```
    typedef double XFLOAT;
    typedef double OTA_FLOAT;

namespace POLQAV2
{
    class CDelayPara
    {
        public:
            CDelayPara()
            {
                Init();
            };

            ~CDelayPara()
            {
                if(pIgnoreFrameFlags) delete[] pIgnoreFrameFlags;
                if(pActiveFrameFlags) delete[] pActiveFrameFlags;
                if(pAslActiveFrameFlags) delete[] pAslActiveFrameFlags;
                if(pDelayReliability) delete[] pDelayReliability;
                if(pPitchVecOfRef) delete[] pPitchVecOfRef;
                if(pPitchVecOfDeg) delete[] pPitchVecOfDeg;
            };

            void operator= (const CDelayPara &Src)
            {
                Free();

                Framesize = Src.Framesize;
                FramesUsed = Src.FramesUsed;
                MaxModelFrames = Src.MaxModelFrames;
                FramesUsed = Src.FramesUsed;
                MaxSigLen = Src.MaxSigLen;
                LogFile = Src.LogFile;
                OriginalNumberOfSamples = Src.OriginalNumberOfSamples;
                pOriginalSamples = Src.pOriginalSamples;
                DistortedNumberOfSamples = Src.DistortedNumberOfSamples;
                pDistortedSamples = Src.pDistortedSamples;
                pStartSampleUtterance = Src.pStartSampleUtterance;
                pStopSampleUtterance = Src.pStopSampleUtterance;
                pDelayUtterance = Src.pDelayUtterance;
                FirstRefSample = Src.FirstRefSample;
                FirstDegSample = Src.FirstDegSample;
                PitchFrameSize = Src.PitchFrameSize;
                PitchFreqRef = Src.PitchFreqRef;
                PitchFreqDeg = Src.PitchFreqDeg;
                AslFramelength = Src.AslFramelength;
                AslFrames = Src.AslFrames;
                mh = Src.mh;

                AllocVectors(Src.MaxModelFrames);
                pNoiseDuringSpeechdB[0] = Src.pNoiseDuringSpeechdB[0];
                pNoiseDuringSpeechdB[1] = Src.pNoiseDuringSpeechdB[1];
                pNoiseDuringSilencedB[0] = Src.pNoiseDuringSilencedB[0];
                pNoiseDuringSilencedB[1] = Src.pNoiseDuringSilencedB[1];
                pBGNSwitchingLevel[0] = Src.pBGNSwitchingLevel[0];
                pBGNSwitchingLevel[1] = Src.pBGNSwitchingLevel[1];

                matbCopy(Src.pPitchVecOfRef, pPitchVecOfRef, MaxModelFrames);
                matbCopy(Src.pPitchVecOfDeg, pPitchVecOfDeg, MaxModelFrames);
                matbCopy(Src.pDelayReliability, pDelayReliability, MaxModelFrames);
                matbCopy(Src.pIgnoreFrameFlags, pIgnoreFrameFlags, MaxModelFrames);

                memcpy(pActiveFrameFlags, Src.pActiveFrameFlags,
sizeof(bool)*MaxModelFrames);
                memcpy(pAslActiveFrameFlags, Src.pAslActiveFrameFlags,
sizeof(bool)*MaxModelFrames);
            }

            void Check()
            {
                if (PitchFreqRef != PitchFreqRef)
                    DebugBreak();
                if (PitchFreqDeg != PitchFreqDeg)
                    DebugBreak();
```

```
        if (pNoiseDuringSpeechdB[0] != pNoiseDuringSpeechdB[0])
            DebugBreak();
        if (pNoiseDuringSpeechdB[1] != pNoiseDuringSpeechdB[1])
            DebugBreak();
        if (pNoiseDuringSilencedB[0] != pNoiseDuringSilencedB[0])
            DebugBreak();
        if (pNoiseDuringSilencedB[1] != pNoiseDuringSilencedB[1])
            DebugBreak();
        if (pBGNSwitchingLevel[0] != pBGNSwitchingLevel[0])
            DebugBreak();
        if (pBGNSwitchingLevel[1] != pBGNSwitchingLevel[1])
            DebugBreak();

        for (int i=0; i<MaxModelFrames; i++)
            if (pPitchVecOfRef[i] != pPitchVecOfRef[i])
                DebugBreak();
        for (int i=0; i<MaxModelFrames; i++)
            if (pPitchVecOfDeg[i] != pPitchVecOfDeg[i])
                DebugBreak();
        for (int i=0; i<MaxModelFrames; i++)
            if (pDelayReliability[i] != pDelayReliability[i])
                DebugBreak();

        if (FramesUsed<0 || FramesUsed>MaxModelFrames)
            DebugBreak();
        if (AslFrames<0 || AslFrames>MaxModelFrames)
            DebugBreak();

        if (OriginalNumberOfSamples<0 || OriginalNumberOfSamples>MaxSigLen)
            DebugBreak();
        if (DistortedNumberOfSamples<0 || DistortedNumberOfSamples>MaxSigLen)
            DebugBreak();

        if (FirstRefSample<0 || FirstRefSample>OriginalNumberOfSamples)
            DebugBreak();
        if (FirstDegSample<0 || FirstDegSample>DistortedNumberOfSamples)
            DebugBreak();

        if (AslFramelength<0 || AslFramelength>4096)
            DebugBreak();

    }

    void Init()
    {
        pOriginalSamples = 0;
        pDistortedSamples = 0;
        pIgnoreFrameFlags = 0;
        pActiveFrameFlags = 0;
        pAslActiveFrameFlags = 0;
        pStartSampleUtterance = 0;
        pStopSampleUtterance = 0;
        pDelayUtterance = 0;
        pDelayReliability = 0;
        pPitchVecOfRef = 0;
        pPitchVecOfDeg = 0;
    }

    void Free()
    {
        if(pIgnoreFrameFlags) delete[] pIgnoreFrameFlags;
        if(pActiveFrameFlags) delete[] pActiveFrameFlags;
        if(pAslActiveFrameFlags) delete[] pAslActiveFrameFlags;
        if(pDelayReliability) delete[] pDelayReliability;
        if(pPitchVecOfRef) delete[] pPitchVecOfRef;
        if(pPitchVecOfDeg) delete[] pPitchVecOfDeg;
        Init();
    }

    void AllocVectors(int NumFrames)
    {
        pIgnoreFrameFlags = new int[NumFrames];
        pActiveFrameFlags = new bool[NumFrames];
        pAslActiveFrameFlags = new bool[NumFrames];
        pDelayReliability = new XFLOAT[NumFrames];
        pPitchVecOfRef = new XFLOAT[NumFrames];
```

```
                pPitchVecOfDeg = new XFLOAT[NumFrames];
                MaxModelFrames = NumFrames;

                for (int i=0; i<NumFrames; i++) pActiveFrameFlags[i] = false;
                for (int i=0; i<NumFrames; i++) pAslActiveFrameFlags[i] = false;
                matbSet(0.0, pPitchVecOfRef, NumFrames);
                matbSet(0.0, pPitchVecOfDeg, NumFrames);
                matbSet(0.0, pDelayReliability, NumFrames);
            };

            void SetUtteranceInfo(int NumUtterances, int* pStart, int* pStop, int*
pDelay)
            {
                pStartSampleUtterance->SetSize(NumUtterances);
                pStopSampleUtterance->SetSize(NumUtterances);
                pDelayUtterance->SetSize(NumUtterances);
                for (int f=0; f < NumUtterances; f++)
                {
                    (pStartSampleUtterance->m_pData)[f] = pStart[f];
                    (pStopSampleUtterance->m_pData)[f] = pStop[f];
                    (pDelayUtterance->m_pData)[f] = pDelay[f];
                }
            }

            void Print(FILE* pFile);

            int Framesize;
            int MaxModelFrames;
            int FramesUsed;
            long MaxSigLen;
            FILE* LogFile;

            long            OriginalNumberOfSamples;
            XFLOAT*         pOriginalSamples;

            long            DistortedNumberOfSamples;
            XFLOAT*         pDistortedSamples;

            CIntArray*      pStartSampleUtterance;
            CIntArray*      pStopSampleUtterance;
            CIntArray*      pDelayUtterance;
            int             FirstRefSample;
            int             FirstDegSample;

            XFLOAT* pDelayReliability;

            XFLOAT pNoiseDuringSpeechdB[2];
            XFLOAT pNoiseDuringSilencedB[2];
            XFLOAT pBGNSwitchingLevel[2];

            int PitchFrameSize;
            XFLOAT PitchFreqRef;
            XFLOAT PitchFreqDeg;
            XFLOAT* pPitchVecOfRef;
            XFLOAT* pPitchVecOfDeg;

            bool*           pActiveFrameFlags;

            int*            pIgnoreFrameFlags;

            int             AslFramelength;
            int             AslFrames;
            bool*           pAslActiveFrameFlags;

