```
    typedef double XFLOAT;
    typedef double OTA_FLOAT;

namespace POLQAV2
{

BOOL stringEndsWithWav (const CNewStdString &s) {
    int n;

    n = s. GetLength ();
    if (n < 3) {
        return FALSE;
    }

    if ((s [n-1] == 'V') || (s [n-1] == 'v')) {
        if ((s [n-2] == 'A')|| (s [n-2] == 'a')) {
            if ((s [n-3] == 'W') || (s [n-3] == 'w')) {
               return TRUE;
            }
        }
    }
    return FALSE;
}

CTimeSeries::CTimeSeries()
{
    aInitialized = FALSE;

    aName = "";

    aHeaderDelayInBytes = 0;
    aTrailerDelayInBytes = 0;
    aStereoInFile = FALSE;
}

BOOL CTimeSeries::Initialize(CNewStdString pName, CPOLQAData *polqaHandle)
{
    aName = pName;

    if (aInitialized) {
        if (!gBatchMode) {
        }
        return FALSE;
    }

    this->POLQAHandle = polqaHandle;

    statics = polqaHandle->statics;

    ASSERT(polqaHandle->statics->nrTimesSamples > 0);

    SetSize(polqaHandle->statics->nrTimesSamples);
    aInitialized = TRUE;

    SetToConstant(0.);

    return true;
}

void CTimeSeries::SetToConstant (XFLOAT pValue)
{
    matbSet(pValue, this->m_pData, statics->nrTimesSamples);
}

void CDoubleArray::RatioOf (const CDoubleArray &pNominator, const CDoubleArray
&pDenominator, XFLOAT pFuzz)
{
    int i, range;
    int n = GetSize ();

    for (i = 0; i < n; i++) {
        int j;
        XFLOAT totalWeight = 0;
        range = 30;
        for (j = -range; j <= range; j++) {
            if ((i + j >= 0) && (i + j < n)) {
                XFLOAT weight = 1.0 / (1.0 + 0.2*(XFLOAT)abs(j));
                this->m_pData[i+j] += weight * (pNominator.m_pData[i+j] + pFuzz) /
(pDenominator.m_pData[i+j] + pFuzz);
                totalWeight += weight;
            }
        }
        this->m_pData[i] /= totalWeight;
    }
}

void CDoubleArray::CompressOf (const CDoubleArray &pThat, XFLOAT pConstant)
{
    int i;
    int n = GetSize ();

    for (i = 0; i < n; i++) {
        this->m_pData[i] = pow (pThat.m_pData[i], pConstant);
    }
}

XFLOAT CDoubleArray::Power (int pStartIndex, int pStopIndex) const
{
    int    i;
    XFLOAT  power;
    CNewStdString s;

    power = 0;

    if (pStartIndex < 0) {
        if (!gBatchMode) {
        } else {
            s. Format ("TimeSeries. Power : start index negative! " + aName);
            gLogFile. WriteString (s);
        }
        exit (1);
    }

    if (pStartIndex > pStopIndex) {
        if (!gBatchMode) {
        } else {
            s. Format ("TimeSeries. Power : stop index exceeds start index!\n" + aName);
            gLogFile. WriteString (s);
        }
        exit (1);
    }

    int n = GetSize ();

    if (pStopIndex > n) {
        if (!gBatchMode) {
        } else {
            s. Format ("TimeSeries. Power : stop index exceeds length!\n" + aName);
            gLogFile. WriteString (s);
        }

        exit (1);
    }

    for (i = pStartIndex; i < pStopIndex; i++) {
        XFLOAT h = this->m_pData[i];
        power += h * h;
    }

    power /= (pStopIndex - pStartIndex);
    return power;
}

XFLOAT CDoubleArray::PowerInBand(CPOLQAData *POLQAHandle, XFLOAT pLowerFrequency, XFLOAT
pUpperFrequency) const
{
    XFLOAT result = 0;

    const XFLOAT frequencyResolutionHz = POLQAHandle->statics->aFrequencyResolutionHz;
    for(int bandIndex = 0; bandIndex < GetSize(); bandIndex++)
    {
        const XFLOAT frequency = bandIndex * frequencyResolutionHz;
