAvgCompensationActive+0.8),0.45);
        frameFlatnessDisturbanceAvgCompensationActive =
pow((frameFlatnessDisturbanceAvgCompensationActive+1.0),0.03);
        if (frameFlatnessDisturbanceAvgCompensation000active > 0.5)
frameFlatnessDisturbanceAvgCompensation000active = 0.5;
        if (frameFlatnessDisturbanceAvgCompensation000active < 0.2)
frameFlatnessDisturbanceAvgCompensation000active = 0.2;
        frameFlatnessDisturbanceAvgCompensation000active /= 0.5;
        frameFlatnessDisturbanceAvgCompensation000active =
1.02*pow(frameFlatnessDisturbanceAvgCompensation000active,0.2);
    } else {
        frameFlatnessDisturbanceAvgCompensationAddedActive =
pow((frameFlatnessDisturbanceAvgCompensationActive+0.8),0.3);
        frameFlatnessDisturbanceAvgCompensationActive =
pow((frameFlatnessDisturbanceAvgCompensationActive+1.0),0.08);
        if (frameFlatnessDisturbanceAvgCompensation000active > 0.85)
frameFlatnessDisturbanceAvgCompensation000active = 0.85;
        if (frameFlatnessDisturbanceAvgCompensation000active < 0.2)
frameFlatnessDisturbanceAvgCompensation000active = 0.2;
        frameFlatnessDisturbanceAvgCompensation000active =
pow((frameFlatnessDisturbanceAvgCompensation000active),0.1);
    }

    CheckTimeMatInit(POLQAHandle->mh, 3);
//POLQAMAIN PART 0 ADDED

    oldOldScale = 1.0;
    oldScale = 1.0;
    minimumOriginalFramePower = 10000000.0;
    MaxScale = 1.0;
    MinScale = 1.0;
    MinMinScale = 0.3;
    for (frameIndex = (statics->startFrameIdx); frameIndex <= (statics->stopFrameIdx);
frameIndex++)     {
        if (UseThisFrame[frameIndex]) {
            scale = (aDistortedTotalPower.m_pData[frameIndex] + (XFLOAT) 1.0e4) /
(aOriginalTotalPower.m_pData[frameIndex] + (XFLOAT) 1.0e4) ;

            if (scale > MaxScale) scale = MaxScale;
            if (scale < MinMinScale) scale = MinMinScale;
            aScale.m_pData[frameIndex] = scale;

            if (aListeningCondition==WIDE_H) {
                scale = (XFLOAT) 0.15 * oldOldScale + (XFLOAT) 0.35 * oldScale +
(XFLOAT) 0.5 * scale;
            } else {
                scale = (XFLOAT) 0.4 * oldOldScale + (XFLOAT) 0.3 * oldScale + (XFLOAT)
0.3 * scale;
            }
            oldOldScale = oldScale;
            oldScale = scale;
            if (aListeningCondition==WIDE_H) {
                originalPitchPowerDensity. MultiplyWith (frameIndex, pow(scale, 0.7));
```

```
            } else {
                originalPitchPowerDensity. MultiplyWith (frameIndex, sqrt(scale));
            }

        } else {
            originalPitchPowerDensity. MultiplyWith (frameIndex, 0.0);
            distortedPitchPowerDensity. MultiplyWith (frameIndex, 0.0);
        }
    }

    if (aListeningCondition==WIDE_H) {
        originalPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensity, aActiveFreqresponseIntell, 0.2);
        distortedPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensity, aActiveFreqresponseIntell, 0.2);
        distortedPitchPowerDensity. AudibleFreqRespCompensationOf (POLQAHandle,
distortedPitchPowerDensityMainAvg,
                                                                   originalPitchPowerDensity
MainAvg, 4.0E3, 0.6,
                                                                   statics->listeningConditi
on);
    }

    if (aListeningCondition==WIDE_H) {
        originalPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensity, aActiveFreqresponse, 0.4);
        distortedPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensity, aActiveFreqresponse, 0.4);
        originalPitchPowerDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchPowerDensityMainAvg,
                                                                   distortedPitchPowerDensit
yMainAvg, 3.0E3, 1.0,
                                                                   statics->listeningConditi
on);
    } else {
        originalPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensity, aActiveFreqresponse, 0.6);
        distortedPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensity, aActiveFreqresponse, 0.6);
        originalPitchPowerDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchPowerDensityMainAvg,
                                                                   distortedPitchPowerDensit
yMainAvg, 6.0E3, 0.8,
                                                                   statics->listeningConditi
on);
    }

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)     {
    }

    smearedOriginalPitchPowerDensity. ExcitationOf (POLQAHandle,
originalPitchPowerDensity, UseThisFrame, statics->listeningCondition);
    smearedDistortedPitchPowerDensity. ExcitationOf (POLQAHandle,
distortedPitchPowerDensity, UseThisFrame, statics->listeningCondition);

    originalLoudnessDensity. IntensityWarpingOf (POLQAHandle,
smearedOriginalPitchPowerDensity);
    distortedLoudnessDensity. IntensityWarpingOf (POLQAHandle,
smearedDistortedPitchPowerDensity);

//END POLQAMAIN PART 0 ADDED

//POLQAMAIN PART 1 ADDED

    originalPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
originalLoudnessDensity, aActiveFreqresponse, 0.6);
    distortedPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
distortedLoudnessDensity, aActiveFreqresponse, 0.6);
    originalLoudnessDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchLoudnessDensityMainAvg,
                                                                   distortedPitchLoudnessDen
sityMainAvg,
                                                                   0.2, 1.0,
                                                                   statics->listeningConditi
on);
```

```
    if (aListeningCondition==STANDARD_IRS) {

        originalPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(originalLoudnessDensity, aSilent, 5.0, numberOfSilentFrames);
        distortedPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(distortedLoudnessDensity, aSilent, 3.0, numberOfSilentFrames);

        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartlyAdded
(POLQAHandle, originalPitchLoudnessDensityMainAvg, 0.2);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2Added
(distortedPitchLoudnessDensityMainAvg,
0.2*delayVariationCompensationAdded*aDistortedSilencePowerMeanCompensation/pow(
noiseContrastMax1,0.65), 1.0);

    }
    if (aListeningCondition==WIDE_H) {

        originalPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(originalLoudnessDensity, aSilent, 4.0, numberOfSilentFrames);
        distortedPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(distortedLoudnessDensity, aSilent, 2.0, numberOfSilentFrames);

        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartlyAdded
(POLQAHandle, originalPitchLoudnessDensityMainAvg, 0.22);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2Added
(distortedPitchLoudnessDensityMainAvg,
0.6*delayVariationCompensationAdded*aDistortedSilencePowerMeanCompensation/pow(
noiseContrastMax1,0.65), 1.0);

    }
    if (aListeningCondition==NARROW_H) {

        originalPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(originalLoudnessDensity, aSilent, 5.0, numberOfSilentFrames);
        distortedPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(distortedLoudnessDensity, aSilent, 3.0, numberOfSilentFrames);

        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartlyAdded
(POLQAHandle, originalPitchLoudnessDensityMainAvg, 0.2);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2Added
(distortedPitchLoudnessDensityMainAvg,
0.3*aDistortedSilencePowerMeanCompensation, 1.0);
    }

    oldOldLoudnessScaleLow = 1.0;
    oldLoudnessScaleLow = 1.0;

    if (aListeningCondition==WIDE_H) {
        if (maxFreqBarkSource<22.0) {
            if (maxFreqBarkSource<18.0) {
                bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 1.0);
                bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(17.0);
                for (frameIndex = (statics->startFrameIdx); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
                    if (UseThisFrame[frameIndex]) {
                        distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.6,
bandIdxLow, bandIdxHigh);
                        originalLoudnessHulp = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.4,
bandIdxLow, bandIdxHigh);
                            loudnessScaleLow = (distortedLoudnessHulp +
3.0)/(originalLoudnessHulp + 3.0);
                        oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                        oldLoudnessScaleLow = loudnessScaleLow;
                        if (loudnessScaleLow>1.25) loudnessScaleLow = 1.25;
                        originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow,1.0), 0.0, 99.0);
                    } else {
                        originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
0.0, 0.0, 99.0);
                        distortedLoudnessDensity. MultiplyWithOverBandRange
(frameIndex, 0.0, 0.0, 99.0);
                    }
                }
```

```
            } else {
                bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 1.0);
                bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(17.0);
                for (frameIndex = (statics->startFrameIdx); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
                    if (UseThisFrame[frameIndex]) {
                        distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.6,
bandIdxLow, bandIdxHigh);
                        originalLoudnessHulp = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.6,
bandIdxLow, bandIdxHigh);
                        loudnessScaleLow = (distortedLoudnessHulp +
5.0)/(originalLoudnessHulp + 5.0);
                        oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                        oldLoudnessScaleLow = loudnessScaleLow;
                        if (loudnessScaleLow>1.25) loudnessScaleLow = 1.25;
                        originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow,0.85), 0.0, 99.0);
                    } else {
                        originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
0.0, 0.0, 99.0);
                        distortedLoudnessDensity. MultiplyWithOverBandRange
(frameIndex, 0.0, 0.0, 99.0);
                    }
                }
            }
        } else {
            bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
            bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
            for (frameIndex = (statics->startFrameIdx); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
                if (UseThisFrame[frameIndex]) {
                    distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.5, bandIdxLow,
bandIdxHigh);
                    originalLoudnessHulp = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.5, bandIdxLow,
bandIdxHigh);
                    loudnessScaleLow = (distortedLoudnessHulp +
4.0)/(originalLoudnessHulp + 4.0);
                    oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                    oldLoudnessScaleLow = loudnessScaleLow;
                    if (loudnessScaleLow>1.25) loudnessScaleLow = 1.25;
                    originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow,0.8), 0.0, 99.0);
                } else {
                    originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
0.0, 0.0, 99.0);
                    distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
0.0, 0.0, 99.0);
                }
            }
        }
    } else {
        bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 1.0);
        bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(17.0);
        for (frameIndex = (statics->startFrameIdx); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
            if (UseThisFrame[frameIndex]) {
                distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.4, bandIdxLow,
bandIdxHigh);
                originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, 1.4, bandIdxLow, bandIdxHigh);
                loudnessScaleLow = (distortedLoudnessHulp + 4.0)/(originalLoudnessHulp
+ 4.0);
                oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                oldLoudnessScaleLow = loudnessScaleLow;
                if (loudnessScaleLow>1.25) loudnessScaleLow = 1.25;
                originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow,0.5), 0.0, 99.0);
            } else {
                originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0,
0.0, 99.0);
                distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0,
```

```
0.0, 99.0);
            }
        }
    }

    oldOldLoudnessScaleLow = 1.0;
    oldLoudnessScaleLow = 1.0;
    delayReliabilityPerFrameWeightOld = 1.0;

    if (aListeningCondition==WIDE_H) {
        bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 3.0);
        bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(17.0);
        powFac = 0.65;
        for (frameIndex = (statics->startFrameIdx); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
            if (UseThisFrame[frameIndex]) {
                distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.1, bandIdxLow,
bandIdxHigh);
                originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, 1.1, bandIdxLow, bandIdxHigh);
                loudnessScaleLow = (originalLoudnessHulp +
8.0*noiseContrastMax1)/(distortedLoudnessHulp + 8.0*noiseContrastMax1);
                loudnessScaleLow = 0.1*oldOldLoudnessScaleLow + 0.2*oldLoudnessScaleLow
+ 0.7*loudnessScaleLow;
                oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                oldLoudnessScaleLow = loudnessScaleLow;
                if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;
                distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, powFac), 0.0, 99.0);
            } else {
                originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0,
0.0, 99.0);
                distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0,
0.0, 99.0);
            }
        }
    } else {
        bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 3.0);
        bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(17.0);
        powFac = 0.7;
        for (frameIndex = (statics->startFrameIdx); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
            if (UseThisFrame[frameIndex]) {
                distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.1, bandIdxLow,
bandIdxHigh);
                originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, 1.1, bandIdxLow, bandIdxHigh);
                loudnessScaleLow = (originalLoudnessHulp +
9.0*noiseContrastMax1)/(distortedLoudnessHulp + 9.0*noiseContrastMax1);
                loudnessScaleLow = 0.1*oldOldLoudnessScaleLow + 0.2*oldLoudnessScaleLow
+ 0.7*loudnessScaleLow;
                oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                oldLoudnessScaleLow = loudnessScaleLow;
                if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;
                distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, powFac), 0.0, 99.0);
            } else {
                originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0,
0.0, 99.0);
                distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0,
0.0, 99.0);
            }
        }
    }

    originalPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
originalLoudnessDensity, aActiveFreqresponse, 1.6);
    distortedPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
distortedLoudnessDensity, aActiveFreqresponse, 1.6);
    originalLoudnessDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchLoudnessDensityMainAvg,
                                                            distortedPitchLoudnessDen
sityMainAvg,
                                                            0.01, 1.0,
```

```
                                                                statics->listeningConditi
on);

    if (maxFreqBarkSource<18.0) {
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 17.0,
99.0);
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 18.0,
99.0);
        }
    } else {
    }

    oldLoudnessScaleLow = 1.0;

    bandLowBarkPolqaPlus = 10.0;

    if (aListeningCondition==STANDARD_IRS) bandLowBarkPolqaPlus = 7.0;

    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(bandLowBarkPolqaPlus);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){

        distortedLoudnessHulp = distortedLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 7.0, bandIdxLow, bandIdxHigh);
        originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 7.0, bandIdxLow, bandIdxHigh);

        loudnessScaleLow = (originalLoudnessHulp + 0.1)/(distortedLoudnessHulp + 0.1);
        loudnessScaleLow = 0.2*oldLoudnessScaleLow + 0.8*loudnessScaleLow;
        oldLoudnessScaleLow = loudnessScaleLow;
        distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.5), 0.0, bandLowBarkPolqaPlus);
    }

    oldLoudnessScaleLow = 1.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx (0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(5.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
        if (aListeningCondition==WIDE_H) {

            distortedLoudnessHulp = distortedLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 10.0, bandIdxLow, bandIdxHigh);
            originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 10.0, bandIdxLow, bandIdxHigh);

            loudnessScaleLow = (originalLoudnessHulp + 0.1)/(distortedLoudnessHulp +
0.1);
            loudnessScaleLow = 0.05*oldLoudnessScaleLow + 0.95*loudnessScaleLow;
            oldLoudnessScaleLow = loudnessScaleLow;
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.4), 0.0, 4.0);
        } else {
        }
    }

    aDistortedLoudnessMean = 0.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 2.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(16.0);
    if (aListeningCondition==WIDE_H) {
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 2.1, bandIdxLow,
bandIdxHigh);
            aDistortedLoudnessMean += pow(aDistortedLoudness.m_pData[frameIndex],
LpLoudnessMeanPartial);
        }
        aDistortedLoudnessMean /= (numberOfSpeechFrames + 0.5);
        aLoudnessScalingDistorted =
fixedGlobalInternalLevelAdded/(pow(aDistortedLoudnessMean,(1.0/LpLoudnessMeanPa
rtial))+0.9);
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
```

```
frameIndex++)
            distortedLoudnessDensity. MultiplyWith (frameIndex,
pow(aLoudnessScalingDistorted,0.18));
    } else {
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangePartial,
bandIdxLow, bandIdxHigh);
            aDistortedLoudnessMean += pow(aDistortedLoudness.m_pData[frameIndex],
LpLoudnessMeanPartial);
        }
        aDistortedLoudnessMean /= (numberOfSpeechFrames + 0.5);
        aLoudnessScalingDistorted =
fixedGlobalInternalLevelAdded/(pow(aDistortedLoudnessMean,(1.0/LpLoudnessMeanPa
rtial))+0.3);
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
            distortedLoudnessDensity. MultiplyWith (frameIndex,
pow(aLoudnessScalingDistorted,0.3));
    }

    aOriginalLoudnessMean = 0.0;
    aDistortedLoudnessMean = 0.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            aOriginalLoudness.m_pData[frameIndex] = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangeComplete ,
bandIdxLow, bandIdxHigh);
            aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangeComplete,
bandIdxLow, bandIdxHigh);
            aOriginalLoudnessMean += pow(aOriginalLoudness.m_pData[frameIndex],
LpLoudnessMeanComplete);
            aDistortedLoudnessMean += pow(aDistortedLoudness.m_pData[frameIndex],
LpLoudnessMeanComplete);
    }
    aOriginalLoudnessMean /= (numberOfSpeechFrames + 0.5);
    aDistortedLoudnessMean /= (numberOfSpeechFrames + 0.5);
    aLoudnessScalingOriginal  =
(pow(aDistortedLoudnessMean,(1.0/LpLoudnessMeanComplete))+0.1*pow(globalScaleDistor
tedToFixedlevel,0.1)) / (pow(aOriginalLoudnessMean,
(1.0/LpLoudnessMeanComplete))+0.1*pow(globalScaleDistortedToFixedlevel,0.1));
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
        originalLoudnessDensity. MultiplyWith (frameIndex,
pow(aLoudnessScalingOriginal, 0.3));
    }

    originalPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(originalLoudnessDensity, aSilent, 2.0, numberOfSilentFrames);
    distortedPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(distortedLoudnessDensity, aSilent, 10.0, numberOfSilentFrames);

    if (aListeningCondition==WIDE_H) {
        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartlyAdded
(POLQAHandle, originalPitchLoudnessDensityMainAvg, 0.06);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2Added
(distortedPitchLoudnessDensityMainAvg, 0.15*delayVariationCompensationAdded,
2.3*noiseContrastMax1);
    } else {
        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartlyAdded
(POLQAHandle, originalPitchLoudnessDensityMainAvg, 0.05);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2Added
(distortedPitchLoudnessDensityMainAvg, 0.15*delayVariationCompensationAdded,
2.3*noiseContrastMax1);
    }