            MAT_HANDLE mh;
    };

    bool DoCalculateDelayDegPlus(CDelayPara* pDelayPara, POLQA_RESULT_DATA*
PolqaResults);
}

namespace POLQAV2
{

typedef struct
{
```

```
    float FrameWeightWeight;
    bool  UseRelDistance;
    float ViterbiDistanceWeightFactor;
} VITERBI_PARA;

typedef struct
{
    long Samplerate;
    int mSRDetectFineAlignCorrlen;
    int mDelayFineAlignCorrlen;
    int WindowSize[8];
    int CoarseAlignCorrlen[8];
    float pViterbiDistanceWeightFactor[8];
} SPEECH_WINDOW_PARA;

typedef struct
{
    SPEECH_WINDOW_PARA Win[3];
    float LowEnergyThresholdFactor;
    float LowCorrelThreshold;

    float FineAlignLowEnergyThresh;
    float FineAlignLowEnergyCorrel;
    float FineAlignShortDropOfCorrelR;
    float FineAlignShortDropOfCorrelRLastBest;
    float ViterbiDistanceWeightFactorDist;
    float ViterbiDistanceWeightFactor;
} SPEECH_TA_PARA;

typedef struct
{
    SPEECH_WINDOW_PARA Win[3];
    float LowEnergyThresholdFactor;
    float LowCorrelThreshold;

    float FineAlignLowEnergyThresh;
    float FineAlignLowEnergyCorrel;
    float FineAlignShortDropOfCorrelR;
    float FineAlignShortDropOfCorrelRLastBest;
    float ViterbiDistanceWeightFactorDist;
    float ViterbiDistanceWeightFactor;
} AUDIO_TA_PARA;

typedef struct
{
    float mCorrForSkippingInitialDelaySearch;
    int CoarseAlignSegmentLengthInMs;
} GENERAL_TA_PARA;

typedef struct
{
    void Init(long Samplerate)
    {
        if (Samplerate==16000)      MaxWin=4;
        else if (Samplerate==8000)  MaxWin=4;
        else                        MaxWin=4;

        LowPeakEliminationThreshold= 0.20000000298023223;

        if (Samplerate==16000)      PercentageRequired = 0.05F;
        else if (Samplerate==8000)  PercentageRequired = 0.1F;
        else                        PercentageRequired = 0.02F;

        MaxDistance = 14;

        MinReliability = 7;

        PercentageRequired = 0.7;
        OTA_FLOAT MaxGradient = 1.1;
        OTA_FLOAT MaxTimescaling = 0.1;

        if (Samplerate==48000)      MaxStepPerFrame = MaxGradient * 1024.0;
        else if (Samplerate==8000)  MaxStepPerFrame = MaxGradient * 128.0;
        MaxBins = ((int)(MaxStepPerFrame*2.0*0.9));
        MaxStepPerFrame *= 4;
```

```
    }

    float LowEnergyThresholdFactor;
    float LowCorrelThreshold;

    int     MaxStepPerFrame;
    int     MaxBins;
    int     MaxWin;
    int     MinHistogramData;

    float   MinReliability;

    double  LowPeakEliminationThreshold;
    float   MinFrequencyOfOccurrence;
    float   LargeStepLimit;

    float   MaxDistanceToLast;
    float   MaxDistance;
    float   MaxLargeStep;

    float   ReliabilityThreshold;
    float   PercentageRequired;

    float   AllowedDistancePara2;
    float   AllowedDistancePara3;
} SR_ESTIMATION_PARA;

class CParameters
{
    public:
        CParameters()
        {
            int i;
            mTAPara.mCorrForSkippingInitialDelaySearch = 0.6F;
            mTAPara.CoarseAlignSegmentLengthInMs = 600;

            SPEECH_WINDOW_PARA      SpeechWinPara[] =
            {
                    {8000,   32, 32,
                        {128,  256, 128,  64,  32,  0, 0},
                        {-1,    -1,  -1,  85,  35,  0,  0},
                        {-1,    -1,  -1,  16,  12,  0,  0}},
                    {16000,  64, 64,
                        {256,  512, 256, 128,  64,  0},
                        {-1,    -1,  -1,  64,  34,  0},
                        {-1,    -1,  -1,  12,  10,  0}},
                    {48000, 256, 256,
                        {512, 1024, 512, 512, 128,  0},
                        {-1,    -1,  -1, 116,  62,  0},
                        {-1,    -1,  -1,  18,  16,  0}}
            };

            for (i=0; i<3; i++)
            {
                mSpeechTAPara.Win[i].Samplerate = SpeechWinPara[i].Samplerate;
                mSpeechTAPara.Win[i].mDelayFineAlignCorrlen =
SpeechWinPara[i].mDelayFineAlignCorrlen;
                mSpeechTAPara.Win[i].mSRDetectFineAlignCorrlen =
SpeechWinPara[i].mSRDetectFineAlignCorrlen;
                for (int k=0; k<8; k++)
                {
                    mSpeechTAPara.Win[i].CoarseAlignCorrlen[k] =
SpeechWinPara[i].CoarseAlignCorrlen[k];
                    mSpeechTAPara.Win[i].WindowSize[k]         =
SpeechWinPara[i].WindowSize[k];
                    mSpeechTAPara.Win[i].pViterbiDistanceWeightFactor[k] =
SpeechWinPara[i].pViterbiDistanceWeightFactor[k];
                }
            }
            mSpeechTAPara.LowEnergyThresholdFactor = 15.0F;
            mSpeechTAPara.LowCorrelThreshold = 0.4F;
            mSpeechTAPara.FineAlignLowEnergyThresh = 2.0;
            mSpeechTAPara.FineAlignLowEnergyCorrel = 0.6F;
            mSpeechTAPara.FineAlignShortDropOfCorrelR = -1;
            mSpeechTAPara.FineAlignShortDropOfCorrelRLastBest = 0.65F;
```

```
            mSpeechTAPara.ViterbiDistanceWeightFactorDist = 5;

            SPEECH_WINDOW_PARA      AudioWinPara[] =
            {
                    {8000,   32, 32,
                        {64,   128,  64,  64,  16,  0, 0},
                        {-1,    -1,  -1, 128,  32,  0, 0},
                        {-1,    -1,  -1,   6,   6,  0, 0}},
                    {16000,  64, 64,
                        {128,  256, 128, 128,  32,  0},
                        {-1,    -1,  -1,  64,  32,  0},
                        {-1,    -1,  -1,  12,  12,  0}},
                    {48000, 256, 2048,
                        {512, 1024, 512, 512, 256,  128,   0},
                        {-1,    -1,  -1, 512, 1024, 2048,  0},
                        {-1,    -1,  -1,  16,  16,  32,    0}}
            };

            for (i=0; i<3; i++)
            {
                mAudioTAPara.Win[i].Samplerate = AudioWinPara[i].Samplerate;
                mAudioTAPara.Win[i].mDelayFineAlignCorrlen    =
AudioWinPara[i].mDelayFineAlignCorrlen;
                mAudioTAPara.Win[i].mSRDetectFineAlignCorrlen =
AudioWinPara[i].mSRDetectFineAlignCorrlen;
                for (int k=0; k<8; k++)
                {
                    mAudioTAPara.Win[i].CoarseAlignCorrlen[k] =
AudioWinPara[i].CoarseAlignCorrlen[k];
                    mAudioTAPara.Win[i].WindowSize[k]         =
AudioWinPara[i].WindowSize[k];
                    mAudioTAPara.Win[i].pViterbiDistanceWeightFactor[k] =
AudioWinPara[i].pViterbiDistanceWeightFactor[k];
                }
            }
            mAudioTAPara.LowEnergyThresholdFactor = 1;
            mAudioTAPara.LowCorrelThreshold = 0.85F;
            mAudioTAPara.FineAlignLowEnergyThresh = 32.0;
            mAudioTAPara.FineAlignLowEnergyCorrel = 0.8F;
            mAudioTAPara.FineAlignShortDropOfCorrelR = -1;
            mAudioTAPara.FineAlignShortDropOfCorrelRLastBest = 0.8F;
            mAudioTAPara.ViterbiDistanceWeightFactorDist = 6;

            mSREPara.LowEnergyThresholdFactor = 15.0F;
            mSREPara.LowCorrelThreshold = 0.4F;

            mSREPara.MaxStepPerFrame = 160;
            mSREPara.MaxBins = ((int)(mSREPara.MaxStepPerFrame*2.0*0.9));

            mSREPara.MaxWin=4;
            mSREPara.LowPeakEliminationThreshold=0.2000000029802322F;
            mSREPara.PercentageRequired = 0.04F;

            mSREPara.LargeStepLimit = 0.08F;
            mSREPara.MaxDistanceToLast = 7;
            mSREPara.MaxLargeStep = 5;
            mSREPara.MaxDistance = 14;

            mSREPara.MinReliability = 7;
            mSREPara.MinFrequencyOfOccurrence = 3;

            mSREPara.AllowedDistancePara2 = 0.85F;
            mSREPara.AllowedDistancePara3 = 1.5F;

            mSREPara.ReliabilityThreshold = 0.3F;
            mSREPara.MinHistogramData = 8;

            mViterbi.UseRelDistance = false;
            mViterbi.FrameWeightWeight = 1.0F;
        };

        void Init(long Samplerate)
        {
            mSREPara.Init(Samplerate);
        }
```

```
        VITERBI_PARA        mViterbi;
        GENERAL_TA_PARA     mTAPara;
        SPEECH_TA_PARA      mSpeechTAPara;
        AUDIO_TA_PARA       mAudioTAPara;