        if((frequency >= pLowerFrequency) && (frequency <= pUpperFrequency))
        {
            result += this->m_pData[bandIndex];
        }
    }

    return result;
}

void CDoubleArray::InvDb2 (CPOLQAData *POLQAHandle,
                           XFLOAT    pBasisDb [][2],
                           int       pNumberOfPointsBasis,
                           XFLOAT    pDeltaDb [][2],
                           int       pNumberOfPointsDelta)
{
    XFLOAT  centreOfBandHz;
    XFLOAT  gainDb;
    XFLOAT  gain;

    const XFLOAT freqRes = POLQAHandle->statics->aFrequencyResolutionHz;
    const int size = GetSize();
    for(int bandIndex = 0; bandIndex < size; bandIndex++)
    {
        centreOfBandHz = freqRes * (XFLOAT) bandIndex;
        gainDb = interpolate (centreOfBandHz, pBasisDb, pNumberOfPointsBasis);
        gainDb += interpolate (centreOfBandHz, pDeltaDb, pNumberOfPointsDelta);
        gain = pow (10.0, gainDb / 10.0);

        this->m_pData[bandIndex] = gain;
    }
}

void CDoubleArray::TimeAvgOf(const CPOLQAData *POLQAHandle, const CHzSpectrum &pThat)
{
    XFLOAT result;
    int    count;

    const int stopFrameIdx = POLQAHandle->statics->stopFrameIdx;
    const int aNumberOfHzBands = POLQAHandle->statics->aNumberOfHzBands;
    for(int bandIndex = 0; bandIndex < aNumberOfHzBands; bandIndex++)
    {
        result = 0;
        count = 0;
        for(int frameIndex = POLQAHandle->statics->startFrameIdx; frameIndex <=
stopFrameIdx; frameIndex++)
        {
            result += (pThat.m_pData[frameIndex])[bandIndex];
            count++;
        }
        result /= (XFLOAT) count;

        this->m_pData[bandIndex] = result;
    }
}

const char *CTimeSeries::GetName (void) const
{
    return (const char *) aName;
}

void CTimeSeries::operator *= (XFLOAT pFactor)
{
    matbMpy1(pFactor, this->m_pData, statics->nrTimesSamples);
}

void CTimeSeries::operator= (const CTimeSeries &pInputTimeSeries)
{
    matbCopy(pInputTimeSeries.m_pData, this->m_pData, statics->nrTimesSamples);
}

XFLOAT CTimeSeries::Envelope (CPOLQAData *POLQAHandle, int pStartIndex, int frameLength)
const
{
    XFLOAT  envelope;
    SmartBufferPolqa SB(POLQAHandle, frameLength);
    XFLOAT *temp = SB.Buffer;

    int length;
    if(pStartIndex + frameLength - 1 < statics->nrTimesSamples)
        length = frameLength;
    else
        length = statics->nrTimesSamples - pStartIndex;

    matbSqr2(this->m_pData+pStartIndex, temp, length);
    envelope  = matSum(temp, length);

    envelope /= frameLength;
    envelope  = sqrt(envelope);

    return envelope;
}

XFLOAT CTimeSeries::WindowedSample(int pFrameIndex, int pSampleIndex, int pWindowSize)
{
    XFLOAT   result;
    int      i;

    ASSERT ((0 <= pSampleIndex) && (pSampleIndex < pWindowSize));

    i = pFrameIndex * pWindowSize/2 + pSampleIndex;

    if (i < 0) {
        return 0;
    }

    if (i >= this->GetSize()) {
        return 0;
    }

    result = statics->frameWindow[pSampleIndex] * this->m_pData[i];

    return result;
}

BOOL CTimeSeries::ReadFromBuffer (XFLOAT* pSamples, long NumberOfSamples)
{
    matbCopy(pSamples, m_pData, NumberOfSamples);
    aNumberOfSamples = NumberOfSamples;
    return TRUE;
}

void upperCase (char *outputString, const char *inputString) {
    int i, n;

    n = strlen (inputString);
    for (i = 0; i < n; ++i) {
        outputString [i] = (char) toupper (inputString[i]);
    }
}

BOOL CTimeSeries::OpenFile (XFLOAT aSampleFrequencyHz,