    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
        aOriginalLoudness.m_pData[frameIndex] = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpLoudness, bandIdxLow,
```

```
bandIdxHigh);
        aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpLoudness, bandIdxLow,
bandIdxHigh);
    }

    disturbanceDensityAsymAdd. DifferenceOf (distortedLoudnessDensity,
originalLoudnessDensity);
    mask. MinimumOf (distortedLoudnessDensity, originalLoudnessDensity);

    mask *= (XFLOAT) 0.85;

    disturbanceDensityAsymAdd. MaskWith (POLQAHandle, mask);
    disturbanceDensityAsymAdd. MultiplyWithAsymmetryFactorAddOf
(originalPitchPowerDensity, distortedPitchPowerDensity,
statics->listeningCondition, noiseContrastMax1, aSuperSilent,
aDistortedSilencePowerMeanCompensation);

    disturbanceDensityAsymAdd. ComputeLpWeights (POLQAHandle, MINIMUM_POWER_FREQ,
STEP_POWER_FREQ, NUMBER_OF_POWERS_OVER_FREQ, aAddedDisturbance);

//END POLQAMAIN PART 1 ADDED

    CheckTimeMatEval(POLQAHandle->mh, 3, &ClockCycles, &TimeDiff);
    AddProcessingTime(pOverviewHolder, "PESQMAIN PART 2 ADDED", TimeDiff, ClockCycles);

    count = 0;
XFLOAT  sumXY = 0.0;
XFLOAT  sumX = 0.0;
XFLOAT  sumY = 0.0;
XFLOAT  sumX2 = 0.0;
XFLOAT  sumY2 = 0.0;
XFLOAT  correlationOriginalWithDisturbance;
    for (frameIndex = (statics->startFrameIdx); frameIndex <= (statics->stopFrameIdx);
frameIndex++)     {
        if (aActiveFreqresponse.m_pData[frameIndex]) {
            XFLOAT X = aOriginalLoudness.m_pData[frameIndex];
            XFLOAT Y = aDisturbance[2].m_pData[frameIndex];
                sumXY += X*Y;
                sumX += X;
                sumY += Y;
                sumX2 += X*X;
                sumY2 += Y*Y;
                count++;
        }
    }
    if ( count>2 && sumX>0.0 && sumY>0.0 ) {
        correlationOriginalWithDisturbance =
(count*sumXY-sumX*sumY)/sqrt((count*sumX2-sumX*sumX)*(count*sumY2-sumY*sumY));
    } else {
        correlationOriginalWithDisturbance = 0.0;
    }
    correlationOriginalWithDisturbanceCompensation000ForReverb =
correlationOriginalWithDisturbance;
    if (correlationOriginalWithDisturbanceCompensation000ForReverb<0.5)
correlationOriginalWithDisturbanceCompensation000ForReverb = 0.5;
    correlationOriginalWithDisturbanceCompensation000ForReverb *=
SampleRateRatioCompensation2;
    correlationOriginalWithDisturbanceCompensation000ForMNRU =
(correlationOriginalWithDisturbance*SampleRateRatioCompensation2);
    if (correlationOriginalWithDisturbanceCompensation000ForMNRU<0.0)
correlationOriginalWithDisturbanceCompensation000ForMNRU = 0.0;

    if (correlationOriginalWithDisturbanceCompensation000ForMNRU>0.6)
correlationOriginalWithDisturbanceCompensation000ForMNRU = 0.6;

    correlationOriginalWithDisturbanceCompensation000ForMNRU /= 17.0;

    if (correlationOriginalWithDisturbance<0.3) correlationOriginalWithDisturbance =
0.3;

CheckTimeMatEval(POLQAHandle->mh, 3, &ClockCycles, &TimeDiff);
//START BAD FRAMES PROCESING FOR BIG
DISTORTIONS*****************************************************************
*
    CIntArray   frameIsBad;
```

```
    CIntArray   smearedFrameIsBad;
    CIntArray   startFrameBadInterval;
    CIntArray   stopFrameBadInterval;
    CIntArray   numberOfFramesInBadInterval;
    CIntArray   startSampleBadInterval;
    CIntArray   stopSampleBadInterval;
    CIntArray   numberOfSamplesInBadInterval;
    CIntArray   delayInSamplesInBadInterval;
    int         numberOfBadIntervals = 0;

    if (aListeningCondition==WIDE_H) {
        if (globalScaleCorrectionIntellLevelCorrection<1.0) {
            THRESHOLD_BAD_FRAMES = 42.0;
        } else {
            THRESHOLD_BAD_FRAMES = 42.0;
        }
    } else {
        if (globalScaleCorrectionIntellLevelCorrection<1.0) {
            THRESHOLD_BAD_FRAMES = 45.0;
        } else {
            THRESHOLD_BAD_FRAMES =
45.0/pow(globalScaleCorrectionIntellLevelCorrection,0.1);
        }
    }

    frameIsBad. SetSize (statics->nrFrames);
    smearedFrameIsBad. SetSize (statics->nrFrames);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
    {
        frameIsBad.m_pData[frameIndex] = (aDisturbance[2].m_pData[frameIndex] >
THRESHOLD_BAD_FRAMES);

        smearedFrameIsBad.m_pData[frameIndex] = FALSE;
    }
    frameIsBad.m_pData[0] = FALSE;

    for (frameIndex = 2; frameIndex < statics->nrFrames - 2; frameIndex++)
    {
        BOOL maximumItselfAndLeft = frameIsBad.m_pData[frameIndex];

        for (i = -2; i <= 0; i++)
        {
            if (maximumItselfAndLeft < frameIsBad.m_pData[frameIndex  + i])
            {
                maximumItselfAndLeft = frameIsBad.m_pData[frameIndex  + i];
            }
        }

        BOOL maximumItselfAndRight = frameIsBad.m_pData[frameIndex];

        for (i = 0; i <= 2; i++)
        {
            if (maximumItselfAndRight < frameIsBad.m_pData[frameIndex + i])
            {
                maximumItselfAndRight = frameIsBad.m_pData[frameIndex + i];
            }
        }

        BOOL mini = maximumItselfAndLeft;
        if (mini > maximumItselfAndRight)
        {
            mini = maximumItselfAndRight;
        }

        smearedFrameIsBad.m_pData[frameIndex] = (short) mini;

    }

int MINIMUM_NUMBER_OF_BAD_FRAMES_IN_BAD_INTERVAL;

if (aListeningCondition==WIDE_H) {
    MINIMUM_NUMBER_OF_BAD_FRAMES_IN_BAD_INTERVAL = 5;
} else {
    MINIMUM_NUMBER_OF_BAD_FRAMES_IN_BAD_INTERVAL = 2;
}
```

```
    numberOfBadIntervals = 0;
    frameIndex = 0;
    while(frameIndex < statics->nrFrames)
    {
        while((frameIndex < statics->nrFrames) &&
(!smearedFrameIsBad.m_pData[frameIndex]))
        {
            frameIndex++;
        }
        if (frameIndex < statics->nrFrames)
        {
            startFrameBadInterval. SetAtGrow (numberOfBadIntervals, frameIndex);

            while ((frameIndex < statics->nrFrames) &&
(smearedFrameIsBad.m_pData[frameIndex]))
            {
                frameIndex++;
            }

            if (frameIndex < statics->nrFrames)
            {
                stopFrameBadInterval. SetAtGrow (numberOfBadIntervals, frameIndex);
                if (stopFrameBadInterval.m_pData[numberOfBadIntervals] -
startFrameBadInterval.m_pData[numberOfBadIntervals] >=
MINIMUM_NUMBER_OF_BAD_FRAMES_IN_BAD_INTERVAL)
                {
                    numberOfBadIntervals++;
                }
            }
        }
    }

    for(int badIntervalIndex = 0; badIntervalIndex < numberOfBadIntervals;
badIntervalIndex++)
    {
        startSampleBadInterval. SetAtGrow (badIntervalIndex,
FrameToSample(startFrameBadInterval.m_pData[badIntervalIndex]) );
        stopSampleBadInterval. SetAtGrow (badIntervalIndex,
stopFrameBadInterval.m_pData[badIntervalIndex] * aTransformLength / 2 +
aTransformLength);
        if (stopSampleBadInterval.m_pData[badIntervalIndex] > statics->nrTimesSamples)
        {
            stopSampleBadInterval.m_pData[badIntervalIndex] = statics->nrTimesSamples;
        }

        numberOfSamplesInBadInterval. SetAtGrow (badIntervalIndex,
stopSampleBadInterval.m_pData[badIntervalIndex] -
startSampleBadInterval.m_pData[badIntervalIndex]);

    }

    CheckTimeMatInit(POLQAHandle->mh, 3);
//POLQAMAIN PART 0repeat FOR BIG DISTORTIONS

    distortedHzPowerSpectrum. STFTPowerSpectrumOf (POLQAHandle, aDistortedTimeSeries,
aStartSampleUtterance, aStopSampleUtterance, aDelayUtterance, true, true);

    CheckTimeMatInit(POLQAHandle->mh, 3);
    ShowProgress (10, "Spectrum correction");
    distortedHzPowerSpectrumCorrected = new CHzSpectrum();
    distortedHzPowerSpectrumCorrected->Initialize("distortedHzPowerSpectrumCorrected");

    {
        int    *dummySpecShift = (int*)matMalloc((statics->stopFrameIdx + 1) *
sizeof(int));

        SmartBufferPolqa SB_dummyWarpingFacs(POLQAHandle, statics->stopFrameIdx + 1);
        XFLOAT *dummyWarpingFacs = SB_dummyWarpingFacs.Buffer;

        ShiftPitch(&originalHzPowerSpectrum, &distortedHzPowerSpectrum,
distortedHzPowerSpectrumCorrected, pActiveFrameFlags,
                   dummySpecShift, dummyWarpingFacs);

        matFree(dummySpecShift);
    }
```

```
    for(int fr = CPsqmArray::GetStartFrameIndex(); fr <= statics->stopFrameIdx; fr++)
        matbCopy(distortedHzPowerSpectrumCorrected->m_pData[fr],
distortedHzPowerSpectrum.m_pData[fr], statics->aNumberOfHzBands);
    delete distortedHzPowerSpectrumCorrected;
    distortedHzPowerSpectrumCorrected = NULL;

    CheckTimeMatEval(POLQAHandle->mh, 3, &ClockCycles, &TimeDiff);
    AddProcessingTime(pOverviewHolder, "Spectrum correction", TimeDiff, ClockCycles);

    distortedPitchPowerDensity. FrequencyWarpingOf (POLQAHandle,
distortedHzPowerSpectrum, PitchRatio);
    originalPitchPowerDensity. FrequencyWarpingOf (POLQAHandle,
originalHzPowerSpectrum, 1.0);

    oldOldScale = 1.0;
    oldScale = 1.0;
    minimumOriginalFramePower = 10000000.0;
    MaxScale = 1.0;
    MinScale = 1.0;
    MinMinScale = 0.25;
    for (frameIndex = (statics->startFrameIdx); frameIndex <= (statics->stopFrameIdx);
frameIndex++)     {
        if (UseThisFrame[frameIndex]) {
            aOriginalTotalPower.m_pData[frameIndex] = originalPitchPowerDensity. Total
(frameIndex, 0.0, 3.0e4);
            aDistortedTotalPower.m_pData[frameIndex] = distortedPitchPowerDensity.
Total (frameIndex, 0.0, 3.0e4);
            if (aOriginalTotalPower.m_pData[frameIndex] < minimumOriginalFramePower)
{minimumOriginalFramePower = aOriginalTotalPower.m_pData[frameIndex];}
            scale = (aDistortedTotalPower.m_pData[frameIndex] + (XFLOAT) 3.0e4) /
(aOriginalTotalPower.m_pData[frameIndex] + (XFLOAT) 3.0e4) ;

            if (scale > MaxScale) scale = MaxScale;
            if (scale < MinMinScale) scale = MinMinScale;

            aScale.m_pData[frameIndex] = scale;

            if (aListeningCondition==WIDE_H) {
                scale = (XFLOAT) 0.4 * oldOldScale + (XFLOAT) 0.3 * oldScale + (XFLOAT)
0.3 * scale;
            } else {
                scale = (XFLOAT) 0.4 * oldOldScale + (XFLOAT) 0.3 * oldScale + (XFLOAT)
0.3 * scale;
            }
            oldOldScale = oldScale;
            oldScale = scale;
            if (aListeningCondition==WIDE_H) {
                originalPitchPowerDensity. MultiplyWith (frameIndex, pow(scale,
0.65*globalScaleDistortedToFixedlevelHulp));
            } else {
                originalPitchPowerDensity. MultiplyWith (frameIndex, pow(scale,
0.6*globalScaleDistortedToFixedlevelHulp));
            }

        } else {
            originalPitchPowerDensity. MultiplyWith (frameIndex, 0.0);
            distortedPitchPowerDensity. MultiplyWith (frameIndex, 0.0);
        }
    }

    if (aListeningCondition==STANDARD_IRS) {
        originalPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensity, aActiveFreqresponse, 0.7);
        distortedPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensity, aActiveFreqresponse, 0.7);
        originalPitchPowerDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchPowerDensityMainAvg,
                                                                  distortedPitchPowerDensit
yMainAvg, 1.0E5, 0.6,
                                                                  statics->listeningConditi
on);
    }
    if (aListeningCondition==WIDE_H) {

        originalPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensity, aActiveFreqresponse, 0.4);
```

```
        distortedPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensity, aActiveFreqresponse, 0.4);
        originalPitchPowerDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchPowerDensityMainAvg,
                                                                  distortedPitchPowerDensit
yMainAvg, 1.0E4,
frameFlatnessDisturbanceA
vgCompensationActiveFrq,
                                                                  statics->listeningConditi
on);

    }
    if (aListeningCondition==NARROW_H) {
        originalPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensity, aActiveFreqresponse, 0.5);
        distortedPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensity, aActiveFreqresponse, 0.5);
        originalPitchPowerDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchPowerDensityMainAvg,
                                                                  distortedPitchPowerDensit
yMainAvg, 1.0E5, 0.6,
                                                                  statics->listeningConditi
on);
    }

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)     {
    }

    smearedOriginalPitchPowerDensity. ExcitationOf (POLQAHandle,
originalPitchPowerDensity, UseThisFrame, statics->listeningCondition);
    smearedDistortedPitchPowerDensity. ExcitationOf (POLQAHandle,
distortedPitchPowerDensity, UseThisFrame, statics->listeningCondition);

    originalLoudnessDensity. IntensityWarpingOf (POLQAHandle,
smearedOriginalPitchPowerDensity);
    distortedLoudnessDensity. IntensityWarpingOf (POLQAHandle,
smearedDistortedPitchPowerDensity);
//END POLQAMAIN PART 0repeat FOR BIG DISTORTIONS

//TIMBRE Indicators
    distortedLoudnessTimbreWide   = 0.0;
    distortedLoudnessTimbreLow   = 0.0;
    distortedLoudnessTimbreHigh  = 0.0;
    count0 = 0;

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {

        if (!aSilent.m_pData[frameIndex])
        {
            count0++;
            distortedLoudnessTimbreWide += distortedLoudnessDensity.IntegralHigh2
(frameIndex, statics->listeningCondition);
            distortedLoudnessTimbreLow += distortedLoudnessDensity.IntegralLow2
(frameIndex, statics->listeningCondition);
            distortedLoudnessTimbreHigh += distortedLoudnessDensity.IntegralHigh2
(frameIndex, statics->listeningCondition);
        }
    }
    distortedLoudnessTimbreHigh  =
(distortedLoudnessTimbreLow-distortedLoudnessTimbreHigh)/(count0+1.0);

    distortedLoudnessTimbreWide   = 0.0;
    distortedLoudnessTimbreLow   = 0.0;
    distortedLoudnessTimbreHigh2  = 0.0;

    int numActFrames=0;

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {

        if (!aSilent.m_pData[frameIndex]) {
            distortedLoudnessTimbreWide += distortedLoudnessDensity.IntegralHigh2
(frameIndex, statics->listeningCondition);
            distortedLoudnessTimbreLow += distortedLoudnessDensity.IntegralLow2
```

```
(frameIndex, statics->listeningCondition);
            distortedLoudnessTimbreHigh2 += distortedLoudnessDensity.IntegralHigh2
(frameIndex, statics->listeningCondition);

            numActFrames++;