        SR_ESTIMATION_PARA  mSREPara;
};
}

namespace POLQAV2
{

class CProcessData
{
    public:
        CProcessData()
        {
            int i;

            mCurrentIteration = -1;
            mStartPlotIteration=10;
            mLastPlotIteration =10;
            mEnablePlotting=false;
            mpLogFile = 0;

            mWindowSize = 2048;
            mSRDetectFineAlignCorrlen = 1024;
            mDelayFineAlignCorrlen = 1024;
            mOverlap    = 1024;
            mSamplerate = 48000;
            mNumSignals = 0;
            mpMathlibHandle = 0;
            mMinLowVarDelay = -99999999;
            mMaxHighVarDelay = 99999999;

            mMinStaticDelayInMs = -2500;
            mMaxStaticDelayInMs = 2500;

            mMaxToleratedRelativeSamplerateDifference = 1.0;

            for (i=0; i<8; i++)
                mpViterbiDistanceWeightFactor[i] = 0.0001F;
        }

        int mMinStaticDelayInMs;
        int mMaxStaticDelayInMs;

        int mMinLowVarDelayInSamples;
        int mMaxHighVarDelayInSamples;

        int mStartPlotIteration;
        int mLastPlotIteration;
        bool mEnablePlotting;
        long mSamplerate;

        FILE* mpLogFile;

        int mCurrentIteration;

        int mpWindowSize[8];

        int mpOverlap[8];

        int mpCoarseAlignCorrlen[8];

        float mpViterbiDistanceWeightFactor[8];

        int mDelayFineAlignCorrlen;
        int mSRDetectFineAlignCorrlen;
        float mMaxToleratedRelativeSamplerateDifference;
        int mWindowSize;

        int mOverlap;

        int mCoarseAlignCorrlen;

        int mNumSignals;
```

```
        void* mpMathlibHandle;

        int mMinLowVarDelay;
        int mMaxHighVarDelay;
        int mStepSize;

        bool Init(int Iteration, float MoreDownsampling)
        {
            assert(MoreDownsampling);

            mCurrentIteration = Iteration;
            mP.Init(mSamplerate);

            mWindowSize = (int)((float)mpWindowSize[Iteration]*MoreDownsampling);
            mOverlap = (int)((float)mpOverlap[Iteration]*MoreDownsampling);
            mCoarseAlignCorrlen = mpCoarseAlignCorrlen[Iteration];
            mStepSize = mWindowSize - mOverlap;
            mMinLowVarDelay = mMinLowVarDelayInSamples / mStepSize;
            mMaxHighVarDelay = mMaxHighVarDelayInSamples / mStepSize;

            float D = mpViterbiDistanceWeightFactor[Iteration];
            D = D * mSamplerate / mStepSize / 1000;
            float F = ((float)log(1+0.5)) / (D*D);
            mP.mViterbi.ViterbiDistanceWeightFactor = F;

            D = mP.mSpeechTAPara.ViterbiDistanceWeightFactorDist;
            D = D * mSamplerate / 1000;
            F =((float) log(1+0.5) / (D*D));
            mP.mSpeechTAPara.ViterbiDistanceWeightFactor = F;

            return true;
        }

        CParameters   mP;
};

class SECTION
{
    public:
        int Start;
        int End;
        int Len() {return End-Start;};
        void CopyFrom(const SECTION &src)
        {
            this->Start = src.Start;
            this->End   = src.End;
        }
};

typedef struct OTA_RESULT
{
    void CopyFrom(const OTA_RESULT* src)
    {
        mNumFrames           = src->mNumFrames;
        mStepsize            = src->mStepsize;
        mResolutionInSamples = src->mResolutionInSamples;
        if (src->mpDelay != NULL && mNumFrames > 0)
        {
            matFree(mpDelay);
            mpDelay = (long*)matMalloc(mNumFrames * sizeof(long));
            for (int i = 0; i < mNumFrames; i++)
                mpDelay[i] = src->mpDelay[i];
        }
        else
        {
            matFree(mpDelay);
            mpDelay = NULL;
        }

        if (src->mpReliability != NULL && mNumFrames > 0)
        {
            matFree(mpReliability);
            mpReliability = (OTA_FLOAT*)matMalloc(mNumFrames * sizeof(OTA_FLOAT));
            for (int i = 0; i < mNumFrames; i++)
                mpReliability[i] = src->mpReliability[i];
        }
```

```
else
{
    matFree(mpReliability);
    mpReliability = NULL;
}
mAvgReliability   = src->mAvgReliability;
mRelSamplerateDev = src->mRelSamplerateDev;

mNumUtterances = src->mNumUtterances;
if (src->mpStartSampleUtterance != NULL && mNumUtterances > 0)
{
    matFree(mpStartSampleUtterance);
    mpStartSampleUtterance = (int*)matMalloc(mNumUtterances * sizeof(int));
    for (int i = 0; i < mNumUtterances; i++)
        mpStartSampleUtterance[i] = src->mpStartSampleUtterance[i];
}
else
{
    matFree(mpStartSampleUtterance);
    mpStartSampleUtterance = NULL;
}
if (src->mpStopSampleUtterance != NULL && mNumUtterances > 0)
{
    matFree(mpStopSampleUtterance);
    mpStopSampleUtterance = (int*)matMalloc(mNumUtterances * sizeof(int));
    for (int i = 0; i < mNumUtterances; i++)
        mpStopSampleUtterance[i] = src->mpStopSampleUtterance[i];
}
else
{
    matFree(mpStopSampleUtterance);
    mpStopSampleUtterance = NULL;
}
if (src->mpDelayUtterance != NULL && mNumUtterances > 0)
{
    matFree(mpDelayUtterance);
    mpDelayUtterance = (int*)matMalloc(mNumUtterances * sizeof(int));
    for (int i = 0; i < mNumUtterances; i++)
        mpDelayUtterance[i] = src->mpDelayUtterance[i];
}
else
{
    matFree(mpDelayUtterance);
    mpDelayUtterance = NULL;
}

mNumSections = src->mNumSections;
if (src->mpRefSections != NULL && mNumSections > 0)
{
    delete[] mpRefSections;
    mpRefSections = new SECTION[mNumSections];
    for (int i = 0; i < mNumSections; i++)
        mpRefSections[i].CopyFrom(src->mpRefSections[i]);
}
else
{
    delete[] mpRefSections;
    mpRefSections = NULL;
}
if (src->mpDegSections != NULL && mNumSections > 0)
{
    delete[] mpDegSections;
    mpDegSections = new SECTION[mNumSections];
    for (int i = 0; i < mNumSections; i++)
        mpDegSections[i].CopyFrom(src->mpDegSections[i]);
}
else
{
    delete[] mpDegSections;
    mpDegSections = NULL;
}

mSNRRefdB = src->mSNRRefdB;
mSNRDegdB = src->mSNRDegdB;
mNoiseLevelRef = src->mNoiseLevelRef;
mNoiseLevelDeg = src->mNoiseLevelDeg;
```

```
    mSignalLevelRef = src->mSignalLevelRef;
    mSignalLevelDeg = src->mSignalLevelDeg;
    mNoiseThresholdRef = src->mNoiseThresholdRef;
    mNoiseThresholdDeg = src->mNoiseThresholdDeg;

    if (src->mpActiveFrameFlags != NULL && mNumFrames > 0)
    {
        matFree(mpActiveFrameFlags);
        mpActiveFrameFlags = (int*)matMalloc(mNumFrames * sizeof(int));
        for (int i = 0; i < mNumFrames; i++)
            mpActiveFrameFlags[i] = src->mpActiveFrameFlags[i];
    }
    else
    {
        matFree(mpActiveFrameFlags);
        mpActiveFrameFlags = NULL;
    }

    if (src->mpIgnoreFlags != NULL && mNumFrames > 0)
    {
        matFree(mpIgnoreFlags);
        mpIgnoreFlags = (int*)matMalloc(mNumFrames * sizeof(int));
        for (int i = 0; i < mNumFrames; i++)
            mpIgnoreFlags[i] = src->mpIgnoreFlags[i];
    }
    else
    {
        matFree(mpIgnoreFlags);
        mpIgnoreFlags = NULL;
    }

    for (int i = 0; i < 5; i++)
        mTimeDiffs[i] = src->mTimeDiffs[i];

    mAslFrames = src->mAslFrames;
    mAslFramelength = src->mAslFramelength;
    if (src->mpAslActiveFrameFlags != NULL && mAslFrames > 0)
    {
        matFree(mpAslActiveFrameFlags);
        mpAslActiveFrameFlags = (int*)matMalloc(mAslFrames * sizeof(int));
        for (int i = 0; i < mAslFrames; i++)
            mpAslActiveFrameFlags[i] = src->mpAslActiveFrameFlags[i];
    }
    else
    {
        matFree(mpAslActiveFrameFlags);
        mpAslActiveFrameFlags = NULL;
    }

    FirstRefSample = src->FirstRefSample;
    FirstDegSample = src->FirstDegSample;
}

OTA_RESULT()
{
    mNumFrames = 0;
    mpDelay = NULL;

    mpReliability = NULL;

    mNumUtterances = 0;
    mpStartSampleUtterance = NULL;
    mpStopSampleUtterance  = NULL;
    mpDelayUtterance       = NULL;

    mNumSections = 0;
    mpRefSections = NULL;
    mpDegSections = NULL;

    mpActiveFrameFlags = NULL;
    mpIgnoreFlags = NULL;

    mAslFrames = 0;