                            CNewStdString  pSoundFilePathName,
                            int  &pNumberOfSamples,
                            int    pStereoIfNotWavFile,
                            int    pRightIfStereo)
{
    unsigned int      flen, lengthInBytes, fileLength, sampleFrequency, bytesPerSecond;
    int               n;
    short             numChannels, bitsPerSample;
    short             tag;
    char              riffId[5], formatId [5], dataId[5];
    CNewFile             soundFile;
    CNewStdString           s;

    aHeaderDelayInBytes = 0;
    aTrailerDelayInBytes = 0;
    aRightIfStereo = pRightIfStereo;

    if (!soundFile. Open (pSoundFilePathName, "rb")) {
        if (gBatchMode) {

            exit (1);
        } else {
        }
        return FALSE;
    }

    if (!stringEndsWithWav (pSoundFilePathName)) {
        aStereoInFile = pStereoIfNotWavFile;

        soundFile. SeekToEnd ();

        if (pStereoIfNotWavFile) {
            aNumberOfSamples = soundFile. GetLength () / 4;
        } else {
            aNumberOfSamples = soundFile. GetLength () / 2;
        }
    } else {
        soundFile. Read (riffId, 4);
        riffId [4] = '\0';
        upperCase (riffId, riffId);

        if (0 != strcmp (riffId, "RIFF")) {
            if (!gBatchMode) {
            }

            return FALSE;
        }

        soundFile. Read (&fileLength, 4);
        soundFile. Read (riffId, 4);
        riffId[4] = '\0';
        upperCase (riffId, riffId);

        if (0 != strcmp (riffId, "WAVE")) {
            if (!gBatchMode) {
            }

            return FALSE;
        }

        soundFile. Read (formatId, 4);
        formatId [4] = '\0';

        soundFile. Read (&flen, 4);
        soundFile. Read (&tag, 2);

        if (tag != 1) {
            if (!gBatchMode) {
            }
            return FALSE;
        }

        soundFile. Read (&numChannels, 2);
        switch (numChannels) {
        case 1:
            aStereoInFile = FALSE;
            break;
        case 2:
            aStereoInFile = TRUE;
            break;
        default:
            if (!gBatchMode) {
            }
            return FALSE;
        }

        soundFile. Read (&sampleFrequency, 4);

        if (aSampleFrequencyHz != (XFLOAT)sampleFrequency)
        {

            return FALSE;
        }
        soundFile. Read (&bytesPerSecond, 4);

        soundFile. Read (&tag, 2);

        soundFile. Read (&bitsPerSample, 2);
        if (bitsPerSample != 16) {

            exit (1);
        }

        soundFile. Read (dataId, 4);
        dataId[4] = '\0';
        upperCase (dataId, dataId);
        if (0 != strcmp (dataId, "DATA")) {

            exit (1);
         }

        soundFile. Read (&lengthInBytes, 4);

        aHeaderDelayInBytes = soundFile. GetPosition ();

        n = soundFile. GetLength ();
        aTrailerDelayInBytes = n - aHeaderDelayInBytes - lengthInBytes;

        aNumberOfSamples = n/2 - aHeaderDelayInBytes/2 - aTrailerDelayInBytes/2;

        if (aStereoInFile) {
            aNumberOfSamples /= 2;
        }
    }
    if (!aStereoInFile) {
        aRightIfStereo = FALSE;
    }

    pNumberOfSamples = aNumberOfSamples;

    soundFile. Close ();

    return TRUE;
}

short SwapBytes (short a) {
    short b = (short) ((a & 0xff) << 8);
    short c = (short) ((a & 0xff00) >> 8);
    return (short) (b | c);
}

BOOL CTimeSeries::ReadFromDisk (const CNewStdString &pSoundFilePathName,