        }
    }
    distortedLoudnessTimbreHigh2  =
(distortedLoudnessTimbreLow-distortedLoudnessTimbreHigh2)/(count0+1.0);

    distortedLoudnessTimbreLow /= (XFLOAT)numActFrames;

    if (distortedLoudnessTimbreHigh<-5.0) {
        distortedLoudnessTimbreHigh = -5.0 - distortedLoudnessTimbreHigh;
    } else {
        if (distortedLoudnessTimbreHigh>15.0) {
            distortedLoudnessTimbreHigh = distortedLoudnessTimbreHigh - 15.0;
        } else {
            distortedLoudnessTimbreHigh = 0.0;
        }
    }

    distortedLoudnessTimbreLowSilent   = 0.0;
    distortedLoudnessTimbreHighSilent  = 0.0;
    count0 = 0;

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {

        if ( (aSilent.m_pData[frameIndex]) && (aDistortedLoudness.m_pData[frameIndex] >
1.0) ) {
            count0++;
            distortedLoudnessTimbreLowSilent += distortedLoudnessDensity.IntegralLow2
(frameIndex, statics->listeningCondition);
            distortedLoudnessTimbreHighSilent += distortedLoudnessDensity.IntegralHigh2
(frameIndex, statics->listeningCondition);
        }
    }
    distortedLoudnessTimbreHighSilent  =
(distortedLoudnessTimbreLowSilent-distortedLoudnessTimbreHighSilent)/(count0+1.0);
    if (distortedLoudnessTimbreHighSilent<-10.0) {
        distortedLoudnessTimbreHighSilent = -10.0 - distortedLoudnessTimbreHighSilent;
    } else {
        distortedLoudnessTimbreHighSilent =  0.0;
    }

//END TIMBRE Indicators

    CheckTimeMatInit(POLQAHandle->mh, 3);
//POLQAMAIN PART 1repeat FOR BIG DISTORTIONS

    aDistortedLoudnessMeanIndicator1 = 0.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {

        aOriginalLoudness.m_pData[frameIndex] = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle,  frameIndex, 2.0, bandIdxLow,
bandIdxHigh);
        aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle,  frameIndex, 2.0, bandIdxLow,
bandIdxHigh);

        aDistortedLoudnessMeanIndicator1 +=
pow(aDistortedLoudness.m_pData[frameIndex],0.3);
    }
    aDistortedLoudnessMeanIndicator1 /= ( numberOfSpeechFrames + 0.01);
    aDistortedLoudnessMeanIndicator1 = pow(aDistortedLoudnessMeanIndicator1,(1.0/0.3));

    originalPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(originalLoudnessDensity, aSuperSilent, 4.0, numberOfSuperSilentFrames);
    distortedPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(distortedLoudnessDensity, aSuperSilent, 4.0, numberOfSuperSilentFrames);
```

```
    if (aListeningCondition==STANDARD_IRS) {
        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg, 0.4);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2
(distortedPitchLoudnessDensityMainAvg, 0.5, 0.6);
    }
    if (aListeningCondition==WIDE_H) {

        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg, 0.11);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2
(distortedPitchLoudnessDensityMainAvg, 0.3, 0.65);

    }
    if (aListeningCondition==NARROW_H) {
        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg, 0.4);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2
(distortedPitchLoudnessDensityMainAvg, 0.4, 0.6);
    }

    oldOldLoudnessScaleLow = 1.0;
    oldLoudnessScaleLow = 1.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 2.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = (statics->startFrameIdx); frameIndex <= (statics->stopFrameIdx);
frameIndex++) {
        if (UseThisFrame[frameIndex]) {
            distortedLoudnessHulp = distortedLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, LpBandRangeLocal, bandIdxLow, bandIdxHigh);
            originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, LpBandRangeLocal, bandIdxLow, bandIdxHigh);
            loudnessScaleLow = (distortedLoudnessHulp + 10.0)/(originalLoudnessHulp +
10.0);
            if (aListeningCondition==WIDE_H) {
                loudnessScaleLow = 0.02*oldOldLoudnessScaleLow +
0.13*oldLoudnessScaleLow + 0.85*loudnessScaleLow;
                oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                oldLoudnessScaleLow = loudnessScaleLow;
                if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;
                originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.7), 0.0, 99.0);
            } else {
                loudnessScaleLow = 0.02*oldOldLoudnessScaleLow +
0.13*oldLoudnessScaleLow + 0.85*loudnessScaleLow;
                oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                oldLoudnessScaleLow = loudnessScaleLow;
                if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;
                originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.65), 0.0, 99.0);
            }
        } else {
            originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 0.0,
99.0);
            distortedLoudnessDensity.MultiplyWithOverBandRange (frameIndex, 0.0, 0.0,
99.0);
        }
    }

    oldOldLoudnessScaleLow = 1.0;
    oldLoudnessScaleLow = 1.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 2.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(18.0);
    for (frameIndex = (statics->startFrameIdx); frameIndex <= (statics->stopFrameIdx);
frameIndex++) {
        if (UseThisFrame[frameIndex]) {
            if (aListeningCondition==WIDE_H) {
                distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangeLocal,
bandIdxLow, bandIdxHigh);
                originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, LpBandRangeLocal, bandIdxLow, bandIdxHigh);
            } else {
                distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.0, bandIdxLow,
bandIdxHigh);
```

```
                originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, 1.0, bandIdxLow, bandIdxHigh);
            }
            hulp = originalLoudnessHulp;
            if (hulp>10.0) hulp = 10.0;
            loudnessScaleLow = (originalLoudnessHulp + 6.0 +
hulp)/(distortedLoudnessHulp + 6.0 + hulp);
            oldOldLoudnessScaleLow = oldLoudnessScaleLow;
            oldLoudnessScaleLow = loudnessScaleLow;
            if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;

            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.6), 0.0, 99.0);
        } else {
            originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 0.0,
99.0);
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 0.0,
99.0);
        }
    }

    if (aListeningCondition==STANDARD_IRS) {

        originalPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
originalLoudnessDensity, aActiveFreqresponse, 2.0);
        distortedPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
distortedLoudnessDensity, aActiveFreqresponse, 2.0);
        originalLoudnessDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchLoudnessDensityMainAvg,
                                                                distortedPitchLoudnessDen
sityMainAvg, 20.0, 0.5,
                                                                statics->listeningConditi
on);

    }
    if (aListeningCondition==WIDE_H) {

        originalPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
originalLoudnessDensity, aActiveFreqresponse, 0.6);
        distortedPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
distortedLoudnessDensity, aActiveFreqresponse, 0.6);
        originalLoudnessDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchLoudnessDensityMainAvg,
                                                                distortedPitchLoudnessDen
sityMainAvg, 10.0, 0.7,
                                                                statics->listeningConditi
on);

    }

    if (aListeningCondition==NARROW_H) {
        originalPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
originalLoudnessDensity, aActiveFreqresponse, 3.0);
        distortedPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
distortedLoudnessDensity, aActiveFreqresponse, 3.0);
        originalLoudnessDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchLoudnessDensityMainAvg,
                                                                distortedPitchLoudnessDen
sityMainAvg, 20.0, 0.5,
                                                                statics->listeningConditi
on);
    }

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
        originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 21.5,
99.0);
        distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 22.0,
99.0);
    }

    oldLoudnessScaleLow = 1.0;

    if (aListeningCondition==WIDE_H) {
        bandIdxLow  = originalLoudnessDensity.GetBandLowIdx (0.0);
        bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(5.0);
```

```
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {

            distortedLoudnessHulp = distortedLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 10.0, bandIdxLow, bandIdxHigh);
            originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 10.0, bandIdxLow, bandIdxHigh);

            loudnessScaleLow = (originalLoudnessHulp + 0.1)/(distortedLoudnessHulp +
0.1);
            loudnessScaleLow = 0.05*oldLoudnessScaleLow + 0.95*loudnessScaleLow;
            oldLoudnessScaleLow = loudnessScaleLow;
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.15), 0.0, 4.0);
        }
    } else {
        bandIdxLow  = originalLoudnessDensity.GetBandLowIdx (0.0);
        bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(6.0);
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {

            distortedLoudnessHulp = distortedLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 10.0, bandIdxLow, bandIdxHigh);
            originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 10.0, bandIdxLow, bandIdxHigh);

            loudnessScaleLow = (originalLoudnessHulp + 0.1)/(distortedLoudnessHulp +
0.1);
            loudnessScaleLow = 0.05*oldLoudnessScaleLow + 0.95*loudnessScaleLow;
            oldLoudnessScaleLow = loudnessScaleLow;
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.2), 0.0, 5.0);
        }
    }

    aDistortedLoudnessMean = 0.0;

    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(16.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangePartial,
bandIdxLow, bandIdxHigh);
        aDistortedLoudnessMean += pow(aDistortedLoudness.m_pData[frameIndex],
LpLoudnessMeanPartial);
    }
    aDistortedLoudnessMean /= (numberOfSpeechFrames + 0.5);
    aLoudnessScalingDistorted =
fixedGlobalInternalLevel/(pow(aDistortedLoudnessMean,(1.0/LpLoudnessMeanPartial))+0
.9);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
        distortedLoudnessDensity. MultiplyWith (frameIndex,
pow(aLoudnessScalingDistorted,0.4));

    aOriginalLoudnessMean = 0.0;
    aDistortedLoudnessMean = 0.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        aOriginalLoudness.m_pData[frameIndex] = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangeComplete ,
bandIdxLow, bandIdxHigh);
        aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangeComplete,
bandIdxLow, bandIdxHigh);
        aOriginalLoudnessMean += pow(aOriginalLoudness.m_pData[frameIndex],
LpLoudnessMeanComplete);
        aDistortedLoudnessMean += pow(aDistortedLoudness.m_pData[frameIndex],
LpLoudnessMeanComplete);
    }
    aOriginalLoudnessMean /= (numberOfSpeechFrames + 0.5);
    aDistortedLoudnessMean /= (numberOfSpeechFrames + 0.5);
```

```
    aLoudnessScalingOriginal  =
(pow(aDistortedLoudnessMean,(1.0/LpLoudnessMeanComplete))+3.8) /
(pow(aOriginalLoudnessMean, (1.0/LpLoudnessMeanComplete))+3.8);

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        originalLoudnessDensity. MultiplyWith (frameIndex, aLoudnessScalingOriginal);
    }

    originalPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(originalLoudnessDensity, aSuperSilent, 1.0, numberOfSuperSilentFrames);
    distortedPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(distortedLoudnessDensity, aSuperSilent, 1.0, numberOfSuperSilentFrames);

    if (aListeningCondition==STANDARD_IRS) {
        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg, 1.2/noiseContrastMax1);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2
(distortedPitchLoudnessDensityMainAvg, 1.2/noiseContrastMax1, 0.3);
    }
    if (aListeningCondition==WIDE_H) {

        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg, 0.8/noiseContrastMax1);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2
(distortedPitchLoudnessDensityMainAvg, 1.5/noiseContrastMax1, 0.1);

    }
    if (aListeningCondition==NARROW_H) {
        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartly (POLQAHandle,
originalPitchLoudnessDensityMainAvg, 1.2/noiseContrastMax1);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2
(distortedPitchLoudnessDensityMainAvg, 1.2/noiseContrastMax1, 0.3);
    }

    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        aOriginalLoudness.m_pData[frameIndex] = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpLoudness, bandIdxLow,
bandIdxHigh);
        aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpLoudness, bandIdxLow,
bandIdxHigh);
    }

    disturbanceDensity. DifferenceOf (distortedLoudnessDensity,
originalLoudnessDensity);

    mask. MinimumOf (distortedLoudnessDensity, originalLoudnessDensity);

    mask *= (XFLOAT) 0.25;

    disturbanceDensity. MaskWith (POLQAHandle, mask);

    disturbanceDensity. ComputeLpWeights (POLQAHandle, MINIMUM_POWER_FREQ,
STEP_POWER_FREQ, NUMBER_OF_POWERS_OVER_FREQ, aDoubleTweakedDisturbance);

//END POLQAMAIN PART 1repeat FOR BIG DISTORTIONS

//POLQAMAIN PART 0 ADDEDrepeat FOR BIG DISTORTIONS

    oldOldScale = 1.0;
    oldScale = 1.0;
    minimumOriginalFramePower = 10000000.0;
    MaxScale = 1.0;
    MinScale = 1.0;

    MinMinScale = 0.4;
    for (frameIndex = (statics->startFrameIdx); frameIndex <= (statics->stopFrameIdx);
frameIndex++)     {
        if (UseThisFrame[frameIndex]) {
            scale = (aDistortedTotalPower.m_pData[frameIndex] + (XFLOAT) 3.0e4) /
(aOriginalTotalPower.m_pData[frameIndex] + (XFLOAT) 3.0e4) ;
```

```
            if (scale > MaxScale) scale = MaxScale;
            if (scale < MinMinScale) scale = MinMinScale;
            aScale.m_pData[frameIndex] = scale;
            if (aListeningCondition==WIDE_H) {
                scale = (XFLOAT) 0.2 * oldOldScale + (XFLOAT) 0.3 * oldScale + (XFLOAT)
0.5 * scale;
            } else {
                scale = (XFLOAT) 0.3 * oldOldScale + (XFLOAT) 0.3 * oldScale + (XFLOAT)
0.4 * scale;
            }
            oldOldScale = oldScale;
            oldScale = scale;
            if (aListeningCondition==WIDE_H) {
                originalPitchPowerDensity. MultiplyWith (frameIndex, pow(scale, 0.4));
            } else {
                originalPitchPowerDensity. MultiplyWith (frameIndex, pow(scale, 0.55));
            }

        } else {
            originalPitchPowerDensity. MultiplyWith (frameIndex, 0.0);
            distortedPitchPowerDensity. MultiplyWith (frameIndex, 0.0);
        }
    }

    if (aListeningCondition==WIDE_H) {
        originalPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensity, aActiveFreqresponse, 0.7);
        distortedPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensity, aActiveFreqresponse, 0.7);
        originalPitchPowerDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchPowerDensityMainAvg,
                                                                  distortedPitchPowerDensit
yMainAvg, 1.0E4, 0.8,
                                                                  statics->listeningConditi
on);
    } else {
        originalPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
originalPitchPowerDensity, aActiveFreqresponse, 0.5);
        distortedPitchPowerDensityMainAvg. TimeLpAudibleOf (POLQAHandle,
distortedPitchPowerDensity, aActiveFreqresponse, 0.5);
        originalPitchPowerDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchPowerDensityMainAvg,
                                                                  distortedPitchPowerDensit
yMainAvg, 1.0E4, 0.8,
                                                                  statics->listeningConditi
on);
    }

    smearedOriginalPitchPowerDensity. ExcitationOf (POLQAHandle,
originalPitchPowerDensity, UseThisFrame, statics->listeningCondition);
    smearedDistortedPitchPowerDensity. ExcitationOf (POLQAHandle,
distortedPitchPowerDensity, UseThisFrame, statics->listeningCondition);

    originalLoudnessDensity. IntensityWarpingOf (POLQAHandle,
smearedOriginalPitchPowerDensity);
    distortedLoudnessDensity. IntensityWarpingOf (POLQAHandle,
smearedDistortedPitchPowerDensity);

//END POLQAMAIN PART 0 ADDEDrepeat FOR BIG DISTORTIONS

//POLQAMAIN PART 1 ADDEDrepeat FOR BIG DISTORTIONS

    oldOldLoudnessScaleLow = 1.0;
    oldLoudnessScaleLow = 1.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = (statics->startFrameIdx); frameIndex <= (statics->stopFrameIdx);
frameIndex++) {
        if (UseThisFrame[frameIndex]) {
            distortedLoudnessHulp = distortedLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, LpBandRangeLocal, bandIdxLow, bandIdxHigh);
            originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, LpBandRangeLocal, bandIdxLow, bandIdxHigh);
            loudnessScaleLow = (originalLoudnessHulp +
20.0*noiseContrastMax1)/(distortedLoudnessHulp + 20.0*noiseContrastMax1);
            oldOldLoudnessScaleLow = oldLoudnessScaleLow;
```

```
            oldLoudnessScaleLow = loudnessScaleLow;
            if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow((XFLOAT)loudnessScaleLow, (XFLOAT)0.1), 0.0, 99.0);
        } else {
            originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 0.0,
99.0);
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 0.0,
99.0);
        }
    }

    originalPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
originalLoudnessDensity, aActiveFreqresponse, 0.7);
    distortedPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
distortedLoudnessDensity, aActiveFreqresponse, 0.7);
    originalLoudnessDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchLoudnessDensityMainAvg,
                                                            distortedPitchLoudnessDen
sityMainAvg,
                                                            0.2, 1.0,
                                                            statics->listeningConditi
on);

    if (aListeningCondition==STANDARD_IRS) {

        originalPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(originalLoudnessDensity, aSilent, 5.0, numberOfSilentFrames);
        distortedPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(distortedLoudnessDensity, aSilent, 4.0, numberOfSilentFrames);