    mAslFramelength = 0;
    mpAslActiveFrameFlags = NULL;
```

```
        FirstRefSample = FirstDegSample = 0;
    }

    ~OTA_RESULT()
    {
        matFree(mpDelay);
        mpDelay = NULL;

        matFree(mpReliability);
        mpReliability = NULL;

        matFree(mpStartSampleUtterance);
        mpStartSampleUtterance = NULL;

        matFree(mpStopSampleUtterance);
        mpStopSampleUtterance  = NULL;

        matFree(mpDelayUtterance);
        mpDelayUtterance       = NULL;

        delete[] mpRefSections;
        mpRefSections = NULL;
        delete[] mpDegSections;
        mpDegSections = NULL;

        matFree(mpActiveFrameFlags);
        mpActiveFrameFlags = NULL;

        matFree(mpIgnoreFlags);
        mpIgnoreFlags = NULL;

        matFree(mpAslActiveFrameFlags);
        mpAslActiveFrameFlags = NULL;
    }

    long mNumFrames;
    int mStepsize;
    int mResolutionInSamples;
    int mPitchFrameSize;
    long *mpDelay;
    OTA_FLOAT *mpReliability;
    OTA_FLOAT mAvgReliability;
    OTA_FLOAT mRelSamplerateDev;

    int  mNumUtterances;
    int* mpStartSampleUtterance;
    int* mpStopSampleUtterance;
    int* mpDelayUtterance;
    int FirstRefSample;
    int FirstDegSample;

    int         mNumSections;
    SECTION     *mpRefSections;
    SECTION     *mpDegSections;

    double mSNRRefdB, mSNRDegdB;
    double mNoiseLevelRef, mNoiseLevelDeg;
    double mSignalLevelRef, mSignalLevelDeg;
    double mNoiseThresholdRef, mNoiseThresholdDeg;

    int *mpActiveFrameFlags;

    int *mpIgnoreFlags;

    int mAslFrames;
    int mAslFramelength;
    int *mpAslActiveFrameFlags;

    double mTimeDiffs[5];

}OTA_RESULT;

struct FilteringParameters
{
    int pListeningCondition;
    double cutOffFrequencyLow;
```

```
    double cutOffFrequencyHigh;
    double disturbedEnergyQuotient;
};

class ITempAlignment
{
    public:

        virtual bool Init(CProcessData* pProcessData)=0;
        virtual void Free()=0;
        virtual void Destroy()=0;

        virtual bool SetSignal(int Index, unsigned long SampleRate, unsigned long
NumSamples, int NumChannels, OTA_FLOAT** pSignal)=0;

        virtual void GetFilterCharacteristics(FilteringParameters *FilterParams)=0;

        virtual bool FilterSignal(int Index, FilteringParameters *FilterParams)=0;

        virtual bool Run(unsigned long Control, OTA_RESULT* pResult, int TArunIndex)=0;

        virtual void GetNoiseSwitching(OTA_FLOAT* pBGNSwitchingLevel, OTA_FLOAT*
pNoiseLevelSpeechDeg, OTA_FLOAT* pNoiseLevelSilenceDeg)=0;

        virtual OTA_FLOAT GetPitchFreq(int Signal, int Channel)=0;

        virtual OTA_FLOAT GetPitchVector(int Signal, int Channel, OTA_FLOAT* pVector,
int NumFrames, int SamplesPerFrame)=0;
        virtual int GetPitchFrameSize()=0;
};

enum AlignmentType
{
    TA_FOR_SPEECH=0,

};

ITempAlignment* CreateAlignment(AlignmentType Type);

}

namespace POLQAV2
{

extern XFLOAT aCmdlineArray[];
extern int    aElementsInCmdlineArray;

void DeleteUtteranceFromVector(int UttIndexToDelete, CDelayPara* pDelayPara)
{
    for (int i=UttIndexToDelete; i< pDelayPara->pDelayUtterance->GetSize()-1; i++)
    {
        pDelayPara->pDelayUtterance->m_pData[i] =
pDelayPara->pDelayUtterance->m_pData[i+1];
        pDelayPara->pStartSampleUtterance->m_pData[i] =
pDelayPara->pStartSampleUtterance->m_pData[i+1];
        pDelayPara->pStopSampleUtterance->m_pData[i] =
pDelayPara->pStopSampleUtterance->m_pData[i+1];
    }
    pDelayPara->pDelayUtterance->SetSize(pDelayPara->pDelayUtterance->GetSize()-1);
    pDelayPara->pStartSampleUtterance->SetSize(pDelayPara->pStartSampleUtterance->GetSi
ze()-1);
    pDelayPara->pStopSampleUtterance->SetSize(pDelayPara->pStopSampleUtterance->GetSize
()-1);
}

namespace POLQAV2
{

typedef enum
{
    PLOT_TYPE_TIME_SERIES=0,
    PLOT_TYPE_SPECTRUM=1,
    PLOT_TYPE_XY=2,
    PLOT_TYPE_SCATTER=3,
    PLOT_TYPE_CHARTDEF=4
} SERIES_TYPE;
```

```
typedef enum
{
    PLOT_CHARTTYPE_MULTILINE=0,
    PLOT_CHARTTYPE_MULTICHART=1,
    PLOT_CHARTTYPE_SINGLE=2
} CHART_TYPE;

class PLOT_VECTOR
{
    public:
        PLOT_VECTOR() {SetZero();}
        ~PLOT_VECTOR() {Delete();}

        void Delete()
        {
            if (pfY) delete[] pfY;
            if (pdY) delete[] pdY;
            if (piY) delete[] piY;
            if (plY) delete[] plY;
            if (pofY) delete[] pofY;

            if (pfX) delete[] pfX;
            if (pdX) delete[] pdX;
            if (piX) delete[] piX;
            if (plX) delete[] plX;
            if (pofX) delete[] pofX;
            if (pName) delete[] pName;
            if (pXAxisLabel) delete[] pXAxisLabel;
            if (pYAxisLabel) delete[] pYAxisLabel;
            if (pText1) delete[] pText1;
            if (pText2) delete[] pText2;
            if (pEquationOfFunction) delete[] pEquationOfFunction;
        };

        void SetZero()
        {
            pfX=0;pdX=0; piX=0; plX=0; pofX=0;
            pfY=0; pdY=0; piY=0; plY=0; pofY=0;
            pName=0;pXAxisLabel=0;pYAxisLabel=0;
            pMoreVecs=0; Portrait=false;
            pText1=0; pText2=0;
            pEquationOfFunction = 0;
            UseCustomXRange = false;
            UseCustomYRange = false;
        }

        void Free()
        {
            Delete();
            SetZero();
        }

        char* pName;
        char* pXAxisLabel;
        char* pYAxisLabel;
        bool  Portrait;

        OTA_FLOAT* pofY;
        float* pfY;
        double* pdY;
        int* piY;
        long* plY;

        OTA_FLOAT* pofX;
        float* pfX;
        double* pdX;
        int* piX;
        long* plX;

        PLOT_VECTOR* pMoreVecs;
        long  Len;
        SERIES_TYPE Type;

        int XPos;
        int YPos;
```

```
        int FFTLen;
        int WinType;

        char* pText1;
        float XPosText1;
        float YPosText1;
        char* pText2;
        float XPosText2;
        float YPosText2;

        bool UseCustomXRange;
        float XRange[2];
        bool UseCustomYRange;
        float YRange[2];

        char *pEquationOfFunction;
        double XMin;
        double XMax;
};

void SetVecInfoTimeSeries(PLOT_VECTOR* pVecInfo, const OTA_FLOAT* VecY, const unsigned
long VecLen, int XPos, int YPos, float YScaleFac, char * Format, ... );
void SetVecInfoTimeSeries(PLOT_VECTOR* pVecInfo, const int* VecY, const unsigned long
VecLen, int XPos, int YPos, float YScaleFac, char * Format, ... );
void SetVecInfoTimeSeries(PLOT_VECTOR* pVecInfo, const long* VecY, const unsigned long
VecLen, int XPos, int YPos, float YScaleFac, char * Format, ... );

void SetVecInfoXYCore(PLOT_VECTOR* pVecInfo, const OTA_FLOAT* VecX, const OTA_FLOAT*
VecY, const unsigned long VecLen, int XPos, int YPos, float XScaleFac, float
YScaleFac);
void SetVecInfoXYCore(PLOT_VECTOR* pVecInfo, const float* VecX, const float* VecY,
const unsigned long VecLen, int XPos, int YPos, float XScaleFac, float YScaleFac);
void SetVecInfoXYCore(PLOT_VECTOR* pVecInfo, const double* VecX, const double* VecY,
const unsigned long VecLen, int XPos, int YPos, float XScaleFac, float YScaleFac);
void SetVecInfoXYCore(PLOT_VECTOR* pVecInfo, const int* VecX, const int* VecY, const
unsigned long VecLen, int XPos, int YPos, float XScaleFac, float YScaleFac);
void SetVecInfoXYCore(PLOT_VECTOR* pVecInfo, const long* VecX, const long* VecY, const
unsigned long VecLen, int XPos, int YPos, float XScaleFac, float YScaleFac);
void SetFunction(PLOT_VECTOR* pVecInfo, const char *Equation, char *Title, double XMin,
double XMax);

void SetVecInfoSpectrum(PLOT_VECTOR* pVecInfo, const OTA_FLOAT* VecY, const unsigned
long VecLen, int XPos, int YPos, int FFTLen, int WinType, char * Format, ... );

void PlotNVectors(const PLOT_VECTOR* pVecs, const int NumVecs, CHART_TYPE Type, char*
Format, ... );
void PlotNVectors(char* PsFilename, const PLOT_VECTOR* pVecs, const int NumVecs,
CHART_TYPE Type, char* Format, ... );
void PlotNVectors(char* PsFilename, const char* pTitle, const PLOT_VECTOR* pVecs, const