                                long                 pNumberOfSamples,
                                int                  pSwapBytes,
                                XFLOAT*              pChecksum)
{
    short               h;
    int                 i;
    short               *buffer;
    CNewFile            soundFile;

    bool couldOpenSoundFile = false;
    int openTrials = 0;
    const int maxOpenTrials = 20;

    couldOpenSoundFile = soundFile. Open (pSoundFilePathName, "rb");
    while(!couldOpenSoundFile && openTrials < maxOpenTrials)
    {
        couldOpenSoundFile = soundFile. Open (pSoundFilePathName, "rb");
        openTrials++;
        Sleep(200);
    }

    if (!couldOpenSoundFile) {
        if (gBatchMode)
        {

            exit (1);
        }
        else
        {
        }
        return FALSE;
    }

    buffer = new short [2 * pNumberOfSamples];

    ASSERT (sizeof(short) == 2);

    if (aStereoInFile)
    {
        soundFile.Seek(aHeaderDelayInBytes, SEEK_SET);

        if (sizeof (short) * 2 * pNumberOfSamples != soundFile. Read (buffer, sizeof
(short) * 2 * pNumberOfSamples))
        {
            if (!gBatchMode) {
            }

            exit (1);
        }

    }
    else
    {
        soundFile.Seek(aHeaderDelayInBytes, SEEK_SET);

        if (sizeof (short) * pNumberOfSamples != soundFile.Read(buffer, sizeof (short) *
pNumberOfSamples)) {
            if (!gBatchMode)
            {
            }

            exit (1);
        }

    }

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

        if (aStereoInFile) {
            if (aRightIfStereo) {
                h = buffer [2*i+1];
            } else {
                h = buffer [2*i];
            }
        } else {
            h = buffer [i];
        }

        if (pSwapBytes) {
            this->m_pData[i] = (XFLOAT) SwapBytes (h);
        } else {
            this->m_pData[i] = (XFLOAT) h;
        }
    }

    delete [] buffer;
    soundFile.Close();

    if (pChecksum)
    {
        XFLOAT sum = 0; for (int i=0; i< pNumberOfSamples; i++) sum+=m_pData[i];
        *pChecksum = sum;
    }

    return TRUE;
}

BOOL MakeStereoFile (FILE* pOutputFile,
                     const CTimeSeries  &pOriginalTimeSeries,
                     const CTimeSeries  &pDistortedTimeSeries,
                     CPOLQAData *POLQAHandle)
{
    int            i;
    int             h;
    short           *buffer;
    CNewLogFile     outputFile(pOutputFile);
    int             n;

    n = POLQAHandle->statics->nrTimesSamples;

    ASSERT (sizeof(short) == 2);

    buffer = new short [2*n];

    for (i = 0; i < n; i++) {
        h = (int) round (pOriginalTimeSeries.m_pData[i]/2.0);
        if (h < -32767) h = -32767;
        if (h > 32767)  h = 32767;
        h = (short) round (h);
        buffer [2*i] = (short) h;
        h = (int) round (pDistortedTimeSeries.m_pData[i]/2.0);
        if (h < -32767) h = -32767;
        if (h > 32767)  h = 32767;
        h = (short) round (h);
        buffer [2*i + 1] = (short) h;
    }

        outputFile.Write (buffer, sizeof (short) * 2 * n);

    outputFile. Close ();
    delete [] buffer;

    return TRUE;
}

void CTimeSeries::SetToSine(XFLOAT pAmplitude, const XFLOAT pOmega)
{
    for(int i = 0; i < statics->nrTimesSamples; i++)
    {
       this->m_pData[i] = sin(pOmega * i);
    }
    matbMpy1(pAmplitude, this->m_pData, statics->nrTimesSamples);
}

void CTimeSeries::FilterWith (CPOLQAData            *POLQAHandle,