        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartlyAdded
(POLQAHandle, originalPitchLoudnessDensityMainAvg, 0.2);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2Added
(distortedPitchLoudnessDensityMainAvg,
0.3*aDistortedSilencePowerMeanCompensation, 1.0);
    }
    if (aListeningCondition==WIDE_H) {

        originalPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(originalLoudnessDensity, aSilent, 3.6, numberOfSilentFrames);
        distortedPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(distortedLoudnessDensity, aSilent, 1.3, numberOfSilentFrames);
        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartlyAdded
(POLQAHandle, originalPitchLoudnessDensityMainAvg, 0.27);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2Added
(distortedPitchLoudnessDensityMainAvg,
0.7*aDistortedSilencePowerMeanCompensation/pow(noiseContrastMax1,0.7), 1.2);

    }
    if (aListeningCondition==NARROW_H) {

        originalPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(originalLoudnessDensity, aSilent, 5.0, numberOfSilentFrames);
        distortedPitchLoudnessDensityMainAvg. TimeLpAudibleOfSilent
(distortedLoudnessDensity, aSilent, 3.0, numberOfSilentFrames);

        originalLoudnessDensity. AudibleNoiseRespCompensationOfPartlyAdded
(POLQAHandle, originalPitchLoudnessDensityMainAvg, 0.2);
        distortedLoudnessDensity. AudibleNoiseRespCompensationOfPartly2Added
(distortedPitchLoudnessDensityMainAvg,
0.3*aDistortedSilencePowerMeanCompensation, 1.0);
    }

    oldOldLoudnessScaleLow = 1.0;
    oldLoudnessScaleLow = 1.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = (statics->startFrameIdx); frameIndex <= (statics->stopFrameIdx);
frameIndex++) {
        if (UseThisFrame[frameIndex]) {

            if (aListeningCondition==WIDE_H) {
                distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.0, bandIdxLow,
bandIdxHigh);
```

```
                originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, 1.0, bandIdxLow, bandIdxHigh);
                loudnessScaleLow = (distortedLoudnessHulp + 5.0)/(originalLoudnessHulp
+ 5.0);
                oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                oldLoudnessScaleLow = loudnessScaleLow;
                if (loudnessScaleLow>1.3) loudnessScaleLow = 1.3;
                originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow((XFLOAT)loudnessScaleLow, (XFLOAT)0.5/SampleRateRatioCompensation),
0.0, 99.0);
            } else {
                distortedLoudnessHulp = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, 1.15, bandIdxLow,
bandIdxHigh);
                originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, 1.15, bandIdxLow, bandIdxHigh);
                loudnessScaleLow = (distortedLoudnessHulp + 5.0)/(originalLoudnessHulp
+ 5.0);
                oldOldLoudnessScaleLow = oldLoudnessScaleLow;
                oldLoudnessScaleLow = loudnessScaleLow;
                if (loudnessScaleLow>1.3) loudnessScaleLow = 1.3;
                originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow((XFLOAT)loudnessScaleLow, (XFLOAT)0.5), 0.0, 99.0);
            }

        } else {
            originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 0.0,
99.0);
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 0.0,
99.0);
        }
    }

    oldOldLoudnessScaleLow = 1.0;
    oldLoudnessScaleLow = 1.0;

        bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
        bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
        powFac = 0.8;
    for (frameIndex = (statics->startFrameIdx); frameIndex <= (statics->stopFrameIdx);
frameIndex++) {
        if (UseThisFrame[frameIndex]) {
            distortedLoudnessHulp = distortedLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, LpBandRangeLocal, bandIdxLow, bandIdxHigh);
            originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle, frameIndex, LpBandRangeLocal, bandIdxLow, bandIdxHigh);
            loudnessScaleLow = (originalLoudnessHulp +
5.0*noiseContrastMax1)/(distortedLoudnessHulp + 5.0*noiseContrastMax1);
            oldOldLoudnessScaleLow = oldLoudnessScaleLow;
            oldLoudnessScaleLow = loudnessScaleLow;
            if (loudnessScaleLow>1.0) loudnessScaleLow = 1.0;
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, powFac), 0.0, 99.0);
        } else {
            originalLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 0.0,
99.0);
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex, 0.0, 0.0,
99.0);
        }
    }

    if (aListeningCondition==WIDE_H) {
        originalPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
originalLoudnessDensity, aActiveFreqresponse, 3.0);
        distortedPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
distortedLoudnessDensity, aActiveFreqresponse, 3.0);
    } else {
        originalPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
originalLoudnessDensity, aActiveFreqresponse, 0.5);
        distortedPitchLoudnessDensityMainAvg. TimeLpOf (POLQAHandle,
distortedLoudnessDensity, aActiveFreqresponse, 0.5);
    }
    originalLoudnessDensity. AudibleFreqRespCompensationOf (POLQAHandle,
originalPitchLoudnessDensityMainAvg,
                                                            distortedPitchLoudnessDen
sityMainAvg,
```

```
                                                                0.05, 1.0,
                                                                statics->listeningConditi
on);

    oldLoudnessScaleLow = 1.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(10.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){

        distortedLoudnessHulp = distortedLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 7.0, bandIdxLow, bandIdxHigh);
        originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 7.0, bandIdxLow, bandIdxHigh);

        loudnessScaleLow = (originalLoudnessHulp + 0.1)/(distortedLoudnessHulp + 0.1);
        loudnessScaleLow = 0.2*oldLoudnessScaleLow + 0.8*loudnessScaleLow;
        oldLoudnessScaleLow = loudnessScaleLow;
        distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
sqrt(loudnessScaleLow), 0.0, 10.0);
    }

    oldLoudnessScaleLow = 1.0;

    if (aListeningCondition==WIDE_H) {
        bandIdxLow  = originalLoudnessDensity.GetBandLowIdx (0.0);
        bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(5.0);
        for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){

            distortedLoudnessHulp = distortedLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 10.0, bandIdxLow, bandIdxHigh);
            originalLoudnessHulp = originalLoudnessDensity. IntegralLpOverBandRange
(POLQAHandle,  frameIndex, 10.0, bandIdxLow, bandIdxHigh);

            loudnessScaleLow = (originalLoudnessHulp + 0.1)/(distortedLoudnessHulp +
0.1);
            loudnessScaleLow = 0.05*oldLoudnessScaleLow + 0.95*loudnessScaleLow;
            oldLoudnessScaleLow = loudnessScaleLow;
            distortedLoudnessDensity. MultiplyWithOverBandRange (frameIndex,
pow(loudnessScaleLow, 0.4), 0.0, 4.0);
        }
    }

    aDistortedLoudnessMean = 0.0;

    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(16.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangePartial,
bandIdxLow, bandIdxHigh);
            aDistortedLoudnessMean += pow(aDistortedLoudness.m_pData[frameIndex], 0.6);
    }
    aDistortedLoudnessMean /= (numberOfSpeechFrames + 0.5);
    aLoudnessScalingDistorted =
fixedGlobalInternalLevelAdded/(pow(aDistortedLoudnessMean,(1.0/0.6))+1.5);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
            distortedLoudnessDensity. MultiplyWith (frameIndex,
pow(aLoudnessScalingDistorted,0.25));

    aOriginalLoudnessMean = 0.0;
    aDistortedLoudnessMean = 0.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            aOriginalLoudness.m_pData[frameIndex] = originalLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangeComplete ,
bandIdxLow, bandIdxHigh);
            aDistortedLoudness.m_pData[frameIndex] = distortedLoudnessDensity.
IntegralLpOverBandRange (POLQAHandle, frameIndex, LpBandRangeComplete,
bandIdxLow, bandIdxHigh);
            aOriginalLoudnessMean += pow(aOriginalLoudness.m_pData[frameIndex],
```

```
LpLoudnessMeanComplete);
            aDistortedLoudnessMean += pow(aDistortedLoudness.m_pData[frameIndex],
LpLoudnessMeanComplete);
    }
    aOriginalLoudnessMean /= (numberOfSpeechFrames + 0.5);
    aDistortedLoudnessMean /= (numberOfSpeechFrames + 0.5);
    aLoudnessScalingOriginal  =
(pow(aDistortedLoudnessMean,(1.0/LpLoudnessMeanComplete))+0.1*pow(globalScaleDistor
tedToFixedlevel,0.1)) / (pow(aOriginalLoudnessMean,
(1.0/LpLoudnessMeanComplete))+0.1*pow(globalScaleDistortedToFixedlevel,0.1));

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
        originalLoudnessDensity. MultiplyWith (frameIndex,
pow(aLoudnessScalingOriginal, 0.3));
    }

    disturbanceDensityAsymAdd. DifferenceOf (distortedLoudnessDensity,
originalLoudnessDensity);
    mask. MinimumOf (distortedLoudnessDensity, originalLoudnessDensity);

    if (aListeningCondition==WIDE_H) {
        mask *= (XFLOAT) 0.85;
    } else {
        mask *= (XFLOAT) 0.8;
    }

    disturbanceDensityAsymAdd. MaskWith (POLQAHandle, mask);
    disturbanceDensityAsymAdd. MultiplyWithAsymmetryFactorAddOf
(originalPitchPowerDensity, distortedPitchPowerDensity,
statics->listeningCondition, noiseContrastMax1, aSuperSilent,
aDistortedSilencePowerMeanCompensation);

    disturbanceDensityAsymAdd. ComputeLpWeights (POLQAHandle, MINIMUM_POWER_FREQ,
STEP_POWER_FREQ, NUMBER_OF_POWERS_OVER_FREQ, aDoubleTweakedAddedDisturbance);

//END POLQAMAIN PART 1 ADDEDrepeat FOR BIG DISTORTIONS

//END BAD FRAMES PROCESING FOR BIG
DISTORTIONS*****************************************************************************
*

CheckTimeMatInit(POLQAHandle->mh, 3);
//POLQAMAIN PART 2

//Start with calculating the normalized LoudnessDensities for use in timbre and frame
correlations
aDistortedLoudnessMean = 0.0;
bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            hulp1 = distortedLoudnessDensity. IntegralLpOverBandRange (POLQAHandle,
frameIndex, LpBandRangePartial, bandIdxLow, bandIdxHigh);
            aDistortedLoudnessMean += pow(hulp1, LpLoudnessMeanPartial);
    }
    aDistortedLoudnessMean /= (numberOfSpeechFrames + 0.5);
    aLoudnessScalingDistorted =
fixedGlobalInternalLevel/(pow(aDistortedLoudnessMean,(1.0/LpLoudnessMeanPartial))+0
.9);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
        distortedLoudnessDensity. MultiplyWith (frameIndex, aLoudnessScalingDistorted);

//Complete global loudness scaling original towards distorted
    aOriginalLoudnessMean = 0.0;
    aDistortedLoudnessMean = 0.0;
    bandIdxLow  = originalLoudnessDensity.GetBandLowIdx ( 0.0);
    bandIdxHigh = originalLoudnessDensity.GetBandHighIdx(99.0);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){

            hulp1 = originalLoudnessDensity. IntegralLpOverBandRange (POLQAHandle,
frameIndex, LpBandRangeComplete , bandIdxLow, bandIdxHigh);
            hulp2 = distortedLoudnessDensity. IntegralLpOverBandRange (POLQAHandle,
frameIndex, LpBandRangeComplete, bandIdxLow, bandIdxHigh);
```

```
            aOriginalLoudnessMean += pow(hulp1, LpLoudnessMeanComplete);
            aDistortedLoudnessMean += pow(hulp2, LpLoudnessMeanComplete);
    }
    aOriginalLoudnessMean /= (numberOfSpeechFrames + 0.5);
    aDistortedLoudnessMean /= (numberOfSpeechFrames + 0.5);
    aLoudnessScalingOriginal  =
(pow(aDistortedLoudnessMean,(1.0/LpLoudnessMeanComplete))+0.01) /
(pow(aOriginalLoudnessMean, (1.0/LpLoudnessMeanComplete))+0.01);
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
        originalLoudnessDensity. MultiplyWith (frameIndex, aLoudnessScalingOriginal);

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++){
            aOriginalLoudnessCompletelyScaled.m_pData[frameIndex] =
originalLoudnessDensity. IntegralLpOverBandRange (POLQAHandle, frameIndex,
2.0 , bandIdxLow, bandIdxHigh);
            aDistortedLoudnessCompletelyScaled.m_pData[frameIndex] =
distortedLoudnessDensity. IntegralLpOverBandRange (POLQAHandle, frameIndex,
2.0, bandIdxLow, bandIdxHigh);
    }

//++++++++++++++++++++++++++++++++++++++START COMBINING BAD AND NORMAL FRAMES
PROCESING
    for(int badIntervalIndex = 0; badIntervalIndex < numberOfBadIntervals;
badIntervalIndex++)
    {
        for (frameIndex = startFrameBadInterval [badIntervalIndex];
             frameIndex < stopFrameBadInterval [badIntervalIndex];
             frameIndex++)
             {
             for (int i = 0; i < NUMBER_OF_POWERS_OVER_FREQ; i++)
             {
                XFLOAT d = aDisturbance[i].m_pData[frameIndex];
                XFLOAT a = aAddedDisturbance[i].m_pData[frameIndex];
                XFLOAT dd = aDoubleTweakedDisturbance[i].m_pData[frameIndex];
                XFLOAT da = aDoubleTweakedAddedDisturbance[i].m_pData[frameIndex];
                if (d > dd) aDisturbance[i].m_pData[frameIndex] = dd;
                if (a > da) aAddedDisturbance[i].m_pData[frameIndex] = da;
             }
        }
    }
//++++++++++++++++++++++++++++++++++++++END COMBINING BAD AND NORMAL FRAMES PROCESING

    CreateAlignTimeSeries(POLQAHandle, aOriginalTimeSeries, aStartSampleUtterance,
aStopSampleUtterance, aDelayUtterance, GetTransformLength(),
aAlignedOriginalTimeSeries);

    if(mpAlignedPairFile)
        fclose(mpAlignedPairFile);

    XFLOAT quantileDisturbanceOverFile = 1.0;

    XFLOAT hulpLevel = ((XFLOAT) log10 (10.0 + aAvgDistortedPower/1.0e8))/1.02;

    pitchFreqReference = mPitchFreqRef;
    pitchFreqDegraded = mPitchFreqDeg;

//compensation and correction factors including maximum used in final disturbance
calculation
    averageScale = 0.0;
    maxDisturbanceFrame = 0;
    maxDisturbanceOverFile = 0.0;
    frameCorrelationTimeOriginalOld  = 0.0;
    frameCorrelationTimeDistortedOld = 0.0;
    overallAvgDisturbance = 0.0;
    overallMovingAvgDisturbance = 0.0;
    overallMovingAvgDisturbanceOld = 0.0;
    overallMovingAvgDisturbanceOldOld = 0.0;

    SNRloudnessCountTotal = 0;
    SNRloudnessCountExcellent = 0;
    SNRloudnessCountGood = 0;
    SNRloudnessCountFair = 0;
    SNRloudnessCountPoor = 0;
```

```
    SNRloudnessCountBad = 0;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        if (aOriginalLoudnessCompletelyScaled.m_pData[frameIndex]>1.0) {
            SNRloudnessCountTotal++;
            if
(aOriginalLoudnessCompletelyScaled.m_pData[frameIndex]>aDistortedLoudnessCo
mpletelyScaled.m_pData[frameIndex]) {
                hulp = (aOriginalLoudnessCompletelyScaled.m_pData[frameIndex] -
aDistortedLoudnessCompletelyScaled.m_pData[frameIndex])/aOriginalLoudne
ssCompletelyScaled.m_pData[frameIndex];
                if (hulp<=0.7) SNRloudnessCountExcellent++;
                if ((hulp>1.0) && (hulp<=2.0)) SNRloudnessCountGood++;
                if ((hulp>2.0) && (hulp<=3.0)) SNRloudnessCountFair++;
                if ((hulp>3.0) && (hulp<=4.0)) SNRloudnessCountPoor++;
                if (hulp>4.0) SNRloudnessCountBad++;
            } else {
                hulp = (aDistortedLoudnessCompletelyScaled.m_pData[frameIndex] -
aOriginalLoudnessCompletelyScaled.m_pData[frameIndex])/aOriginalLoudnes
sCompletelyScaled.m_pData[frameIndex];
                if (hulp<1.5) SNRloudnessCountExcellent++;
                if ((hulp>1.0) && (hulp<=2.0)) SNRloudnessCountGood++;
                if ((hulp>2.0) && (hulp<=3.0)) SNRloudnessCountFair++;
                if ((hulp>3.0) && (hulp<=4.0)) SNRloudnessCountPoor++;
                if (hulp>4.0) SNRloudnessCountBad++;
            }
        }
    }
    SNRloudnessWeightedRatio1 =
(SNRloudnessCountExcellent+0.1)/(SNRloudnessCountTotal+0.1);
    SNRloudnessWeightedRatio2 = pow (SNRloudnessWeightedRatio1, 0.1);
    SNRloudnessWeightedRatio3 = pow (SNRloudnessWeightedRatio1, 0.2);
    SNRloudnessWeightedRatio4 = pow (SNRloudnessWeightedRatio1, 0.33);
    SNRloudnessWeightedRatio5 = pow (SNRloudnessWeightedRatio1, 0.4);
    SNRloudnessWeightedRatio1 = pow (SNRloudnessWeightedRatio1, 0.03);