int NumVecs, CHART_TYPE Type);

void AddText1(PLOT_VECTOR* pChartInfo, float X, float Y, char* Format, ... );
void AddText2(PLOT_VECTOR* pChartInfo, float X, float Y, char* Format, ... );
void AddXRange(PLOT_VECTOR* pChartInfo, float XMin, float XMax);
void AddYRange(PLOT_VECTOR* pChartInfo, float YMin, float YMax);

template <typename T> void SetVecInfoXYPlot(PLOT_VECTOR* pVecInfo, T VecX, T VecY,
const unsigned long VecLen, int XPos, int YPos, float XScaleFac, float YScaleFac, const
char* XLabel, const char* YLabel, const char * Format, ... )
{

    pVecInfo->Free();
    va_list Args;
    va_start(Args, Format);
    if (Format)
    {
        int TitleLen = _vscprintf( Format, Args )+1;
        pVecInfo->pName = new char[TitleLen];
        vsprintf_s( pVecInfo->pName, TitleLen, Format, Args );
    }
    SetVecInfoXYCore(pVecInfo, VecX, VecY, VecLen, XPos, YPos, XScaleFac, YScaleFac);
    pVecInfo->Type = PLOT_TYPE_XY;
    int LabelLen = strlen(XLabel)+1;
    if (LabelLen)
    {
        pVecInfo->pXAxisLabel = new char[LabelLen];
```

```
        strcpy(pVecInfo->pXAxisLabel, XLabel);
    }
    LabelLen = strlen(YLabel)+1;
    if (LabelLen)
    {
        pVecInfo->pYAxisLabel = new char[LabelLen];
        strcpy(pVecInfo->pYAxisLabel, YLabel);
    }

    pVecInfo->Len = VecLen;
    pVecInfo->XPos = XPos;
    pVecInfo->YPos = YPos;

}

template <typename T> void SetVecInfoScatterPlot(PLOT_VECTOR* pVecInfo, T VecX, T VecY,
const unsigned long VecLen, int XPos, int YPos, float XScaleFac, float YScaleFac, const
char* XLabel, const char* YLabel, const char * Format, ... )
{

    pVecInfo->Free();
    va_list Args;
    va_start(Args, Format);
    if (Format)
    {
        int TitleLen = _vscprintf( Format, Args )+1;
        pVecInfo->pName = new char[TitleLen];
        vsprintf_s( pVecInfo->pName, TitleLen, Format, Args );
    }
    SetVecInfoXYCore(pVecInfo, VecX, VecY, VecLen, XPos, YPos, XScaleFac, YScaleFac);
    pVecInfo->Type = PLOT_TYPE_SCATTER;
    int LabelLen = strlen(XLabel)+1;
    if (LabelLen)
    {
        pVecInfo->pXAxisLabel = new char[LabelLen];
        strcpy(pVecInfo->pXAxisLabel, XLabel);
    }
    LabelLen = strlen(YLabel)+1;
    if (LabelLen)
    {
        pVecInfo->pYAxisLabel = new char[LabelLen];
        strcpy(pVecInfo->pYAxisLabel, YLabel);
    }

    pVecInfo->Len = VecLen;
    pVecInfo->XPos = XPos;
    pVecInfo->YPos = YPos;

}

void SetChartInfo(PLOT_VECTOR* pChartInfo, PLOT_VECTOR* pVecInfo, const unsigned int
NumVecs, int XPos, int YPos, const char* XLabel, const char* YLabel, const char *
Format, ... );

bool CreateMasterPSFile(const char* pPlotFilename, int NumPages);

void PlotVector(const char* pTitle, const long* Vec1, const unsigned long Size1);
void PlotVector(const char* pTitle, const int* Vec1, const unsigned long Size1);
void PlotMatrix(const char* pTitle, const int** Mat, unsigned long Dim1, unsigned long
Dim2, int Offset);
void PlotSpectrum(const char* pTitle, const float* Vec, const unsigned long Size1,
const int FFTLen, const int WinType=0);
void PlotSpectrum(const char* pTitle, const double* Vec, const unsigned long Size1,
const int FFTLen, const int WinType=0);

void PlotMatrix(const char* pTitle, const OTA_FLOAT** Mat, const unsigned long Dim1,
const unsigned long Dim2, const OTA_FLOAT Offset);
void PlotVector(const char* pTitle, const OTA_FLOAT* Vec1, const unsigned long Size1);
void PlotTwoVectors(const char* pTitle, const OTA_FLOAT* Vec1, const unsigned long
Size1, const OTA_FLOAT* Vec2, unsigned long Size2);

void SetVecInfo(PLOT_VECTOR* pVecInfo, const char* pTitle, const double*    Vec1, const
unsigned long VecLen, int XPos, int YPos, SERIES_TYPE Type=PLOT_TYPE_TIME_SERIES, int
FFTLen=4096, int WinType=0);
void SetVecInfo(PLOT_VECTOR* pVecInfo, const char* pTitle, const float* Vec1,     const
unsigned long VecLen, int XPos, int YPos, SERIES_TYPE Type=PLOT_TYPE_TIME_SERIES, int
```

```
FFTLen=4096, int WinType=0);
void SetVecInfo(PLOT_VECTOR* pVecInfo, const char* pTitle, const int*       Vec1, const
unsigned long VecLen, int XPos, int YPos, SERIES_TYPE Type=PLOT_TYPE_TIME_SERIES, int
FFTLen=4096, int WinType=0);
void SetVecInfo(PLOT_VECTOR* pVecInfo, const char* pTitle, const long*      Vec1, const
unsigned long VecLen, int XPos, int YPos, SERIES_TYPE Type=PLOT_TYPE_TIME_SERIES, int
FFTLen=4096, int WinType=0);

}

XFLOAT GetNearestPitch(XFLOAT Target, XFLOAT a, XFLOAT b, XFLOAT c)
{
    if (Target<0.6*a && Target>0.4*a) a /= 2.0;
    if (Target<0.6*b && Target>0.4*b) b /= 2.0;
    if (Target<0.6*c && Target>0.4*c) c /= 2.0;
    if (a<0.6*Target && a>0.4*Target) Target /= 2.0;

    XFLOAT Res = a;
    if (abs(Target-Res)>abs(Target-b)) Res = b;
    if (abs(Target-Res)>abs(Target-c)) Res = c;
    return Res;
}

void CorrectPitchVectors(CDelayPara* pDelayPara)
{
    int NumFrames = pDelayPara->MaxModelFrames;
    for (int i=0; i<NumFrames; i++)
    {
        if (pDelayPara->pPitchVecOfRef[i] && pDelayPara->pPitchVecOfDeg[i])
        {
            if (pDelayPara->pPitchVecOfRef[i] / pDelayPara->pPitchVecOfDeg[i] > 0.4 &&
pDelayPara->pPitchVecOfRef[i] / pDelayPara->pPitchVecOfDeg[i] <0.6)
                pDelayPara->pPitchVecOfDeg[i] /= 2.0;
            if (pDelayPara->pPitchVecOfRef[i] / pDelayPara->pPitchVecOfDeg[i] > 1.8 &&
pDelayPara->pPitchVecOfRef[i] / pDelayPara->pPitchVecOfDeg[i] <2.2)
                pDelayPara->pPitchVecOfRef[i] /= 2.0;
        }
    }
}

void AlignPitchVectors(CDelayPara* pDelayPara, XFLOAT* pPitchVec, XFLOAT* pTarget)
{
    int i, frameIndex;

    int FrameLength = pDelayPara->Framesize;
    int NumFrames = pDelayPara->MaxModelFrames;
    XFLOAT *pPitchVecAligned = (XFLOAT*)matMalloc(NumFrames * sizeof(XFLOAT));
    for (frameIndex = 0; frameIndex<NumFrames; frameIndex++)
    {
        pPitchVecAligned[frameIndex] = 0;
        int utt = GetUtteranceForFrame(*pDelayPara->pStartSampleUtterance,
*pDelayPara->pStopSampleUtterance, *pDelayPara->pDelayUtterance, frameIndex,
2*FrameLength);
        if (utt>=0)
        {
            ASSERT(utt<pDelayPara->pDelayUtterance->GetSize());
            int DegFrame = (*pDelayPara->pDelayUtterance).m_pData[utt];
            DegFrame = frameIndex+(DegFrame+FrameLength/2)/FrameLength;
            if (DegFrame>=0 && DegFrame<NumFrames)
            {

                if (DegFrame>0 && DegFrame<NumFrames-2)
                {
                    pPitchVecAligned[frameIndex] = GetNearestPitch(pTarget[frameIndex],
pPitchVec[DegFrame], pPitchVec[DegFrame+1], pPitchVec[DegFrame-1]);
                }
                else

                    pPitchVecAligned[frameIndex] = pPitchVec[DegFrame];
            }
            else
                pPitchVecAligned[frameIndex] = 0;
        }
    }

    for (i=0; i<NumFrames; i++)
```

```
        pPitchVec[i] = pPitchVecAligned[i];

    CorrectPitchVectors(pDelayPara);

    matFree(pPitchVecAligned);

}

XFLOAT AveragePitchFrequencyFromPitchVector(CDelayPara* pDelayPara, XFLOAT* pPitchVec)
{
    const XFLOAT BinWidth = 1.0;
    const int MaxBins = (int)(500.0 / BinWidth);
    int NumFrames = pDelayPara->MaxModelFrames;
    XFLOAT MaxPitch = matMax(pPitchVec, NumFrames);
    XFLOAT MinPitch = MaxPitch;
    for (int i=0; i<NumFrames; i++)
        if (pPitchVec[i]>0 && pPitchVec[i]<MinPitch)
            MinPitch = pPitchVec[i];

    if (MaxPitch-MinPitch<5*BinWidth)
    {
        MinPitch -= 3*BinWidth;
        MaxPitch += 3*BinWidth;
    }

    int NumBins = (int)((MaxPitch-MinPitch)/BinWidth+0.5);
    if (NumBins>MaxBins) NumBins = MaxBins;
    XFLOAT *pPitchHistogram = (XFLOAT*)matMalloc(NumBins * sizeof(XFLOAT));
    matbSet(0.0, pPitchHistogram, NumBins);
    for (int i=0; i<NumFrames; i++)
    {
        if (pPitchVec[i]>0.0)
        {
            int Bin = (((0.0) > ((((NumBins-1) < ((pPitchVec[i]-MinPitch) / BinWidth +
0.5)) ? (NumBins-1) : ((pPitchVec[i]-MinPitch) / BinWidth + 0.5)))) ? (0.0)
: ((((NumBins-1) < ((pPitchVec[i]-MinPitch) / BinWidth + 0.5)) ?