                              const CTimeSeries     &pInputTimeSeries,
                              XFLOAT*               pTaps,
                              int                   TapsLength)
{
    int rc;
    rc = matRunFIRFilter(POLQAHandle->mh, pInputTimeSeries.m_pData, this->m_pData,
statics->nrTimesSamples, pTaps, TapsLength, MAT_FIRDelayComp);
    ASSERT(rc == 0);
}

void CTimeSeries::FilterWith (CPOLQAData         *POLQAHandle,
                              BOOL               pInputFFTAvailable,
                              XFLOAT             pSampleFrequencyHz,
                              XFLOAT             pFilterCurve [][2],
                              int                pNumberOfPoints,
                              const CTimeSeries &pInputTimeSeries,
                              CDoubleArray       &pInputFFT,
                              CDoubleArray       &pOutputFFT)
{
    XFLOAT                factorDb, factor;
    XFLOAT                overallGainFilter = interpolate ((XFLOAT) 1000, pFilterCurve,
pNumberOfPoints);
    long                  i, powerOf2 = 1, order = 0;
    XFLOAT                *x;
    XFLOAT                frequencyResolution;

    const int aTimeSeriesLength = statics->nrTimesSamples;

    while (powerOf2 < aTimeSeriesLength)
    {
        powerOf2 *= 2;
        order++;
    }
    SmartBufferPolqa SB_x(POLQAHandle, powerOf2 + 2);
    x = SB_x.Buffer;

    if (!pInputFFTAvailable)
    {
        matbZero(x, powerOf2 + 2);

        matbCopy(pInputTimeSeries.m_pData, x, aTimeSeriesLength);

        for (i = 0; i < aTimeSeriesLength; i++)
        {
            if (i < 100) {
                x [i] *= (XFLOAT) i / (XFLOAT) 100;
            }
            if (aTimeSeriesLength - 1 - i < 100) {
                x [i] *= (XFLOAT) (aTimeSeriesLength - 1 - i) / (XFLOAT) 100;
            }
        }

        matRealFft (POLQAHandle->mh, x, order, MAT_Forw);

        pInputFFT.Initialize("pInputFFT", powerOf2 + 2);
        matbCopy(x, pInputFFT.m_pData, powerOf2 + 2);
    }
    else
    {
        matbCopy(pInputFFT.m_pData, x, powerOf2 + 2);
    }

    frequencyResolution = pSampleFrequencyHz / powerOf2;

    for (i = 0; i <= powerOf2/2; i++) {
        factorDb = interpolate (i * frequencyResolution, pFilterCurve, pNumberOfPoints)
- overallGainFilter;
        factor = pow (10.0, factorDb / 20.0);

        x [2 * i] *= factor;
        x [2 * i + 1] *= factor;
    }

    pOutputFFT.Initialize("pOutputFFT", powerOf2 + 2);
    matbCopy(x, pOutputFFT.m_pData, powerOf2 + 2);

    matCcsFft (POLQAHandle->mh, x, order, MAT_Inv);

    matbCopy(x, this->m_pData, aTimeSeriesLength);
}

inline void SearchForMaxInRange(const XFLOAT *vec, const int startIdx, const int
stopIdx, XFLOAT *curMax, int *curMaxIdx)
{
    const int searchLen = (((0) > (stopIdx - startIdx)) ? (0) : (stopIdx - startIdx));
    if(searchLen)
    {
        int newMaxIdx = 0;
        XFLOAT newMax = 0.0;
        newMax = matMaxExt(vec + startIdx, searchLen, &newMaxIdx);
        newMaxIdx += startIdx;

        if(newMax > *curMax)
        {
            *curMax = newMax;
            *curMaxIdx = newMaxIdx;
        }
    }
}

XFLOAT CTimeSeries::ReverberationIndicator( CPOLQAData          *POLQAHandle,
                                            const XFLOAT        pSampleFrequencyHz,
                                            const CTimeSeries   &pInputTimeSeriesOrg,