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        overallAvgDisturbance += aDisturbance[2].m_pData[frameIndex];
        if (aDisturbance[2].m_pData[frameIndex] > maxDisturbanceOverFile) {
            maxDisturbanceOverFile = aDisturbance[2].m_pData[frameIndex];
            maxDisturbanceFrame = frameIndex;
        }
    }
    overallAvgDisturbance /= ( numberOfSpeechFrames + 0.01);
    hulp1 = overallAvgDisturbance - 1.0;
    if (hulp1 < 2.0) hulp1 = 2.0;
    maxDisturbance = 130.0 + maxDisturbanceOverFile/hulp1;

    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        if (aDisturbance[5].m_pData[frameIndex] > maxDisturbanceOverFile/6.0) {
            quantileDisturbanceOverFile += 1.0;
        }
    }
    quantileDisturbanceOverFile /= (numberOfSpeechFrames+1.0);
    quantileDisturbanceOverFile = pow(quantileDisturbanceOverFile,0.011);

    XFLOAT varianceDisturbanceDown = 0.0;
    XFLOAT varianceDisturbanceUp = 0.0;
    for (frameIndex = (statics->startFrameIdx+1); frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        varianceDisturbanceDown += (aDisturbance[2].m_pData[frameIndex] -
aDisturbance[2].m_pData[frameIndex-1]);
        if (varianceDisturbanceDown > 0.0) varianceDisturbanceDown = 0.0;
        varianceDisturbanceUp += (aDisturbance[2].m_pData[frameIndex-1] -
aDisturbance[2].m_pData[frameIndex]);
        if (varianceDisturbanceUp > 0.0) varianceDisturbanceUp = 0.0;
    }
    varianceDisturbanceUp /= ( numberOfSpeechFrames + 0.01);
    varianceDisturbanceDown /= ( numberOfSpeechFrames + 0.01);
    if (varianceDisturbanceUp < -0.85) varianceDisturbanceUp = -0.85;
    if (varianceDisturbanceDown < -0.85) varianceDisturbanceDown = -0.85;

    globalScaleCorrection = pow(globalScaleCorrection,0.06) + 0.02;
    if (globalScaleCorrection<1.0) globalScaleCorrection = 1.0;
```

```
    scaleCorrectionQualityPlusOld = 1.0;
    scaleCorrectionQualityPlusAddedOld = 1.0;
    scaleCorrectionIntellOld = 1.0;
    scaleCorrectionMusicOld = 1.0;

    frameCorrelationTimeDisturbanceAvgCompensation000 = 0.0;
    frameCorrelationTimeDisturbanceAvgCompensation000silent = 0.0;

    hulpLowOld = 0.0;
    hulpHighOld = 0.0;
    hulpLowOldNarrowband = 0.0;
    hulpHighOldNarrowband = 0.0;
    distortedLoudnessTimbrePerFrameNarrowbandAvg = 0.0;
    distortedLoudnessTimbreHighPerFrameAvg = 0.0;
    distortedLoudnessTimbreHighPerFrameAvgSilent = 0.0;
    distortedLoudnessTimbreHighPerFrameAvgActive = 0.0;
    originalLoudnessTimbrePerFrameDifferenceOld = 0.0;
    distortedLoudnessTimbrePerFrameDifferenceOld = 0.0;
//FINAL DISTURBANCE CALCULATION PER FRAME FOR ALL Lpqr powers OF WHICH A SUBSET IS USED
IN THE FINAL MOS MODEL, START WITH COMPENSATION FACTORS

    int LastVoicedFrame=-1;
    XFLOAT sumWeights = 0;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
    {
        if (UseThisFrame[frameIndex])
        {

            XFLOAT NearUnvoicedSectionEnd=0;

            {
                bool FrameIsVoiced = mpPitchVec[frameIndex]>30;
                if (FrameIsVoiced)
                {
                    for (i=frameIndex+1; i<frameIndex+(2.0)+1 &&
i<=statics->stopFrameIdx && mpPitchVec[i]>30; i++);
                    NearUnvoicedSectionEnd = (((0) > ((((1.0) <
(1-(i-frameIndex-1)/(2.0))) ? (1.0) : (1-(i-frameIndex-1)/(2.0)))))
? (0) : ((((1.0) < (1-(i-frameIndex-1)/(2.0))) ? (1.0) :
(1-(i-frameIndex-1)/(2.0)))));
                    LastVoicedFrame = frameIndex;
                }
                else if (LastVoicedFrame>=0)
                {
                    NearUnvoicedSectionEnd = (((0) > ((((1.0) < (1-(frameIndex -
LastVoicedFrame)/(2.0))) ? (1.0) : (1-(frameIndex -
LastVoicedFrame)/(2.0)))))) ? (0) : ((((1.0) < (1-(frameIndex -
LastVoicedFrame)/(2.0))) ? (1.0) : (1-(frameIndex -
LastVoicedFrame)/(2.0)))));
                }
            }

        scaleCorrectionQuality = (XFLOAT) (aOriginalTotalPower.m_pData[frameIndex] +
(XFLOAT) 1.0) / (aDistortedTotalPower.m_pData[frameIndex] + (XFLOAT) 1.0);

        scaleCorrectionQualityPlus = (XFLOAT) (aOriginalTotalPower.m_pData[frameIndex]
+ (XFLOAT) 100.0) / (aDistortedTotalPower.m_pData[frameIndex] + (XFLOAT)
100.0);
        scaleCorrectionQualityPlusAdded = (XFLOAT)
(aOriginalTotalPower.m_pData[frameIndex] + (XFLOAT) 100.0) /
(aDistortedTotalPower.m_pData[frameIndex] + (XFLOAT) 100.0);
        scaleCorrectionIntell = scaleCorrectionQuality;
        scaleCorrectionMusic = scaleCorrectionQuality;
        if (scaleCorrectionQuality<1.0)
        {
            scaleCorrectionQuality = (XFLOAT) 1.0;
            if (aSilent.m_pData[frameIndex]) {
                if (aListeningCondition==WIDE_H) {
                    scaleCorrectionQualityPlus = (XFLOAT) 1.0 / (XFLOAT) pow (
scaleCorrectionQualityPlus , (XFLOAT) 0.004);
                    scaleCorrectionQualityPlusAdded = (XFLOAT) pow (
scaleCorrectionQualityPlusAdded , (XFLOAT) 0.07);
                } else {
                    scaleCorrectionQualityPlus = (XFLOAT) 1.0;
```

```
                    scaleCorrectionQualityPlusAdded = (XFLOAT) pow (
scaleCorrectionQualityPlusAdded , (XFLOAT) 0.05);
                }
                scaleCorrectionIntell = (XFLOAT) pow ( scaleCorrectionIntell , (XFLOAT)
0.01);
                scaleCorrectionMusic = (XFLOAT) pow ( scaleCorrectionMusic , (XFLOAT)
0.06);
            } else {
                if (aListeningCondition==WIDE_H) {
                    scaleCorrectionQualityPlus = (XFLOAT) 1.0 ;
                    scaleCorrectionQualityPlusAdded = (XFLOAT) 1.0 / (XFLOAT) pow (
scaleCorrectionQualityPlusAdded , (XFLOAT) 0.02);
                } else {
                    scaleCorrectionQualityPlus = pow ( scaleCorrectionQualityPlus ,
(XFLOAT) 0.004);
                    scaleCorrectionQualityPlusAdded = (XFLOAT) 1.0 ;
                }
                scaleCorrectionIntell = (XFLOAT) 1.0 /(XFLOAT) pow (
scaleCorrectionIntell , (XFLOAT) 0.004);
                scaleCorrectionMusic = (XFLOAT) pow ( scaleCorrectionMusic , (XFLOAT)
0.02);
            }
        } else {
            scaleCorrectionQuality = (XFLOAT) pow ( scaleCorrectionQuality , (XFLOAT)
0.002);
            if (aSilent.m_pData[frameIndex]) {
                if (aListeningCondition==WIDE_H) {
                    scaleCorrectionQualityPlus = (XFLOAT) 1.0 / (XFLOAT) pow (
scaleCorrectionQualityPlus , (XFLOAT) 0.009);
                } else {
                    scaleCorrectionQualityPlus = (XFLOAT) pow (
scaleCorrectionQualityPlus , (XFLOAT) 0.012);
                }
                scaleCorrectionQualityPlusAdded = (XFLOAT) 1.0 / (XFLOAT) pow (
scaleCorrectionQualityPlusAdded , (XFLOAT) 0.1);
                scaleCorrectionIntell = (XFLOAT) 1.0 / (XFLOAT) pow (
scaleCorrectionIntell , (XFLOAT) 0.003);
                scaleCorrectionMusic = (XFLOAT) pow ( scaleCorrectionMusic , (XFLOAT)
0.01);
            } else {
                if (aListeningCondition==WIDE_H) {
                    scaleCorrectionQualityPlus = (XFLOAT) 1.0 ;
                } else {
                    scaleCorrectionQualityPlus = (XFLOAT) pow (
scaleCorrectionQualityPlus , (XFLOAT) 0.01);
                }
                scaleCorrectionQualityPlusAdded = (XFLOAT) 1.0 ;
                scaleCorrectionIntell = (XFLOAT) 1.0 / pow ( scaleCorrectionIntell ,
(XFLOAT) 0.002);
                scaleCorrectionMusic = (XFLOAT) pow ( scaleCorrectionMusic , (XFLOAT)
0.03);
            }
        }
        if (frameIndex > (statics->startFrameIdx + 1) ) scaleCorrectionIntell *=
pow(scaleCorrectionIntellOld, 0.1);
        scaleCorrectionMusic *= pow(scaleCorrectionMusicOld, 0.2);
        scaleCorrectionQualityPlusOld = scaleCorrectionQualityPlus;
        scaleCorrectionQualityPlusAddedOld = scaleCorrectionQualityPlusAdded;
        scaleCorrectionIntellOld = scaleCorrectionIntell;
        scaleCorrectionMusicOld = scaleCorrectionMusic;

        overallMovingAvgDisturbance = 0.01*aDisturbance[4].m_pData[frameIndex] +
0.09*overallMovingAvgDisturbanceOld + 0.9*overallMovingAvgDisturbanceOldOld;
        overallMovingAvgDisturbanceOldOld = overallMovingAvgDisturbanceOld;
        overallMovingAvgDisturbanceOld = overallMovingAvgDisturbance;

        aTimeWeight.m_pData[frameIndex] = 1.0;
        sumWeights++;

        if (statics->nrSpeechFrames > 1000)
        {
            XFLOAT timeWeightFactor = (statics->nrSpeechFrames - (XFLOAT) 1000) /
(XFLOAT) 5500;
            if (timeWeightFactor > (XFLOAT) 0.5) timeWeightFactor = (XFLOAT) 0.5;
            aTimeWeight.m_pData[frameIndex] = (XFLOAT) (((XFLOAT) 1.0 -
timeWeightFactor) + timeWeightFactor * (XFLOAT) frameIndex / (XFLOAT)
```

```
statics->nrSpeechFrames);
        }

        if (frameIndex>(statics->startFrameIdx+2)) {
            frameCorrelationTimeOriginal =  originalLoudnessDensity.
FrameCorrelationTime (frameIndex, statics->nrFrames);
            frameCorrelationTimeDistorted = distortedLoudnessDensity.
FrameCorrelationTime (frameIndex, statics->nrFrames);
            frameCorrelationTimeDisturbance = disturbanceDensity. FrameCorrelationTime
(frameIndex, statics->nrFrames);
            frameCorrelationTimeDisturbanceAvgCompensation000 +=
frameCorrelationTimeDisturbance;
            if (aSilent.m_pData[frameIndex])
frameCorrelationTimeDisturbanceAvgCompensation000silent +=
frameCorrelationTimeDisturbance;
        } else {
            frameCorrelationTimeOriginal = 0.0;
            frameCorrelationTimeDistorted = 0.0;
            frameCorrelationTimeDisturbance = 0.0;
        }

        frameCorrelationTimeOriginal  = 0.2*frameCorrelationTimeOriginal +
0.8*frameCorrelationTimeOriginalOld;
        frameCorrelationTimeDistorted = 0.2*frameCorrelationTimeDistorted +
0.8*frameCorrelationTimeDistortedOld;
        frameCorrelationTimeOriginalOld  = frameCorrelationTimeOriginal;
        frameCorrelationTimeDistortedOld = frameCorrelationTimeDistorted;

        if (aListeningCondition==WIDE_H) {
            if (frameCorrelationTimeOriginal<0.55) frameCorrelationTimeOriginal = 0.55;
            if (frameCorrelationTimeDistorted<0.55) frameCorrelationTimeDistorted =
0.55;
            if (frameCorrelationTimeOriginal<frameCorrelationTimeDistorted)
                frameCorrelationTimeCompensationDifference = 1.0 +
(frameCorrelationTimeDistorted-frameCorrelationTimeOriginal);
            else
                frameCorrelationTimeCompensationDifference = 1.0;
        } else {
            if (frameCorrelationTimeOriginal<0.5) frameCorrelationTimeOriginal = 0.5;
            if (frameCorrelationTimeDistorted<0.5) frameCorrelationTimeDistorted = 0.5;
            if (frameCorrelationTimeOriginal<frameCorrelationTimeDistorted)
                frameCorrelationTimeCompensationDifference = 2.0 +
(frameCorrelationTimeDistorted-frameCorrelationTimeOriginal)/2.0;
            else
                frameCorrelationTimeCompensationDifference = 2.0 +
(frameCorrelationTimeOriginal-frameCorrelationTimeDistorted)/2.0;
        }

        frameCorrelationTimeOriginal =  pow(frameCorrelationTimeOriginal,15.0);
        frameCorrelationTimeDistorted = pow(frameCorrelationTimeDistorted,15.0);
        frameCorrelationTimeCompensationOriginal = 1.0 - frameCorrelationTimeOriginal;
        frameCorrelationTimeCompensationDistorted = 1.0 -
frameCorrelationTimeDistorted;
        if (frameCorrelationTimeCompensationOriginal<0.4)
frameCorrelationTimeCompensationOriginal = 0.4;
        if (frameCorrelationTimeCompensationDistorted<0.4)
frameCorrelationTimeCompensationDistorted = 0.4;

//compensation factor per frame for spectral flatness disturbance (pure tone=0.0 <
spectral flatness < 1.0=pure white noise)
        frameFlatnessTimeOriginal = originalLoudnessDensity. SpectralFlatness
(frameIndex);
        frameFlatnessTimeDistorted =    distortedLoudnessDensity. SpectralFlatness
(frameIndex);

        frameFlatnessDisturbance = disturbanceDensity. SpectralFlatness (frameIndex);

        if (aListeningCondition==WIDE_H) {
            if (frameFlatnessDisturbance<0.5) frameFlatnessDisturbance = 0.5;
            frameFlatnessDisturbance = (1.0+frameFlatnessDisturbance);
        } else {
            if (frameFlatnessDisturbance<0.4) frameFlatnessDisturbance = 0.4;
            frameFlatnessDisturbance =  pow((1.0+frameFlatnessDisturbance),1.3);
        }
        frameFlatnessDisturbanceAdded = pow(frameFlatnessDisturbance,1.25);
```

```
//compensation factor per frame for timbre differences in speech
        if (aActiveFreqresponse.m_pData[frameIndex]) {
            hulpLowNarrowband = distortedLoudnessDensity.IntegralLowNarrowband
(POLQAHandle, frameIndex);
            hulpHighNarrowband = distortedLoudnessDensity.IntegralHighNarrowband
(frameIndex);
            distortedLoudnessTimbrePerFrameNarrowband  =
4.0*hulpHighNarrowband-hulpLowNarrowband;
            if (distortedLoudnessTimbrePerFrameNarrowband  <0.0)
distortedLoudnessTimbrePerFrameNarrowband = 0.0;
        } else {
            distortedLoudnessTimbrePerFrameNarrowband = 0.0;
        }
        distortedLoudnessTimbrePerFrameNarrowbandAvg +=
distortedLoudnessTimbrePerFrameNarrowband;