(NumBins-1) : ((pPitchVec[i]-MinPitch) / BinWidth + 0.5))));
            pPitchHistogram[Bin]++;
        }
    }

    int MaxPos;
    matMaxExt(pPitchHistogram, NumBins, &MaxPos);

    int MinBin = (((0) > (MaxPos - 30.0 / BinWidth + 0.5)) ? (0) : (MaxPos - 30.0 /
BinWidth + 0.5));
    int MaxBin = (((NumBins) < (MaxPos + 30.0 / BinWidth + 0.5)) ? (NumBins) : (MaxPos
+ 30.0 / BinWidth + 0.5));
    XFLOAT Avg=0.0;
    int Num = 0;
    for (int i=MinBin; i<MaxBin; i++)
    { Avg+=i*BinWidth*pPitchHistogram[i]; Num+=pPitchHistogram[i];}
    if (Num>0)
    {
        Avg /= Num;
        Avg += MinPitch;
    }
    else Avg = 0;

      matFree(pPitchHistogram);

      return Avg;
}

template <class T>
int ConvertFrameRate(int NumFramesIn, T const *pFramesIn, int FrameSizeIn, int
NumFramesOut, T* pFramesOut, int FrameSizeOut)
{
    if (FrameSizeOut==FrameSizeIn)
    {
        memcpy(pFramesOut, pFramesIn, sizeof(T)*(((NumFramesIn) < (NumFramesOut)) ?
(NumFramesIn) : (NumFramesOut)));
        for (int i=NumFramesIn; i<NumFramesOut; i++)
            pFramesOut[i] = pFramesIn[NumFramesIn-1];
        NumFramesOut = NumFramesIn;
    }
```

```
    else if (FrameSizeOut>FrameSizeIn)
    {
        int NextFramePosIn=FrameSizeIn>>1;
        int NextFrameIn=0;
        int NextFramePosOut=0;
        int i;
        for (i=0; i<NumFramesOut; i++)
        {
            while (NextFrameIn<NumFramesIn-1 && NextFramePosIn<NextFramePosOut)
            { NextFramePosIn+=FrameSizeIn; NextFrameIn++;}

            pFramesOut[i] = pFramesIn[NextFrameIn];
            NextFramePosOut += FrameSizeOut;
        }
        NumFramesOut = i;
    }
    else if (FrameSizeOut<FrameSizeIn)
    {
        int NextFramePosIn=0;
        int NextFrameIn=0;
        int NextFramePosOut=FrameSizeOut>>1;
        int i;
        for (i=0; i<NumFramesOut; i++)
        {
            while (NextFrameIn<NumFramesIn-1 && NextFramePosIn<NextFramePosOut)
            { NextFramePosIn+=FrameSizeIn; NextFrameIn++;}

            pFramesOut[i] = pFramesIn[NextFrameIn];
            NextFramePosOut += FrameSizeOut;
        }
        NumFramesOut = i;
    }
    return NumFramesOut;
}

void ConvertFrameRateOfVectors(OTA_RESULT* pTaResult, CDelayPara* pDelayPara,
POLQA_RESULT_DATA* PolqaResults)
{
    pDelayPara->FramesUsed=ConvertFrameRate(pTaResult->mNumFrames,
pTaResult->mpReliability, pTaResult->mStepsize, pDelayPara->MaxModelFrames,
PolqaResults->m_DelayReliabilityPerFrame, pDelayPara->Framesize);
    pDelayPara->FramesUsed=ConvertFrameRate(pTaResult->mNumFrames, pTaResult->mpDelay,
pTaResult->mStepsize, pDelayPara->MaxModelFrames, PolqaResults->m_DelayPerFrame,
pDelayPara->Framesize);
}

int ConvertActivityFlag(FILE*  pLogFile, OTA_RESULT* pTaResult, CDelayPara* pDelayPara,
POLQA_RESULT_DATA* PolqaResults)
{
    float modelToMacroSize = (float)pDelayPara->Framesize/(float)pTaResult->mStepsize;

    int macroFrame;
    for(int fr = 0; fr < pDelayPara->MaxModelFrames; fr++)
    {
        macroFrame = (int)(modelToMacroSize * fr + 0.5);
        if (macroFrame < pTaResult->mNumFrames)
        {
            pDelayPara->pActiveFrameFlags[fr] =
pTaResult->mpActiveFrameFlags[macroFrame] == 1;
            pDelayPara->pIgnoreFrameFlags[fr] = pTaResult->mpIgnoreFlags[macroFrame];
        }
        else
        {
            pDelayPara->pActiveFrameFlags[fr] = false;
            pDelayPara->pIgnoreFrameFlags[fr] = 0;
        }
    }
    return (int)((pTaResult->mNumFrames-0.5)/modelToMacroSize);
}

void ;

bool DoAlignmentPlus(CDelayPara* pDelayPara, POLQA_RESULT_DATA* PolqaResults,
ITempAlignment* pTA, OTA_RESULT** pTaResult, unsigned long Mode, bool* pDone)
{
    bool rc = true;
```

```
    long f;
    long ClockCycles;
    double TimeDiff;
    MAT_HANDLE mh = pDelayPara->mh;
    TACheckTimeMatInit(mh, 1);

    CProcessData TAInitData;
    TAInitData.mpMathlibHandle = pDelayPara->mh;
    TAInitData.mpLogFile = pDelayPara->LogFile;
    TAInitData.mSamplerate = PolqaResults->m_SampleFrequencyHz;
    TAInitData.mEnablePlotting = 0;
    TAInitData.mStartPlotIteration = -100;
    TAInitData.mLastPlotIteration = -10;
    TAInitData.mMinLowVarDelayInSamples  =
-(int(0.3*PolqaResults->m_SampleFrequencyHz));
    TAInitData.mMaxHighVarDelayInSamples =
(int(0.3*PolqaResults->m_SampleFrequencyHz));

    int const MAX_TA_RUNS = 2;

    int   TArunIndex        = -1;
    int   bestAlignmentIdx  = 0;
    float maxAvgReliability = -1.0f;
    OTA_RESULT *pResamplingResults[MAX_TA_RUNS];
    for (int i = 0; i < MAX_TA_RUNS; i++)
        pResamplingResults[i] = new OTA_RESULT();
    PolqaResults->m_ResamplingApplied = false;
    CDelayPara  pDelayParaSaved[MAX_TA_RUNS];

    XFLOAT *pRefSig[2] = {NULL, NULL}, *pDegSig[2] = {NULL, NULL};
    unsigned long NewLen;
    long numRefSamples = pDelayPara->OriginalNumberOfSamples, numDegSamples =
pDelayPara->DistortedNumberOfSamples;
    pRefSig[0] = (XFLOAT*)matMalloc(numRefSamples * sizeof(XFLOAT));

    pDegSig[0] = (XFLOAT*)matMalloc(numDegSamples * sizeof(XFLOAT));

    matbCopy(pDelayPara->pOriginalSamples,  pRefSig[0], numRefSamples);
    matbCopy(pDelayPara->pDistortedSamples, pDegSig[0], numDegSamples);

    bool Done = false;
    do
    {
        pTA->Init(&TAInitData);

        pTA->SetSignal(0, PolqaResults->m_SampleFrequencyHz,
pDelayPara->OriginalNumberOfSamples, 1, pRefSig);

        pTA->SetSignal(1, PolqaResults->m_SampleFrequencyHz,
pDelayPara->DistortedNumberOfSamples, 1, pDegSig);

        FilteringParameters FilterParas;
        FilterParas.pListeningCondition = PolqaResults->m_ListeningCondition;
        FilterParas.cutOffFrequencyLow  = 3200;
        FilterParas.cutOffFrequencyHigh = 3400;

        pTA->FilterSignal(1, &FilterParas);
        pTA->FilterSignal(0, &FilterParas);

        TACheckTimeMatEval(mh, 1, &ClockCycles, &TimeDiff);
        if (TAInitData.mpLogFile)
            fprintf(TAInitData.mpLogFile, "\nTime required to set up the temporal
alignment: %.3lfs\n", TimeDiff);

        AddProcessingTime(PolqaResults, "TA Initialization", TimeDiff, ClockCycles);

        TACheckTimeMatInit(mh, 1);
        TArunIndex++;

        pResamplingResults[TArunIndex]->mAslFramelength = pDelayPara->Framesize;
        pResamplingResults[TArunIndex]->mAslFrames = pDelayPara->MaxModelFrames;

        rc = pTA->Run(Mode | 0x2 | 0x1, pResamplingResults[TArunIndex], TArunIndex);
        if (pResamplingResults[TArunIndex]->mNumFrames>pDelayPara->MaxModelFrames)
            DebugBreak();
```

```
        if (rc)
        {
            pDelayParaSaved[TArunIndex].LogFile = pDelayPara->LogFile;
            pDelayParaSaved[TArunIndex].mh = pDelayPara->mh;
            pDelayParaSaved[TArunIndex].MaxSigLen = pDelayPara->MaxSigLen;
            pDelayParaSaved[TArunIndex].pStartSampleUtterance =
pDelayPara->pStartSampleUtterance;
            pDelayParaSaved[TArunIndex].pStopSampleUtterance =
pDelayPara->pStopSampleUtterance;
            pDelayParaSaved[TArunIndex].pDelayUtterance = pDelayPara->pDelayUtterance;
            pDelayParaSaved[TArunIndex].pOriginalSamples =
pDelayPara->pOriginalSamples;
            pDelayParaSaved[TArunIndex].OriginalNumberOfSamples =
pDelayPara->OriginalNumberOfSamples;
            pDelayParaSaved[TArunIndex].pDistortedSamples =
pDelayPara->pDistortedSamples;
            pDelayParaSaved[TArunIndex].DistortedNumberOfSamples =
pDelayPara->DistortedNumberOfSamples;
            pDelayParaSaved[TArunIndex].Framesize = pDelayPara->Framesize;

            pDelayParaSaved[TArunIndex].AllocVectors(pDelayPara->MaxModelFrames);
            pDelayParaSaved[TArunIndex].FramesUsed = (((pDelayPara->MaxModelFrames) <
(pResamplingResults[TArunIndex]->mNumFrames)) ?