                                            const CTimeSeries   &pInputTimeSeriesDis)
{
    const int           aTimeSeriesLength = statics->nrTimesSamples;
    XFLOAT              *varianceETC = 0;
    CNewStdString       s;
```

```
const int order = matFFTOrder(aTimeSeriesLength);
const int powerOf2 = 1<<order;

XFLOAT *x1 = (XFLOAT*)matMalloc((powerOf2 + 2) * sizeof(XFLOAT));
XFLOAT *x2 = (XFLOAT*)matMalloc((powerOf2 + 2) * sizeof(XFLOAT));

XFLOAT levelRef = 0;
XFLOAT levelDeg = 0;
levelRef  = matbNormL2(pInputTimeSeriesOrg.m_pData, aTimeSeriesLength);
levelRef *= levelRef;

levelDeg  = matbNormL2(pInputTimeSeriesDis.m_pData, aTimeSeriesLength);
levelDeg *= levelDeg;

XFLOAT scalingFactor = sqrt((levelRef+1.0)/(levelDeg+1.0));

matbCopy(pInputTimeSeriesOrg.m_pData, x1, aTimeSeriesLength);
matbMpy4(pInputTimeSeriesDis.m_pData, scalingFactor, x2, aTimeSeriesLength);

matbZero(x1 + aTimeSeriesLength, powerOf2 + 2 - aTimeSeriesLength);
matbZero(x2 + aTimeSeriesLength, powerOf2 + 2 - aTimeSeriesLength);

const int hulp2 = (int)floor(pSampleFrequencyHz/(XFLOAT)250.0);

const int windowLength = 2*hulp2 + 1;

for (int i = hulp2; i < (aTimeSeriesLength-hulp2); i++) {

    matbAbs2(x1+i-hulp2, temp, hulpLength);
    hulpIn = matSum(temp, hulpLength)/(hulpLength);
    matbAbs2(x2+i-hulp2, temp, hulpLength);
    hulpOut = matSum(temp, hulpLength)/(hulpLength);

    if ( (hulpOut>3*hulpIn) && (hulpIn > 500.0) )
        x2 [i] *= (3*hulpIn/hulpOut);
    else
    {
        if ( (hulpOut>4*hulpIn) && (hulpIn >200.0) )
            x2 [i] *= (4*hulpIn/hulpOut);
        else
        {
            if ( (hulpOut>5*hulpIn) )
                x2 [i] *= (5*hulpIn/hulpOut);
        }
    }
}

SB_temp.Free();
temp = 0;

levelRef = matbNormL2(x1, aTimeSeriesLength);
levelRef *= levelRef;

levelDeg = matbNormL2(x2, aTimeSeriesLength);
levelDeg *= levelDeg;
scalingFactor = sqrt((levelRef+1.0)/(levelDeg+1.0));

matbMpy1(scalingFactor, x2, aTimeSeriesLength);

matRealFft(POLQAHandle->mh, x1, order, MAT_Forw);
matRealFft(POLQAHandle->mh, x2, order, MAT_Forw);

SmartBufferPolqa SB_H_ETC(POLQAHandle, powerOf2 + 2);
XFLOAT *H = SB_H_ETC.Buffer;

const XFLOAT samplesPerHz = (XFLOAT)powerOf2/pSampleFrequencyHz;

matbZero(powerSpectrum1, powerOf2 + 2);
matbZero(powerSpectrum2, powerOf2 + 2);

reverbIndicator = 0.0;
for (i = 80*samplesPerHz; i < (5000*samplesPerHz) / 2; i++) {
    a1 = x1 [2 * i];
    b1 = x1 [2 * i + 1];
    a2 = x2 [2 * i];
    b2 = x2 [2 * i + 1];
```

```
        powerSpectrum1 [i] = a1 * a1 + b1 * b1;

        powerSpectrum2 [i] = a2 * a2 + b2 * b2;
        H [2 * i] = ((a2 * a1) + (b2 * b1)) / ((a1 * a1) + (b1 * b1));
        H [2 * i + 1] = ((b2 * a1) - (a2 * b1)) / ((a1 * a1) + (b1 * b1));
    }

    matFree(x1); x1 = 0;
    matFree(x2); x2 = 0;

    SmartBufferPolqa SB_varianceETC(POLQAHandle, powerOf2 + 2);
    varianceETC = SB_varianceETC.Buffer;

    matbZero(varianceETC, powerOf2 + 2);

    matCcsFft(POLQAHandle->mh, H, order, MAT_Inv);

    const int lowerSearchRangeETC = (int)floor(pSampleFrequencyHz/15.0);
    const int upperSearchRangeETC = ((((int)(pSampleFrequencyHz*2)) < (powerOf2)) ?
((int)(pSampleFrequencyHz*2)) : (powerOf2));

    XFLOAT avgETCtail  = 0.0;

    XFLOAT *ETC = H;
    H = 0;

    matbAbs1(ETC + lowerSearchRangeETC, powerOf2/2 - lowerSearchRangeETC);
    matbSqrt1(ETC + lowerSearchRangeETC, powerOf2/2 - lowerSearchRangeETC);

    const int L10start = (((lowerSearchRangeETC) > (powerOf2/8 + 1)) ?