//compensation factor per frame for timbre differences in speech and noise
        originalLoudnessTimbrePerFrame = (originalLoudnessDensity.IntegralLow2
(frameIndex, statics->listeningCondition) -
originalLoudnessDensity.IntegralHigh2 (frameIndex,
statics->listeningCondition));
        distortedLoudnessTimbrePerFrame  = (distortedLoudnessDensity.IntegralLow2
(frameIndex,
statics->listeningCondition)-distortedLoudnessDensity.IntegralHigh2
(frameIndex, statics->listeningCondition));
        if ( (frameIndex>(statics->startFrameIdx+2)) &&
(!aSuperSilent.m_pData[frameIndex]) ) {
            originalLoudnessTimbrePerFrameDifference =
(originalLoudnessDensity.IntegralHigh3 (frameIndex-2) -
originalLoudnessDensity.IntegralHigh3 (frameIndex));
            distortedLoudnessTimbrePerFrameDifference =
(distortedLoudnessDensity.IntegralHigh3 (frameIndex-2) -
distortedLoudnessDensity.IntegralHigh3 (frameIndex));
        } else {
            originalLoudnessTimbrePerFrameDifference = 0.0;
            distortedLoudnessTimbrePerFrameDifference = 0.0;
        }
        if (originalLoudnessTimbrePerFrameDifference<0.0)
originalLoudnessTimbrePerFrameDifference = 0.0;
        if (distortedLoudnessTimbrePerFrameDifference<1.0)
distortedLoudnessTimbrePerFrameDifference = 1.0;
        if (distortedLoudnessTimbrePerFrameDifference>9.0)
distortedLoudnessTimbrePerFrameDifference = 9.0;
        originalLoudnessTimbrePerFrameDifference +=
originalLoudnessTimbrePerFrameDifferenceOld;
        distortedLoudnessTimbrePerFrameDifference +=
distortedLoudnessTimbrePerFrameDifferenceOld;

        originalLoudnessTimbrePerFrameDifferenceOld =
originalLoudnessTimbrePerFrameDifference;
        distortedLoudnessTimbrePerFrameDifferenceOld =
distortedLoudnessTimbrePerFrameDifference;

        if (aListeningCondition==WIDE_H){
            originalLoudnessTimbrePerFrameDifferenceCompensation = pow(
(originalLoudnessTimbrePerFrameDifference+1.0)/(distortedLoudnessTimbrePerF
rameDifference+1.0), 0.005);
            if (originalLoudnessTimbrePerFrameDifferenceCompensation <1.0)
originalLoudnessTimbrePerFrameDifferenceCompensation =1.0;
            originalLoudnessTimbrePerFrameDifferenceCompensationAdd = pow(
(originalLoudnessTimbrePerFrameDifference+1.0)/(distortedLoudnessTimbrePerF
rameDifference+1.0), 0.04);
            if (originalLoudnessTimbrePerFrameDifferenceCompensationAdd <0.95)
originalLoudnessTimbrePerFrameDifferenceCompensationAdd =0.95;
        } else {
            originalLoudnessTimbrePerFrameDifferenceCompensation = pow(
(originalLoudnessTimbrePerFrameDifference+1.0)/(distortedLoudnessTimbrePerF
rameDifference+1.0), 0.006);
            if (originalLoudnessTimbrePerFrameDifferenceCompensation <1.0)
originalLoudnessTimbrePerFrameDifferenceCompensation =1.0;
            originalLoudnessTimbrePerFrameDifferenceCompensationAdd = pow(
(originalLoudnessTimbrePerFrameDifference+1.0)/(distortedLoudnessTimbrePerF
rameDifference+1.0), 0.04);
            if (originalLoudnessTimbrePerFrameDifferenceCompensationAdd <1.0)
originalLoudnessTimbrePerFrameDifferenceCompensationAdd =1.0;
        }
```

```
        differenceInLoudnessTimbrePerFrame  =
fabs(distortedLoudnessTimbrePerFrame-originalLoudnessTimbrePerFrame);

        hulpLow = 0.7*distortedLoudnessDensity.IntegralLow2 (frameIndex,
statics->listeningCondition) + 0.3*hulpLowOld;
        hulpHigh = 0.7*distortedLoudnessDensity.IntegralHigh2 (frameIndex,
statics->listeningCondition) + 0.3*hulpHighOld;

        hulpLowOld = hulpLow;
        hulpHighOld = hulpHigh;

        distortedLoudnessTimbrePerFrame  = fabs(hulpLow-2.0*hulpHigh) - 140.0;
        if (distortedLoudnessTimbrePerFrame  <0.0) distortedLoudnessTimbrePerFrame =
0.0;

        distortedLoudnessTimbreHighPerFrame  =
2.3*hulpHigh*SampleRateRatioCompensation4-hulpLow;

        if (distortedLoudnessTimbreHighPerFrame  <0.0)
distortedLoudnessTimbreHighPerFrame = 0.0;

        if (aSilent.m_pData[frameIndex]) {
            distortedLoudnessTimbreHighPerFrameAvgSilent +=
distortedLoudnessTimbreHighPerFrame;
        } else {
            distortedLoudnessTimbreHighPerFrameAvgActive +=
distortedLoudnessTimbreHighPerFrame;
        }
        distortedLoudnessTimbreHighPerFrameAvg += distortedLoudnessTimbreHighPerFrame;
        XFLOAT silentWeight = pow(
(numberOfSuperSilentFrames+10.0)/(numberOfSpeechFrames+10.0),0.1);

//FINAL DISTURBANCE CALCULATION IN EACH FRAME FOR AL Lp's OVER FREQUENCY OF WHICH A
SUBSET IS USED IN THE FINAL MOS MODEL
        for (i = 0; i < NUMBER_OF_POWERS_OVER_FREQ; i++) {

            if (aListeningCondition==WIDE_H){
                aDisturbance[i].m_pData[frameIndex] *=
0.87*originalLoudnessTimbrePerFrameDifferenceCompensation;
                aAddedDisturbance[i].m_pData[frameIndex] *=
0.85*originalLoudnessTimbrePerFrameDifferenceCompensationAdd;
            } else {
                aDisturbance[i].m_pData[frameIndex] *=
1.07*originalLoudnessTimbrePerFrameDifferenceCompensation;
                aAddedDisturbance[i].m_pData[frameIndex] *=
1.1*originalLoudnessTimbrePerFrameDifferenceCompensationAdd;
            }

//LOW LEVEL compensation for global zero output in case of severe time clipping
            aDisturbance[i].m_pData[frameIndex] /= globalScaleCorrectionActive;
            aAddedDisturbance[i].m_pData[frameIndex] /=
globalScaleCorrectionActiveAdded;

            if (aListeningCondition==WIDE_H){
                aDisturbance[i].m_pData[frameIndex] /= pow((aPowerRatioaAvg+1.0),0.02);
                aAddedDisturbance[i].m_pData[frameIndex] /=
pow((aPowerRatioaAvg+1.0),0.14);
                aDisturbance[i].m_pData[frameIndex] *= scaleCorrectionQualityPlus;
                aAddedDisturbance[i].m_pData[frameIndex] *=
scaleCorrectionQualityPlusAdded;
            } else {
                aDisturbance[i].m_pData[frameIndex] /= pow((aPowerRatioaAvg+1.0),0.02);
                aAddedDisturbance[i].m_pData[frameIndex] /=
pow((aPowerRatioaAvg+1.0),0.1);
                aDisturbance[i].m_pData[frameIndex] *= scaleCorrectionQualityPlus;
                aAddedDisturbance[i].m_pData[frameIndex] *=
scaleCorrectionQualityPlusAdded;
            }

//BEGIN compensation per frame factor for correlation differences, FRAME REPEAT repeat
modelling

            if (aListeningCondition==WIDE_H){
                aDisturbance[i].m_pData[frameIndex] *=
1.04*pow(frameCorrelationTimeCompensationDifference,0.8);
```

```
                aAddedDisturbance[i].m_pData[frameIndex] *= 1.04;
            } else {
                aDisturbance[i].m_pData[frameIndex] *= pow(
(frameCorrelationTimeCompensationDifference+1.0),0.07);
                aAddedDisturbance[i].m_pData[frameIndex] *= pow(
(frameCorrelationTimeCompensationDifference+1.0),0.07);
            }

            if (aListeningCondition==WIDE_H){
                aDisturbance[i].m_pData[frameIndex] /= pow(hulpLevel,0.4);
                aAddedDisturbance[i].m_pData[frameIndex] *= pow(hulpLevel,0.4);
            } else {
                aDisturbance[i].m_pData[frameIndex] /= pow(hulpLevel,0.4);
                aAddedDisturbance[i].m_pData[frameIndex] /= pow(hulpLevel,0.1);
            }

//compensation factor per frame for TIMBRE differences
            if ( frameIndex>(statics->startFrameIdx+10) &&
aActiveFreqresponse.m_pData[frameIndex]) {

                if (aListeningCondition==WIDE_H) {
                    aAddedDisturbance[i].m_pData[frameIndex] *=
pow((1.0+distortedLoudnessTimbrePerFrameNarrowband),0.04);
                } else {
                    aAddedDisturbance[i].m_pData[frameIndex] *=
pow((1.0+distortedLoudnessTimbrePerFrameNarrowband),0.03);
                }

            }

            if (frameIndex>(statics->startFrameIdx)) {
                if (aListeningCondition==WIDE_H) {

                    aAddedDisturbance[i].m_pData[frameIndex] +=
0.006*distortedLoudnessTimbreHighPerFrame/noiseIndicatorTimbreAdd;
                    aDisturbance[i].m_pData[frameIndex] /=
pow((1.0+distortedLoudnessTimbreHighPerFrame/noiseIndicatorTimbre),
0.08);
                    aAddedDisturbance[i].m_pData[frameIndex] /=
pow((1.0+distortedLoudnessTimbreHighPerFrame/noiseIndicatorTimbre),
0.03);

                } else {
                    aDisturbance[i].m_pData[frameIndex] /=
pow((1.0+distortedLoudnessTimbreHighPerFrame/noiseIndicatorTimbre),
0.02);
                }
            }

//compensation factor for SPECTRAL FLATNESS

            if (aSilent.m_pData[frameIndex]) {
                if (aListeningCondition==WIDE_H) {
                    aDisturbance[i].m_pData[frameIndex] *=
(frameFlatnessDisturbance/frameFlatnessDisturbanceAvgCompensationSi
lent);
                    aAddedDisturbance[i].m_pData[frameIndex] *=
(frameFlatnessDisturbanceAdded/frameFlatnessDisturbanceAvgCompensat
ionAddedSilent);
                } else {
                    aDisturbance[i].m_pData[frameIndex] *=
pow((frameFlatnessDisturbance/frameFlatnessDisturbanceAvgCompensati
onSilent),0.6);
                    aAddedDisturbance[i].m_pData[frameIndex] *=
pow((frameFlatnessDisturbanceAdded/frameFlatnessDisturbanceAvgCompe
nsationAddedSilent),0.6);
                }
            } else {
                if (aListeningCondition==WIDE_H) {
                    aDisturbance[i].m_pData[frameIndex] *=
(frameFlatnessDisturbance/frameFlatnessDisturbanceAvgCompensationAc
tive);
                    aAddedDisturbance[i].m_pData[frameIndex] *=
(frameFlatnessDisturbanceAdded/frameFlatnessDisturbanceAvgCompensat
ionAddedActive);
                } else {
```

```
                    aDisturbance[i].m_pData[frameIndex] *=
pow((frameFlatnessDisturbance*frameFlatnessDisturbanceAvgCompensati
onActive),0.4);
                    aAddedDisturbance[i].m_pData[frameIndex] *=
pow((frameFlatnessDisturbanceAdded*frameFlatnessDisturbanceAvgCompe
nsationAddedActive),0.65);
                }
            }

//compensation for NOISE CONTRAST IN SILENT PERIODS

            if (aListeningCondition==WIDE_H) {
                if (aSuperSilent.m_pData[frameIndex])
aDisturbance[i].m_pData[frameIndex] *= pow(noiseContrastMax1,0.8);
            } else {
                if (aSuperSilent.m_pData[frameIndex])
aDisturbance[i].m_pData[frameIndex] *= pow(noiseContrastMax1,0.6);
            }

//compensation for delay ALIGN JUMPS

        if (aListeningCondition==WIDE_H) {
            if (FrameFlags[frameIndex] & 0x0000001 ) {
                for ( count = (-noiseIndicatorAlignJumpsIntWB); count < 0; count++) {
                    hulp = (1.0 -
1.0*count/(delayJumpCompWB*noiseIndicatorAlignJumpsIntWB))/(1.0+1.0
/delayJumpCompWB);
                    if (frameIndex>(statics->startFrameIdx-count)) {
                        aDisturbance[i].m_pData[frameIndex+count]  *= pow(hulp,0.6);
                        aAddedDisturbance[i].m_pData[frameIndex+count]  *=
pow(hulp,0.6);
                    }
                }
                for (count = 0; count < (noiseIndicatorAlignJumpsIntWB+1); count++) {
                    hulp = (1.0 +
1.0*count/(delayJumpCompWB*noiseIndicatorAlignJumpsIntWB))/(1.0+1.0
/delayJumpCompWB);
                    if (frameIndex<(statics->stopFrameIdx-count )) {
                        aDisturbance[i].m_pData[frameIndex+count]  *= pow(hulp,0.6);
                        aAddedDisturbance[i].m_pData[frameIndex+count] *=
pow(hulp,0.6);
                    }
                }
            }
        } else {
            if (FrameFlags[frameIndex] & 0x0000001 ) {
                for ( count = (-noiseIndicatorAlignJumpsIntNB); count < 0; count++) {
                    hulp = (1.0 -
1.0*count/(delayJumpCompNB*noiseIndicatorAlignJumpsIntNB))/(1.0+1.0
/delayJumpCompNB);
                    if (frameIndex>(statics->startFrameIdx-count)) {
                        aDisturbance[i].m_pData[frameIndex+count]  /= pow(hulp,0.6);
                        aAddedDisturbance[i].m_pData[frameIndex+count]  /=
pow(hulp,0.6);
                    }
                }
                for (count = 0; count < (noiseIndicatorAlignJumpsIntNB+1); count++) {
                    hulp = (1.0 +
1.0*count/(delayJumpCompNB*noiseIndicatorAlignJumpsIntNB))/(1.0+1.0
/delayJumpCompNB);
                    if (frameIndex<(statics->stopFrameIdx-count )) {
                        aDisturbance[i].m_pData[frameIndex+count]  /= pow(hulp,0.6);
                        aAddedDisturbance[i].m_pData[frameIndex+count] /=
pow(hulp,0.6);
                    }
                }
            }
        }

            if (aListeningCondition==STANDARD_IRS) {
                if (aDisturbance[i].m_pData[frameIndex] < 1.0)
aDisturbance[i].m_pData[frameIndex] = 1.0;
                if (aAddedDisturbance[i].m_pData[frameIndex] < 1.0)
aAddedDisturbance[i].m_pData[frameIndex] = 1.0;
            } else {
                if (aDisturbance[i].m_pData[frameIndex] < 1.0)
```

```
aDisturbance[i].m_pData[frameIndex] = 1.0;
                if (aAddedDisturbance[i].m_pData[frameIndex] < 1.0)
aAddedDisturbance[i].m_pData[frameIndex] = 1.0;
            }

//CLIP TO MAXIMUM DEGRADATION

            if (aListeningCondition==WIDE_H) {
                if (aDisturbance[i].m_pData[frameIndex]      >
1.07*maxDisturbance*globalScaleCorrectionIntellLevelCorrectionForMaximu
mD)      aDisturbance[i].m_pData[frameIndex] =
1.07*maxDisturbance*globalScaleCorrectionIntellLevelCorrectionForMaximu
mD;
                if (aAddedDisturbance[i].m_pData[frameIndex] >
(2.0*maxDisturbance*globalScaleCorrectionIntellLevelCorrectionForMaximu
mA)) aAddedDisturbance[i].m_pData[frameIndex] =
2.0*maxDisturbance*globalScaleCorrectionIntellLevelCorrectionForMaximum
A;
            } else {
                if (aDisturbance[i].m_pData[frameIndex] > 130.0)
aDisturbance[i].m_pData[frameIndex] = 130.0;
                if (aAddedDisturbance[i].m_pData[frameIndex] > 340.0)
aAddedDisturbance[i].m_pData[frameIndex] = 340.0;
            }

            if (aListeningCondition==WIDE_H) {
                hulp1 = (aOriginalLoudness.m_pData[frameIndex]) - 28.0;
                if (hulp1<1.0) hulp1 = 1.0;
                hulp1 = pow (hulp1, 0.3);
                aDisturbance[i].m_pData[frameIndex] /= hulp1;
            } else {
                hulp1 = (aOriginalLoudness.m_pData[frameIndex]) - 24.0;
                if (hulp1<1.0) hulp1 = 1.0;
                hulp1 = pow (hulp1, 0.4);
                aDisturbance[i].m_pData[frameIndex] /= hulp1;
                hulp1 = (aDistortedLoudness.m_pData[frameIndex]) - 33.0;
                if (hulp1<1.0) hulp1 = 1.0;
                hulp1 = pow (hulp1, 0.1);
                aDisturbance [i][frameIndex] *= hulp1;
            }

            if (aListeningCondition==WIDE_H) {
                hulp1 = (aOriginalLoudness.m_pData[frameIndex]) - 18.0;
                if (hulp1<1.0) hulp1=1.0;
                hulp1 = pow (hulp1, 0.20*pow(aPureFrqLoudnessMeanCompensation,2.0));
                aAddedDisturbance[i].m_pData[frameIndex] /= hulp1;
            } else {
                hulp1 = (aOriginalLoudness.m_pData[frameIndex]) - 15.0;
                if (hulp1<1.0) hulp1=1.0;
                hulp1 = pow (hulp1, 0.20/pow(aPureFrqLoudnessMeanCompensation,3.0));
                aAddedDisturbance[i].m_pData[frameIndex] /= hulp1;
            }
            if (aListeningCondition==WIDE_H) {
                hulp1 = (aDistortedLoudness.m_pData[frameIndex]) - 23.0;
                if (hulp1<1.0) hulp1=1.0;
                hulp1 = pow (hulp1, 0.18*pow(aPureFrqLoudnessMeanCompensation,2.0));
                aAddedDisturbance[i].m_pData[frameIndex] /= hulp1;
            } else {
                hulp1 = (aOriginalLoudness.m_pData[frameIndex]) - 25.0;
                if (hulp1<1.0) hulp1=1.0;
                hulp1 = pow (hulp1, 0.12*pow(aPureFrqLoudnessMeanCompensation,3.0));
                aAddedDisturbance[i].m_pData[frameIndex] /= hulp1;
            }
            if (aListeningCondition==WIDE_H) {
                aDisturbance[i].m_pData[frameIndex] *= pow(fractionOfUsedFrames,0.75);
                aAddedDisturbance[i].m_pData[frameIndex] *=
pow(fractionOfUsedFrames,0.7);
                aAddedDisturbance [i][frameIndex] *= pow(fractionOfSilentFrames,0.05);
            } else {
                aDisturbance[i].m_pData[frameIndex] *= pow(fractionOfUsedFrames,0.9);
                aAddedDisturbance[i].m_pData[frameIndex] *=
pow(fractionOfUsedFrames,0.95);
                aAddedDisturbance[i].m_pData[frameIndex] *=
pow(fractionOfSilentFrames,0.6);
            }
```