(pDelayPara->MaxModelFrames) :
(pResamplingResults[TArunIndex]->mNumFrames));

            pDelayParaSaved[TArunIndex].AslFramelength =
pResamplingResults[TArunIndex]->mAslFramelength;
            pDelayParaSaved[TArunIndex].AslFrames =
pResamplingResults[TArunIndex]->mAslFrames;
            pDelayParaSaved[TArunIndex].FramesUsed =
pResamplingResults[TArunIndex]->mNumFrames;
            for(int i=0; i<(((pDelayParaSaved[TArunIndex].AslFrames) <
(pDelayParaSaved[TArunIndex].MaxModelFrames)) ?
(pDelayParaSaved[TArunIndex].AslFrames) :
(pDelayParaSaved[TArunIndex].MaxModelFrames)); i++)
                pDelayParaSaved[TArunIndex].pAslActiveFrameFlags[i] =
(pResamplingResults[TArunIndex]->mpAslActiveFrameFlags[i] != 0);

            for(int i=0; i<pDelayParaSaved[TArunIndex].FramesUsed; i++)
                pDelayParaSaved[TArunIndex].pActiveFrameFlags[i] =
pResamplingResults[TArunIndex]->mpActiveFrameFlags[i];

            for(int i=0; i<pDelayParaSaved[TArunIndex].FramesUsed; i++)
                pDelayParaSaved[TArunIndex].pIgnoreFrameFlags[i] =
pResamplingResults[TArunIndex]->mpIgnoreFlags[i];

            if (pResamplingResults[TArunIndex]->mpReliability)
                matbCopy(pResamplingResults[TArunIndex]->mpReliability,
pDelayParaSaved[TArunIndex].pDelayReliability,
pDelayParaSaved[TArunIndex].FramesUsed);

            ShowProgress(0.0, "... Detect noise switching");
            pTA->GetNoiseSwitching(pDelayParaSaved[TArunIndex].pBGNSwitchingLevel,
pDelayParaSaved[TArunIndex].pNoiseDuringSpeechdB,
pDelayParaSaved[TArunIndex].pNoiseDuringSilencedB);

            pDelayParaSaved[TArunIndex].FirstRefSample =
pResamplingResults[TArunIndex]->FirstRefSample;
            pDelayParaSaved[TArunIndex].FirstDegSample =
pResamplingResults[TArunIndex]->FirstDegSample;

            {
                ShowProgress(0.0, "... Getting Pitch information");
                TACheckTimeMatInit(mh, 1);
                pDelayParaSaved[TArunIndex].PitchFreqRef = pTA->GetPitchVector(0, 0,
pDelayParaSaved[TArunIndex].pPitchVecOfRef,
pDelayParaSaved[TArunIndex].MaxModelFrames,
pDelayParaSaved[TArunIndex].Framesize);
                pDelayParaSaved[TArunIndex].PitchFreqDeg = pTA->GetPitchVector(1, 0,
pDelayParaSaved[TArunIndex].pPitchVecOfDeg,
pDelayParaSaved[TArunIndex].MaxModelFrames,
pDelayParaSaved[TArunIndex].Framesize);
                pDelayParaSaved[TArunIndex].PitchFrameSize = pTA->GetPitchFrameSize();
                TACheckTimeMatEval(mh, 1, &ClockCycles, &TimeDiff);
                if (pDelayPara->LogFile)
```

```
                    fprintf(pDelayPara->LogFile, "\nTime required to calculate pitch
information: %.3lfs\n", TimeDiff);
            }

            TACheckTimeMatEval(mh, 1, &ClockCycles, &TimeDiff);
            if (TAInitData.mpLogFile)
                fprintf(TAInitData.mpLogFile, "\nTime spent in pTA->Run() function:
%.3lfs\n", TimeDiff);
            TACheckTimeMatInit(mh, 1);
            AddProcessingTime(PolqaResults, "TA Run", TimeDiff, ClockCycles);

            AddProcessingTime(PolqaResults, "TA BeforeInitalDelay",
pResamplingResults[TArunIndex]->mTimeDiffs[0], 0);
            AddProcessingTime(PolqaResults, "TA InitalDelay",
pResamplingResults[TArunIndex]->mTimeDiffs[1], 0);
            AddProcessingTime(PolqaResults, "TA CoarseAlignment",
pResamplingResults[TArunIndex]->mTimeDiffs[2], 0);
            AddProcessingTime(PolqaResults, "TA FineAlignment",
pResamplingResults[TArunIndex]->mTimeDiffs[3], 0);

            OTA_FLOAT SrcThreshold = 0.005;
            if (1 && TArunIndex < MAX_TA_RUNS-1 &&
pResamplingResults[TArunIndex]->mRelSamplerateDev > 0 &&
(pResamplingResults[TArunIndex]->mRelSamplerateDev > 1.0 + SrcThreshold ||
pResamplingResults[TArunIndex]->mRelSamplerateDev < 1.0 - SrcThreshold))
            {

                    ShowProgress(0.0, "... Resampling signals");
                    PolqaResults->m_ResamplingApplied = true;

                    XFLOAT Ratio = 1.0 + (1.0 -
pResamplingResults[TArunIndex]->mRelSamplerateDev);

                    if (Ratio<1.0)
                    {
                        if (TAInitData.mpLogFile)
                            fprintf(TAInitData.mpLogFile, "\n\n***** Downsampling
degraded signal by %.3lf%%\n\n",
(float)(100.0*(1.0-Ratio)));

                        matConvertSamplerate(pDegSig[0], numDegSamples, pDegSig[0],
pDelayPara->MaxSigLen, Ratio, &NewLen);

                        numDegSamples = NewLen;
                    }
                    else
                    {
                        if (TAInitData.mpLogFile)
                            fprintf(TAInitData.mpLogFile, "\n\n***** Downsampling
reference signal by %.3lf%%\n\n",
(float)(100.0*(Ratio-1.0)));

                        matConvertSamplerate(pRefSig[0], numRefSamples, pRefSig[0],
pDelayPara->MaxSigLen, 1.0/Ratio, &NewLen);

                        numRefSamples = NewLen;
                    }

            }
            else
            {
                Done = true;
            }
        }
    }   while (!Done && rc);

    if (rc)
    {
        for (int i = 0; i <= TArunIndex; i++)
            ;

        if (TArunIndex==0)
        {
            bestAlignmentIdx = 0;
            maxAvgReliability  = pResamplingResults[0]->mAvgReliability;
        }
```

```
        else
        {

            if (((abs(1.0 - pResamplingResults[1]->mRelSamplerateDev)) < (abs(1.0 -
pResamplingResults[0]->mRelSamplerateDev)) ||
(pResamplingResults[1]->mRelSamplerateDev == -1.0)) &&
(pResamplingResults[1]->mAvgReliability >
pResamplingResults[0]->mAvgReliability))

            {
                bestAlignmentIdx = 1;
                maxAvgReliability  = pResamplingResults[1]->mAvgReliability;
            }
            else
            {
                bestAlignmentIdx = 0;
                maxAvgReliability  = pResamplingResults[0]->mAvgReliability;
            }
        }

        ;

        *pDelayPara = pDelayParaSaved[bestAlignmentIdx];

        pDelayPara->SetUtteranceInfo(pResamplingResults[bestAlignmentIdx]->mNumUtteranc
es, pResamplingResults[bestAlignmentIdx]->mpStartSampleUtterance,
pResamplingResults[bestAlignmentIdx]->mpStopSampleUtterance,
pResamplingResults[bestAlignmentIdx]->mpDelayUtterance);

        OTA_FLOAT FinalRatioApplied=1.0;
        OTA_FLOAT FinalRatioMeasured=1.0;
        for (int i=0; i<bestAlignmentIdx; i++)
        {
            if (pResamplingResults[i]->mRelSamplerateDev>0)
                FinalRatioMeasured*=pResamplingResults[i]->mRelSamplerateDev;
        }
        FinalRatioApplied = FinalRatioMeasured;
        if (pResamplingResults[bestAlignmentIdx]->mRelSamplerateDev>0)
            FinalRatioMeasured*=pResamplingResults[bestAlignmentIdx]->mRelSamplerateDev
;

        PolqaResults->m_MeasuredSamplerate =
FinalRatioMeasured*PolqaResults->m_SampleFrequencyHz;
        PolqaResults->m_AppliedSamplerate  = FinalRatioApplied
*PolqaResults->m_SampleFrequencyHz;

        (*pTaResult)->CopyFrom(pResamplingResults[bestAlignmentIdx]);