(lowerSearchRangeETC) : (powerOf2/8 + 1));

    const int L10stop = powerOf2/2;
    const int L10len = L10stop - L10start;

    SmartBufferPolqa SB_L10(POLQAHandle, L10len);
    XFLOAT *L10 = SB_L10.Buffer;

    matbPow2(ETC + L10start, 10.0, L10, L10len);
    avgETCtail = matSum(L10, L10len);

    SB_L10.Free();
    L10 = 0;

    avgETCtail /= (powerOf2/5.0);
    avgETCtail = pow(avgETCtail, 0.1);
    matbAdd1(-1.9*avgETCtail, ETC +lowerSearchRangeETC, upperSearchRangeETC
-lowerSearchRangeETC);
    matbThresh1(ETC +lowerSearchRangeETC, upperSearchRangeETC -lowerSearchRangeETC, 0.0,
MAT_LT);
    matbThresh1(ETC +lowerSearchRangeETC, upperSearchRangeETC -lowerSearchRangeETC,
0.015, MAT_GT);

    XFLOAT maxETCtemp = 0;
    int maxNumbertemp = 0;

    int maxNumber1 = 0;
    XFLOAT max1ETC = 0.0;

    for (i = lowerSearchRangeETC; i < powerOf2/2; i++) {
        if ( ETC [i]>max1ETC && i < (pSampleFrequencyHz*2.0) )
        {
            max1ETC = ETC [i];
            maxNumber1 = i;
        }
    }

    maxNumber1 -= lowerSearchRangeETC;
    if (maxNumber1 <0) maxNumber1 = 0;

    const int hulp3 = (int)floor(pSampleFrequencyHz/10.0);
    int maxNumber2 = 0;
    XFLOAT max2ETC = 0.0;

    for (i = lowerSearchRangeETC; i < (maxNumber1-hulp3); i++) {
        if ( ETC [i]>max2ETC && i < (pSampleFrequencyHz*2.0) ) {
            max2ETC = ETC [i];
            maxNumber2 = i;
        }
    }
    for (i = (maxNumber1+hulp3); i < powerOf2/2; i++) {
        if ( ETC [i]>max2ETC && i < (pSampleFrequencyHz*2.0) ) {
            max2ETC = ETC [i];
            maxNumber2 = i;
        }
    }

    maxNumber2 -= lowerSearchRangeETC;
    if (maxNumber2 <0) maxNumber2 = 0;

    int hulpNumber1 = maxNumber1;
    int hulpNumber2 = maxNumber2;
    if(hulpNumber2 < hulpNumber1)
    {
        int hulp = hulpNumber2;
        hulpNumber2 = hulpNumber1;
        hulpNumber1 = hulp;
    }

    int maxNumber3 = 0;
    XFLOAT max3ETC = 0.0;

    for (i = lowerSearchRangeETC; i < (hulpNumber1-hulp3); i++) {
        if ( ETC [i]>max3ETC && i < (pSampleFrequencyHz*2.0) ) {
            max3ETC = ETC [i];
            maxNumber3 = i;
        }
    }
    for (i = (hulpNumber1+hulp3); i < (hulpNumber2-hulp2); i++) {
        if ( ETC [i]>max3ETC && i < (pSampleFrequencyHz*2.0) ) {
            max3ETC = ETC [i];
            maxNumber3 = i;
        }
    }
    for (i = (hulpNumber2+hulp3); i < powerOf2/2; i++) {
        if ( ETC [i]>max3ETC && i < (pSampleFrequencyHz*2.0) ) {
            max3ETC = ETC [i];
            maxNumber3 = i;
        }
    }

    maxNumber3 -= lowerSearchRangeETC;
    if (maxNumber3 <0)  maxNumber3 = 0;

    avgVarETC = 0.0;
    hulpCount = 0;
    for (i = 200; i < powerOf2 + 2 - 200; i += 50) {
        ETCWindowed = 0.0;
        for (j = i - 200 + 1; j < i + 200; j++) {
            ETCWindowed += ETC [j] * exp (-0.5*((-(i*1.0) +
(j*1.0))/(0.4*200))*((-(i*1.0) + (j*1.0))/(0.4*200)));
        }
        if (((ETCWindowed / 200) - avgETC) > 0.0) varianceETC [(i - 200) / 50] +=
pow(((ETCWindowed / 200) - avgETC), 2);
        else varianceETC [(i - 200) / 50] = 0;
        if ( (i>600) && (varianceETC [(i - 200) / 50]>0.001) ) {
            avgVarETC += varianceETC [(i - 200) / 50];
            hulpCount += 1;
        }
    }

    const XFLOAT normFactor = 8000.f/pSampleFrequencyHz;
    XFLOAT reverbIndicator = 1.0*(max1ETC*maxNumber1*normFactor) +
6.0*(max2ETC*maxNumber2*normFactor) + 15.0*(max3ETC*maxNumber3*normFactor);

    return reverbIndicator;
}

int CTimeSeries::GetLength() const
{
    return statics->nrTimesSamples - SkipAtStart;
```

```
}

int CTimeSeries::GetFrameLength() const
{
    return statics->frameLength;
}

}
```