```
    if (aListeningCondition==STANDARD_IRS) aAddedDisturbance [i][frameIndex] /=
pow((frameCorrelationTimeDisturbance+2.0),0.1);

//compensation for DISTURBANCE VARIANCE

            if (aListeningCondition==WIDE_H) {
                aDisturbance[i].m_pData[frameIndex] *=
pow((varianceDisturbanceUp+1.0),0.7);
                aAddedDisturbance[i].m_pData[frameIndex] *=
pow((varianceDisturbanceUp+1.0),0.9);
                aDisturbance[i].m_pData[frameIndex] *=
pow((varianceDisturbanceDown+1.0),0.1);
            } else {
                aDisturbance[i].m_pData[frameIndex] *=
pow((varianceDisturbanceUp+1.0),0.4);
                aAddedDisturbance[i].m_pData[frameIndex] *=
pow((varianceDisturbanceUp+1.0),0.4);
            }

            if (frameIndex>(statics->startFrameIdx+10)) {
//compensation for LOUDNESS JUMPS

                if (aListeningCondition==WIDE_H) {
                    hulp1 = (aOriginalLoudness.m_pData[frameIndex]+0.0001) /
(aOriginalLoudness.m_pData[frameIndex-1] + 0.0001);
                    if (hulp1<1.0) hulp1 = 1.0;
                    aDisturbance[i].m_pData[frameIndex] /= pow((hulp1+1.0),0.04);
                    aAddedDisturbance[i].m_pData[frameIndex] *= pow((hulp1+1.0),0.01);
                    hulp = aDistortedLoudness.m_pData[frameIndex];
                    if (hulp<11.0) hulp = 11.0;
                    hulp = pow(hulp,3.0);
                    hulp2 = (aDistortedLoudness.m_pData[frameIndex]*hulp+1.0e-8) /
(aDistortedLoudness.m_pData[frameIndex-1]*hulp + 1.0e-8);
                    if (hulp2<1.0) hulp2 = 1.0;
                    aAddedDisturbance[i].m_pData[frameIndex] /= pow((hulp2+1.0),0.02);
                    hulp1 = (aOriginalLoudness.m_pData[frameIndex-1]*hulp+1.0e-8) /
(aOriginalLoudness.m_pData[frameIndex]*hulp + 1.0e-8);
                    if (hulp1<1.0) hulp1 = 1.0;
                    aDisturbance[i].m_pData[frameIndex] *= pow((hulp1+1.0),0.03);
                } else {
                    hulp1 = (aOriginalLoudness.m_pData[frameIndex]+0.0001) /
(aOriginalLoudness.m_pData[frameIndex-1] + 0.0001);
                    if (hulp1<1.0) hulp1 = 1.0;
                    aDisturbance[i].m_pData[frameIndex] /= pow((hulp1+1.0),0.01);
                    aAddedDisturbance[i].m_pData[frameIndex] *= pow((hulp1+1.0),0.02);
                    hulp = aDistortedLoudness.m_pData[frameIndex];
                    if (hulp<12.0) hulp = 12.0;
                    hulp = pow(hulp,3.0);
                    hulp2 = (aDistortedLoudness.m_pData[frameIndex]*hulp+1.0e-8) /
(aDistortedLoudness.m_pData[frameIndex-1]*hulp + 1.0e-8);
                    if (hulp2<1.0) hulp2 = 1.0;
                    aAddedDisturbance[i].m_pData[frameIndex] /= pow((hulp2+1.0),0.01);
                    hulp1 = (aOriginalLoudness.m_pData[frameIndex-1]*hulp+0.001) /
(aOriginalLoudness.m_pData[frameIndex]*hulp + 0.001);
                    if (hulp1<1.0) hulp1 = 1.0;
                    aDisturbance[i].m_pData[frameIndex] *= pow((hulp1+1.0),0.04);
                }

                if (frameIndex<statics->stopFrameIdx) {
                    hulp1 = mpPitchVec[frameIndex];
                    hulp2 = mpPitchVecDeg[frameIndex];
                    if (hulp1<40.0) hulp1 = 40.0;
                    if (hulp1>400.0) hulp1 = 400.0;
                    if (hulp2<40.0) hulp2 = 40.0;
                    if (hulp2>400.0) hulp2 = 400.0;
                    hulp = (hulp2+1.0)/(hulp1+1.0);
                    if (hulp>1.0) hulp = 1.0/hulp;

                    if (mpPitchVec[frameIndex] <0.1 && mpPitchVecDeg[frameIndex]>20.0)
{
                        aDisturbance[i].m_pData[frameIndex] /= 1.09;
                        aAddedDisturbance[i].m_pData[frameIndex] /= 1.09;
                    }
                    if (mpPitchVec[frameIndex] >20.0 && mpPitchVecDeg[frameIndex]<0.1)
{
                        aDisturbance[i].m_pData[frameIndex] *= 1.07;
```

```
                        aAddedDisturbance[i].m_pData[frameIndex] *= 1.07;
                    }
                }
            }

        }
    } else {
        for (i = 0; i < NUMBER_OF_POWERS_OVER_FREQ; i++) {
            aDisturbance[i].m_pData[frameIndex] = 0.0;
            aAddedDisturbance[i].m_pData[frameIndex] = 0.0;
        }
    }
    }

    distortedLoudnessTimbrePerFrameNarrowbandAvg /= (numberOfaActiveFreqresponse+0.1);
    distortedLoudnessTimbrePerFrameNarrowbandAvg000 =
distortedLoudnessTimbrePerFrameNarrowbandAvg;
    if (distortedLoudnessTimbrePerFrameNarrowbandAvg000<250.0)
distortedLoudnessTimbrePerFrameNarrowbandAvg000 = 250.0;
    if (distortedLoudnessTimbrePerFrameNarrowbandAvg000>800.0)
distortedLoudnessTimbrePerFrameNarrowbandAvg000 = 800.0;
    distortedLoudnessTimbrePerFrameNarrowbandAvg000 =
1.0/pow((distortedLoudnessTimbrePerFrameNarrowbandAvg000/500.0),0.01);

    distortedLoudnessTimbreHighPerFrameAvg /= (numberOfSpeechFrames+0.1);
    distortedLoudnessTimbreHighPerFrameAvgSilent /= (numberOfSilentFrames+0.1);

    distortedLoudnessTimbreHighPerFrameAvgActive /= (numberOfActiveFrames+0.1);
    distortedLoudnessTimbreHighPerFrameAvgActive000 =
distortedLoudnessTimbreHighPerFrameAvgActive;
    if (distortedLoudnessTimbreHighPerFrameAvgActive<15.0)
distortedLoudnessTimbreHighPerFrameAvgActive = 15.0;
    if (distortedLoudnessTimbreHighPerFrameAvgActive>60.0)
distortedLoudnessTimbreHighPerFrameAvgActive = 60.0;
    distortedLoudnessTimbreHighPerFrameAvgActive =
1/distortedLoudnessTimbreHighPerFrameAvgActive;
    distortedLoudnessTimbreHighPerFrameAvgActive *= 2.5;

    distortedLoudnessTimbreHighPerFrameAvgActive000 -= 250.0;
    if (distortedLoudnessTimbreHighPerFrameAvgActive000>80.0)
distortedLoudnessTimbreHighPerFrameAvgActive000 = 80.0;
    distortedLoudnessTimbreHighPerFrameAvgActive000 /= 3000.0;
    distortedLoudnessTimbreHighPerFrameAvgActive000 +=
distortedLoudnessTimbreHighPerFrameAvgActive;

    frameCorrelationTimeDisturbanceAvgCompensation000silent /=
(numberOfSilentFrames+0.1);
    frameCorrelationTimeDisturbanceAvgCompensation000silent =
1.01/pow((frameCorrelationTimeDisturbanceAvgCompensation000silent+1.0),0.05);

    frameCorrelationTimeDisturbanceAvgCompensation000 /= (numberOfSpeechFrames+0.1);
    frameCorrelationTimeDisturbanceAvgCompensation000 =
1.0/pow((frameCorrelationTimeDisturbanceAvgCompensation000+1.1),0.05);

    if (aListeningCondition==WIDE_H) {
        distortedLoudnessTimbreHighPerFrameAvgXnoiseIndicatorTimbre =
pow((1.0+distortedLoudnessTimbreHighPerFrameAvg*noiseIndicatorTimbre),0.03);
    } else {
        distortedLoudnessTimbreHighPerFrameAvgXnoiseIndicatorTimbre = 1.0;
    }

    avgPitchLoudFrames000 = 0.0;
    count0 = 0;
    avgPitchLoudFramesRise000 = 0.0;
    avgPitchLoudFramesDrop000 = 0.0;
    for (frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++) {
        if (frameIndex>(statics->startFrameIdx+5)){
            if (aOriginalLoudness.m_pData[frameIndex-5]>1.0 &&
aOriginalLoudness.m_pData[frameIndex-4]>1.0 &&
            aOriginalLoudness.m_pData[frameIndex-3]>1.0 &&
aOriginalLoudness.m_pData[frameIndex-2]>1.0 &&
            aOriginalLoudness.m_pData[frameIndex-1]>1.0 &&
aOriginalLoudness.m_pData[frameIndex]>1.0 &&
            mpPitchVec[frameIndex-5]>30.0 && mpPitchVec[frameIndex-4]>30.0 &&
mpPitchVec[frameIndex-3]>30.0 &&
```

```
            mpPitchVec[frameIndex-2]>30.0 && mpPitchVec[frameIndex-1]>30.0 &&
mpPitchVec[frameIndex]>30.0 ) {
                avgPitchLoudFrames000 += mpPitchVec[frameIndex];
                count0++;
                hulp1 = (mpPitchVec[frameIndex-5] + mpPitchVec[frameIndex-4] +
mpPitchVec[frameIndex-3])/3.0;
                hulp2 = (mpPitchVec[frameIndex-2] + mpPitchVec[frameIndex-1] +
mpPitchVec[frameIndex])/3.0;
                if (hulp2>hulp1) {
                    avgPitchLoudFramesRise000 += (hulp2 - hulp1);
                } else {
                    avgPitchLoudFramesDrop000 += (hulp1 - hulp2);
                }
            }
        }

        for (i = 0; i < NUMBER_OF_POWERS_OVER_FREQ; i++) {

            if (frameIndex > 10 && aActiveFreqresponse.m_pData[frameIndex]) {

                if (aListeningCondition==WIDE_H) {

                    aAddedDisturbance[i].m_pData[frameIndex] *=
(distortedLoudnessTimbreHighPerFrameAvgXnoiseIndicatorTimbre/Sample
RateRatioCompensation5);

                } else {
                    aAddedDisturbance[i].m_pData[frameIndex] *=
distortedLoudnessTimbreHighPerFrameAvgXnoiseIndicatorTimbre;
                }

            }
        }
    }
    avgPitchLoudFrames000 /= (count0+0.1);
    avgPitchLoudFramesCompensation = pow((avgPitchLoudFrames000+1.0),0.006)/1.01;
    avgPitchLoudFramesRise000 /= (count0+0.1);
    avgPitchLoudFramesDrop000 /= (count0+0.1);
    if (avgPitchLoudFrames000<50.0) avgPitchLoudFrames000 = 50.0;
    if (avgPitchLoudFrames000>170.0) avgPitchLoudFrames000 = 170.0;
    if (avgPitchLoudFramesRise000<1.0) avgPitchLoudFramesRise000 = 1.0;
    if (avgPitchLoudFramesDrop000<1.0) avgPitchLoudFramesDrop000 = 1.0;
    if (avgPitchLoudFramesRise000>20.0) avgPitchLoudFramesRise000 = 20.0;
    if (avgPitchLoudFramesDrop000>20.0) avgPitchLoudFramesDrop000 = 20.0;

//END POLQAMAIN PART 2

    hulp1 = 0.0;
    hulp2 = 0.0;
    for (frameIndex = (statics->startFrameIdx+5); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
        if (UseThisFrame[frameIndex]) {
            if (aActiveFreqresponse.m_pData[frameIndex]) {
                hulp1 += originalPitchPowerDensity. Total ((frameIndex), 50.0, 3500.0);
                hulp2 += distortedPitchPowerDensity. Total ((frameIndex), 50.0,
3500.0);
            }
        }
    }
    hulp1 /= (numberOfaActiveFreqresponse+1.0);
    hulp2 /= (numberOfaActiveFreqresponse+1.0);
    scale = 1.0e9/(hulp1+1.0);
    for (frameIndex = (statics->startFrameIdx+5); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
        if (UseThisFrame[frameIndex]) {
            originalPitchPowerDensity. MultiplyWith (frameIndex, scale);
        }
    }
    scale = 1.0e9/(hulp2+1.0);
    for (frameIndex = (statics->startFrameIdx+5); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
        if (UseThisFrame[frameIndex]) {
            distortedPitchPowerDensity. MultiplyWith (frameIndex, scale);
        }
    }
```

```
    count = 0;
    scaleDistortion = 0.0;
    scaleDistortion2 = 0.0;
    scaleDistortion3 = 0.0;
    scaleDistortion4 = 0.0;
    oldOldScale = 1.0;
    oldScale = 1.0;
    for (frameIndex = (statics->startFrameIdx+22); frameIndex <=
(statics->stopFrameIdx); frameIndex++)     {
        if (UseThisFrame[frameIndex]) {
            aOriginalTotalPower.m_pData[frameIndex] = originalPitchPowerDensity. Total
((frameIndex), 50.0, 3500.0);
            aDistortedTotalPower.m_pData[frameIndex] = distortedPitchPowerDensity.
Total ((frameIndex), 50.0, 3500.0);
            scale = 1.0;
            count1 = 0;

            for (i = -10; i < 0; i++) {
                if ( (aOriginalTotalPower.m_pData[frameIndex+i]>1.0e7) ) {
                    count1++;
                    hulp1 = originalPitchPowerDensity. Total ((frameIndex+i), 50.0,
3500.0);
                    hulp2 = distortedPitchPowerDensity. Total ((frameIndex+i), 50.0,
3500.0);
                    scale *= (hulp2 + (XFLOAT) 1.0e7) / (hulp1 + (XFLOAT) 1.0e7) ;
                }
            }

            scale = pow(scale,1.0/(count1+1));
            if (scale > 10.0) scale = 10.0;
            if (scale < 0.1) scale = 0.1;
            if ( (aOriginalTotalPower.m_pData[frameIndex]>1.0e7) &&
(aOriginalTotalPower.m_pData[frameIndex-1]>1.0e7) &&
(aOriginalTotalPower.m_pData[frameIndex-2]>1.0e7)
            && (aOriginalTotalPower.m_pData[frameIndex-3]>1.0e7) &&
(aOriginalTotalPower.m_pData[frameIndex-4]>1.0e7) &&
(aOriginalTotalPower.m_pData[frameIndex-5]>1.0e7)
            && (aOriginalTotalPower.m_pData[frameIndex-6]>1.0e7) &&
(aOriginalTotalPower.m_pData[frameIndex-7]>1.0e7) &&
(aOriginalTotalPower.m_pData[frameIndex-8]>1.0e7)
            && (aDistortedTotalPower.m_pData[frameIndex]>1.0e7) &&
(aDistortedTotalPower.m_pData[frameIndex-1]>1.0e7) &&
(aDistortedTotalPower.m_pData[frameIndex-2]>1.0e7)
            && (aDistortedTotalPower.m_pData[frameIndex-3]>1.0e7) &&
(aDistortedTotalPower.m_pData[frameIndex-4]>1.0e7) &&
(aDistortedTotalPower.m_pData[frameIndex-5]>1.0e7)
            && (aDistortedTotalPower.m_pData[frameIndex-6]>1.0e7) &&
(aDistortedTotalPower.m_pData[frameIndex-7]>1.0e7) &&
(aDistortedTotalPower.m_pData[frameIndex-8]>1.0e7) )
            {
                hulp = aDisturbance[2].m_pData[frameIndex]-10.0;
                if (hulp<0.0) hulp=0.0;
                hulp = pow(hulp, 4);
                hulp3 = fabs(oldOldScale - scale)/(hulp+1.0e-3)-200.0;
                if (hulp3<1.0) hulp3 = 1.0;
                scaleDistortion += hulp3;

                hulp = aDisturbance[2].m_pData[frameIndex]-10.0;
                if (hulp<0.0) hulp=0.0;
                hulp = pow(hulp, 3);
                hulp3 = fabs(oldOldScale - scale)/(hulp+1.0e-3)-200.0;
                if (hulp3<1.0) hulp3 = 1.0;
                scaleDistortion2 += hulp3;

                hulp = aDisturbance[2].m_pData[frameIndex]-10.0;
                if (hulp<0.0) hulp=0.0;
                hulp = pow(hulp, 4);
                hulp3 = fabs(oldOldScale - scale)/(hulp+1.0e-3)-300.0;
                if (hulp3<1.0) hulp3 = 1.0;
                scaleDistortion3 += hulp3;

                hulp = aDisturbance[2].m_pData[frameIndex]-10.0;
                if (hulp<0.0) hulp=0.0;
                hulp = pow(hulp, 4);
                hulp3 = fabs(oldOldScale - scale)/(hulp+1.0e-4)-200.0;
                if (hulp3<1.0) hulp3 = 1.0;
```

```
                scaleDistortion4 += hulp3;

                count++;

                oldOldScale = oldScale;
                oldScale = scale;
            }
        }
    }
    scaleDistortion /= (count+0.1);
    scaleDistortion2 /= (count+0.1);
    scaleDistortion3 /= (count+0.1);
    scaleDistortion4 /= (count+0.1);

    scaleDistortion -= 30.0;
    if (scaleDistortion<1.0)  scaleDistortion = 1.0;

    scaleDistortion2 -= 30.0;
    if (scaleDistortion2<1.0) scaleDistortion2 = 1.0;

    scaleDistortion3 -= 30.0;
    if (scaleDistortion3<1.0)  scaleDistortion3 = 1.0;

    scaleDistortion4 -= 35.0;
    if (scaleDistortion4<1.0) scaleDistortion4 = 1.0;

    s. Format ("SCALEDISTORTION SCALEDISTORTION SCALEDISTORTION sd=%f sd2=%f sd3=%f
sd4=%f count=%d \n", scaleDistortion,scaleDistortion2, scaleDistortion3,
scaleDistortion4, count );
    gLogFile. WriteString (s);

    hulpDelayMem = 0.0;
    XFLOAT freqShiftChanges;
    for (frameIndex = (statics->startFrameIdx+6); frameIndex <=
(statics->stopFrameIdx-5); frameIndex++) {
        hulp0 = ( (XFLOAT)
(abs(pOverviewHolder->m_DelayPerFrame[frameIndex+1]-pOverviewHolder->m_DelayPer
Frame[frameIndex]))*1000.0/ (XFLOAT) pOverviewHolder->m_SampleFrequencyHz)-1.0;
        if (hulp0<0.0) hulp0 = 0.0;
        if (hulp0>8.0) hulp0 = 8.0;
        hulp1 = ( (XFLOAT)
(abs(pOverviewHolder->m_DelayPerFrame[frameIndex]-pOverviewHolder->m_DelayPerFr
ame[frameIndex-1]))*1000.0/ (XFLOAT) pOverviewHolder->m_SampleFrequencyHz)-1.0;
        if (hulp1<0.0) hulp1 = 0.0;
        if (hulp1>8.0) hulp1 = 8.0;
        hulp2 = ( (XFLOAT)
(abs(pOverviewHolder->m_DelayPerFrame[frameIndex-1]-pOverviewHolder->m_DelayPer
Frame[frameIndex-2]))*1000.0/ (XFLOAT)
pOverviewHolder->m_SampleFrequencyHz)-1.0;
        if (hulp2<0.0) hulp2 = 0.0;
        if (hulp2>8.0) hulp2 = 8.0;
        hulp3 = ( (XFLOAT)
(abs(pOverviewHolder->m_DelayPerFrame[frameIndex-2]-pOverviewHolder->m_DelayPer
Frame[frameIndex-3]))*1000.0/ (XFLOAT)
pOverviewHolder->m_SampleFrequencyHz)-1.0;
        if (hulp3<0.0) hulp3 = 0.0;
        if (hulp3>8.0) hulp3 = 8.0;
        hulp4 = ( (XFLOAT)
(abs(pOverviewHolder->m_DelayPerFrame[frameIndex-3]-pOverviewHolder->m_DelayPer
Frame[frameIndex-4]))*1000.0/ (XFLOAT)
pOverviewHolder->m_SampleFrequencyHz)-1.0;
        if (hulp4<0.0) hulp4 = 0.0;
        if (hulp4>8.0) hulp4 = 8.0;
        hulpDelay = hulp0 + hulp1 + hulp2 + hulp3 + hulp4;
        if (hulpDelay<0.0) hulpDelay = 0.0;

        if (aListeningCondition==WIDE_H) {
            if (hulpDelay>(9.0/SampleRateRatioCompensation2)) hulpDelay =
(9.0/SampleRateRatioCompensation2);
            hulpDelay *= (XFLOAT)0.028*(pow(aDistortedLoudness.m_pData[frameIndex-2],
(XFLOAT)1.5)+(XFLOAT)0.001);
            hulpDelay *= (XFLOAT)0.028*(pow(aDistortedLoudness.m_pData[frameIndex-1],
(XFLOAT)1.5)+(XFLOAT)0.001);
            hulpDelay *= (XFLOAT)0.028*(pow(aDistortedLoudness.m_pData[frameIndex],
(XFLOAT)1.5)+(XFLOAT)0.001);
            hulpDelay *= (XFLOAT)0.028*(pow(aDistortedLoudness.m_pData[frameIndex+1],
```

```
(XFLOAT)1.5)+(XFLOAT)0.001);
        } else {
            if (hulpDelay>9.0) hulpDelay = 9.0;
            hulpDelay *= (XFLOAT)0.028*(pow(aDistortedLoudness.m_pData[frameIndex-3],
(XFLOAT)1.5)+(XFLOAT)0.001);
            hulpDelay *= (XFLOAT)0.028*(pow(aDistortedLoudness.m_pData[frameIndex-2],
(XFLOAT)1.5)+(XFLOAT)0.001);
            hulpDelay *= (XFLOAT)0.028*(pow(aDistortedLoudness.m_pData[frameIndex-1],
(XFLOAT)1.5)+(XFLOAT)0.001);
            hulpDelay *= (XFLOAT)0.028*(pow(aDistortedLoudness.m_pData[frameIndex],
(XFLOAT)1.5)+(XFLOAT)0.001);
            hulpDelay *= (XFLOAT)0.028*(pow(aDistortedLoudness.m_pData[frameIndex+1],
(XFLOAT)1.5)+(XFLOAT)0.001);
        }

        hulpDelayMem = hulpDelay;

        freqShiftChanges = 0.0;

        for (hulpCount = -1; hulpCount < 1; hulpCount++) freqShiftChanges +=
1.0*abs(bestSpectrumShift[frameIndex+hulpCount]-bestSpectrumShift[frameIndex+hu
lpCount-1]);

        for (i = 0; i < NUMBER_OF_POWERS_OVER_FREQ; i++) {

            for (hulpCount = -5; hulpCount < 6; hulpCount++) {
                if (aListeningCondition==WIDE_H) {
                    aDisturbance[i].m_pData[frameIndex+hulpCount]      -=
0.0001*hulpDelay;
                    aAddedDisturbance[i].m_pData[frameIndex+hulpCount] -=
0.0001*hulpDelay;
                    if (aDisturbance[i].m_pData[frameIndex+hulpCount]<0.0)
aDisturbance[i].m_pData[frameIndex+hulpCount] = 0.0;
                    if (aAddedDisturbance[i].m_pData[frameIndex+hulpCount]<0.0)
aAddedDisturbance[i].m_pData[frameIndex+hulpCount] = 0.0;
                } else {
                    aDisturbance[i].m_pData[frameIndex+hulpCount]      -=
0.0002*hulpDelay;
                    aAddedDisturbance[i].m_pData[frameIndex+hulpCount] -=
0.0002*hulpDelay;
                    if (aDisturbance[i].m_pData[frameIndex+hulpCount]<0.0)
aDisturbance[i].m_pData[frameIndex+hulpCount] = 0.0;
                    if (aAddedDisturbance[i].m_pData[frameIndex+hulpCount]<0.0)
aAddedDisturbance[i].m_pData[frameIndex+hulpCount] = 0.0;
                }
            }

            if (aListeningCondition==WIDE_H) {
                aDisturbance[i].m_pData[frameIndex-5] /= pow((freqShiftChanges+1.0),
0.07*SampleRateRatioCompensation);
                aDisturbance[i].m_pData[frameIndex-4] /= pow((freqShiftChanges+1.0),
0.12*SampleRateRatioCompensation);
                aDisturbance[i].m_pData[frameIndex-3] /= pow((freqShiftChanges+1.0),
0.15*SampleRateRatioCompensation);
                aDisturbance[i].m_pData[frameIndex-2] /= pow((freqShiftChanges+1.0),
0.18*SampleRateRatioCompensation);
            } else {
                aDisturbance[i].m_pData[frameIndex-5] /= pow((freqShiftChanges+1.0),
0.1*SampleRateRatioCompensation);
                aDisturbance[i].m_pData[frameIndex-4] /= pow((freqShiftChanges+1.0),
0.15*SampleRateRatioCompensation);
                aDisturbance[i].m_pData[frameIndex-3] /= pow((freqShiftChanges+1.0),
0.18*SampleRateRatioCompensation);
                aDisturbance[i].m_pData[frameIndex-2] /= pow((freqShiftChanges+1.0),
0.22*SampleRateRatioCompensation);
            }

            if (aListeningCondition==WIDE_H) {
            } else {
                aAddedDisturbance[i].m_pData[frameIndex] *=
(0.9*pow((1.0+correlationOriginalWithDisturbance),0.25));
            }

        }

    }
```

```
number_of_sections_inserted = 0;
number_of_sections_critical = 0;
number_of_sections_invalid = 0;
    for (frameIndex = statics->startFrameIdx; frameIndex < (statics->stopFrameIdx - 1);
frameIndex++) {
        delayReliabilityPerFrameWeight =
(pOverviewHolder->m_DelayReliabilityPerFrame[frameIndex] +
pOverviewHolder->m_DelayReliabilityPerFrame[frameIndex+1])/2.0;
        if (delayReliabilityPerFrameWeight<0.03) delayReliabilityPerFrameWeight = 0.03;
        if (delayReliabilityPerFrameWeight>0.9) delayReliabilityPerFrameWeight = 0.9;
        delayReliabilityPerFrameWeight /= 0.9;
        delayReliabilityPerFrameWeight = pow(delayReliabilityPerFrameWeight,0.04);
        hulp1 = 1.0;
        hulp2 = 1.0;
        if (pMarkSectionFlags[frameIndex] & 4)     {
            number_of_sections_invalid++;
        } else if (pMarkSectionFlags[frameIndex] & 2)    {
               number_of_sections_critical++;
               hulp2 = pow((1.0*pMarkSectionFlags[frameIndex])+0.1,0.4);
            } else if (pMarkSectionFlags[frameIndex] & 1)    {
               number_of_sections_inserted++;
               hulp1 = pow((1.0*pMarkSectionFlags[frameIndex])+0.1,0.3);
            }

        for (i = 0; i < NUMBER_OF_POWERS_OVER_FREQ; i++) {
            if (aListeningCondition==WIDE_H) {
                aAddedDisturbance[i].m_pData[frameIndex] /=
delayReliabilityPerFrameWeight;
            } else {
            }
        }

    }
    fraction_of_sections_inserted =
number_of_sections_inserted/(numberOfSpeechFrames+0.01);
    fraction_of_sections_critical =
number_of_sections_critical/(numberOfSpeechFrames+0.01);
    fraction_of_sections_invalid =
number_of_sections_invalid/(numberOfSpeechFrames+0.01);

    if (statics->nrSpeechFrames > 0)
    {
        averageScale = (XFLOAT) pow ( (averageScale/statics->nrSpeechFrames) , (XFLOAT)
5e-3);
    }else{
        averageScale = (XFLOAT) 1.0;
    }

    int NumFrames = statics->nrFrames;
    pOverviewHolder->m_NumberOfFrames = NumFrames;
    pOverviewHolder->m_aTransformLength = aTransformLength;
    pOverviewHolder->m_SampleFrequencyHz = (long)statics->sampleRate;
    pOverviewHolder->m_NumberOfBands=statics->aNumberOfBarkBands;

    pOverviewHolder->m_ResultFlags |= (RESF_LEVEL1_AVAILABLE);

    ShowProgress (10, "Copying graph data");

    if (gBatchMode)
    {
        s. Format ("PSQM command completed succesfully!\n");

        return TRUE;
    }

    if (Mode & RESF_LEVEL3_AVAILABLE)
    {
        pOverviewHolder->m_pDisturbance = (double*)matMalloc(NumFrames *
sizeof(double));
        pOverviewHolder->m_pAddedDisturbance = (double*)matMalloc(NumFrames *
sizeof(double));
        pOverviewHolder->m_pDistortedLoudness = (double*)matMalloc(NumFrames *
sizeof(double));
        pOverviewHolder->m_pOriginalLoudness = (double*)matMalloc(NumFrames *
sizeof(double));
```

```
        pOverviewHolder->m_pTime = (double*)matMalloc(NumFrames * sizeof(double));

        pOverviewHolder->m_StartFrameIndex = statics->startFrameIdx;
        pOverviewHolder->m_StopFrameIndex = statics->stopFrameIdx;
        for(frameIndex = statics->startFrameIdx; frameIndex <= statics->stopFrameIdx;
frameIndex++)
        {
            pOverviewHolder->m_pDisturbance[frameIndex] = aDisturbance[(3 -
MINIMUM_POWER_FREQ) / STEP_POWER_FREQ].m_pData[frameIndex];
            pOverviewHolder->m_pAddedDisturbance[frameIndex] =
aAddedDisturbance[0].m_pData[frameIndex];
            pOverviewHolder->m_pDistortedLoudness[frameIndex] =
aDistortedLoudness.m_pData[frameIndex];
            pOverviewHolder->m_pOriginalLoudness[frameIndex] =
aOriginalLoudness.m_pData[frameIndex];
            pOverviewHolder->m_pTime[frameIndex] = frameIndex * (aTransformLength / 2)
/ statics->sampleRate;
        }

        CreateArrayFromCSignal(&pOverviewHolder->m_pOriginalHzPowerSpectrum,
&originalHzPowerSpectrum);
        CreateArrayFromCSignal(&pOverviewHolder->m_pDistortedHzPowerSpectrum,
&distortedHzPowerSpectrum);
        CreateArrayFromCSignal(&pOverviewHolder->m_pOriginalPitchPowerDensity,
&originalPitchPowerDensity);
        CreateArrayFromCSignal(&pOverviewHolder->m_pDistortedPitchPowerDensity,
&distortedPitchPowerDensity);
        CreateArrayFromCSignal(&pOverviewHolder->m_pOriginalLoudnessDensity,
&originalLoudnessDensity);
        CreateArrayFromCSignal(&pOverviewHolder->m_pDistortedLoudnessDensity,
&distortedLoudnessDensity);
        CreateArrayFromCSignal(&pOverviewHolder->m_pDisturbanceDensity,
&disturbanceDensity);

        pOverviewHolder->m_ResultFlags |= RESF_LEVEL3_AVAILABLE;
    }
    else
    {
        pOverviewHolder->m_StartFrameIndex=0;
        pOverviewHolder->m_StopFrameIndex=0;

        pOverviewHolder->m_pDisturbance = 0;
        pOverviewHolder->m_pAddedDisturbance = 0;
        pOverviewHolder->m_pDistortedLoudness = 0;
        pOverviewHolder->m_pOriginalLoudness = 0;
        pOverviewHolder->m_pTime = 0;

        pOverviewHolder->m_SizeofAlignedOriginaTimeSeries=0;
        pOverviewHolder->m_AlignedOriginalTimeSeries=0;

        pOverviewHolder->m_SizeofAlignedDistortedTimeSeries=0;
        pOverviewHolder->m_AlignedDistortedTimeSeries=0;

    }

    if (UseThisFrame) delete[] UseThisFrame;
    if (FrameFlags) delete[] FrameFlags;
    if (bestSpectrumShift)
        matFree(bestSpectrumShift);

    if (bestWarpingFacPerFrame)
        matFree(bestWarpingFacPerFrame);

    return TRUE;
}
}
```