        if (bestAlignmentIdx == TArunIndex)
        {
            matbCopy(pRefSig[0], pDelayPara->pOriginalSamples,  numRefSamples);
            pDelayPara->OriginalNumberOfSamples  = numRefSamples;
            matbCopy(pDegSig[0], pDelayPara->pDistortedSamples, numDegSamples);
            pDelayPara->DistortedNumberOfSamples = numDegSamples;
        }
        else if (bestAlignmentIdx != 0)
        {
            XFLOAT Ratio = 1.0;
            for (int i = 0; i < bestAlignmentIdx; i++)
            {

                {

                    Ratio = 1.0 + (1.0 - pResamplingResults[i]->mRelSamplerateDev);

                }
            }

            if (Ratio < 1.0)
            {
                matConvertSamplerate(pDelayPara->pDistortedSamples,
pDelayPara->DistortedNumberOfSamples,
                                     pDelayPara->pDistortedSamples,
pDelayPara->MaxSigLen, Ratio, &NewLen);

                pDelayPara->DistortedNumberOfSamples = NewLen;
```

```
            }
            else
            {
                matConvertSamplerate(pDelayPara->pOriginalSamples,
pDelayPara->OriginalNumberOfSamples,
                                     pDelayPara->pOriginalSamples,
pDelayPara->MaxSigLen, 1.0/Ratio, &NewLen);

                pDelayPara->OriginalNumberOfSamples = NewLen;
            }

        }

    }

    for (int i = 0; i < MAX_TA_RUNS; i++)
    {
        if (pResamplingResults[i]) delete pResamplingResults[i];
        pResamplingResults[i] = NULL;
    }
    for (int i = 0; i < 2; i++)
    {
        matFree(pRefSig[i]);
        matFree(pDegSig[i]);
        pRefSig[i] = pDegSig[i] = NULL;
    }

    *pDone = Done;

    if (rc)
        pDelayPara->Check();

    return rc;
}

bool DoCalculateDelayDegPlus(CDelayPara* pDelayPara, POLQA_RESULT_DATA* PolqaResults)
{
    bool rc=true;

    int i;
    long f;
    long ClockCycles=0;
    double TimeDiff=0;
    MAT_HANDLE mh = pDelayPara->mh;
    TACheckTimeMatInit(mh, 1);

    ShowProgress(0.0, "... Temporal alignment");

    ITempAlignment* pTA = CreateAlignment(TA_FOR_SPEECH);
    OTA_RESULT* pTaResult = new OTA_RESULT();

    bool Done;
    pDelayPara->PitchFreqRef = -1;
    pDelayPara->PitchFreqDeg = -1;
    do
    {
        if (0 && pDelayPara->LogFile)
        {
            fprintf(pDelayPara->LogFile, "\nDoCalculateDelayDeg() 1\n");
            OTA_FLOAT EnergyRef = matSum(pDelayPara->pOriginalSamples,
pDelayPara->OriginalNumberOfSamples);
            OTA_FLOAT EnergyDeg = matSum(pDelayPara->pDistortedSamples,
pDelayPara->DistortedNumberOfSamples);
            fprintf(pDelayPara->LogFile, "\tSample sum ref:\t%.15e\n", EnergyRef);
            fprintf(pDelayPara->LogFile, "\tSample sum deg:\t%.15e\n", EnergyDeg);
        }

        rc = DoAlignmentPlus(pDelayPara, PolqaResults, pTA, &pTaResult, 0x8, &Done);

        XFLOAT* pRefSig=pDelayPara->pOriginalSamples;
        XFLOAT* pDegSig=pDelayPara->pDistortedSamples;

        if (rc && Done)
        {
            TACheckTimeMatEval(mh, 1, &ClockCycles, &TimeDiff);
            if (pDelayPara->LogFile)
```

```
                fprintf(pDelayPara->LogFile, "\nTime between pTA->Run() and calculating
noise switching indicator: %.3lfs\n", TimeDiff);

            TACheckTimeMatInit(mh, 1);

            ConvertFrameRateOfVectors(pTaResult, pDelayPara, PolqaResults);

            ConvertActivityFlag(pDelayPara->LogFile, pTaResult, pDelayPara,
PolqaResults);
            AddProcessingTime(PolqaResults, "Vector conversion", TimeDiff,
ClockCycles);

            {

                AlignPitchVectors(pDelayPara, pDelayPara->pPitchVecOfRef,
pDelayPara->pPitchVecOfDeg);

                pDelayPara->PitchFreqDeg =
AveragePitchFrequencyFromPitchVector(pDelayPara,
pDelayPara->pPitchVecOfDeg);
                pDelayPara->PitchFreqRef =
AveragePitchFrequencyFromPitchVector(pDelayPara,
pDelayPara->pPitchVecOfRef);
                pDelayPara->PitchFrameSize = pTA->GetPitchFrameSize();

                TACheckTimeMatEval(mh, 1, &ClockCycles, &TimeDiff);
                if (pDelayPara->LogFile)
                    fprintf(pDelayPara->LogFile, "\nTime required to align pitch
information: %.3lfs\n", TimeDiff);
            }
        }

        if (rc && Done && pTaResult->mNumUtterances &&
abs(pTaResult->mpDelayUtterance[0]) >
0.8*(((pDelayPara->DistortedNumberOfSamples) <
(pDelayPara->OriginalNumberOfSamples)) ? (pDelayPara->DistortedNumberOfSamples)
: (pDelayPara->OriginalNumberOfSamples)))
        {
            ShowProgress(0.0, "... Checking for very long offsets of the first
utterance");
            TACheckTimeMatInit(mh, 1);
            long RequiredOffset=0;
            if (pTaResult->mpDelayUtterance[0]<0)
            {
                RequiredOffset = pTaResult->mpDelayUtterance[0];
                if (pDelayPara->LogFile)
                    fprintf(pDelayPara->LogFile, "\n\n***** Shifting degraded signal by
%ld samples left\n\n", RequiredOffset);
                unsigned long
NewLen=pDelayPara->DistortedNumberOfSamples+RequiredOffset;
                pDelayPara->DistortedNumberOfSamples = NewLen;
                for (i=0; i<NewLen; i++)
                    pDegSig[i] = pDegSig[i-RequiredOffset];
                for (i=0; i<pDelayPara->pDelayUtterance->GetSize(); i++)
                {
                    pDelayPara->pDelayUtterance->m_pData[i] -= RequiredOffset;
                    pDelayPara->pStartSampleUtterance->m_pData[i] += RequiredOffset;
                    pDelayPara->pStopSampleUtterance->m_pData[i] += RequiredOffset;
                }

                for (i=0; i<pDelayPara->pDelayUtterance->GetSize(); i++)
                {
                    if (pDelayPara->pStopSampleUtterance->m_pData[i]<0)
                    {
                        DeleteUtteranceFromVector(i, pDelayPara);
                        pTaResult->mNumUtterances--;
                    }
                    else if (pDelayPara->pStartSampleUtterance->m_pData[i]<0)
                    {
                        pDelayPara->pStartSampleUtterance->m_pData[i] = 0;
                    }
                }

                const int OffsetInFrames = (RequiredOffset+pDelayPara->Framesize/2) /
pDelayPara->Framesize;
                if (OffsetInFrames)
```

```
                {
                    for (i=0; i<pDelayPara->MaxModelFrames+OffsetInFrames; i++)
                    {
                        pDelayPara->pPitchVecOfRef[i] =
pDelayPara->pPitchVecOfRef[i-OffsetInFrames];
                        pDelayPara->pPitchVecOfDeg[i] =
pDelayPara->pPitchVecOfDeg[i-OffsetInFrames];
                        pDelayPara->pActiveFrameFlags[i] =
pDelayPara->pActiveFrameFlags[i-OffsetInFrames];
                        pDelayPara->pIgnoreFrameFlags[i] =
pDelayPara->pIgnoreFrameFlags[i-OffsetInFrames];
                    }
                    pDelayPara->MaxModelFrames += OffsetInFrames;

                    for (i=0; i<PolqaResults->m_NumberOfFrames+OffsetInFrames; i++)
                    {
                        PolqaResults->m_DelayReliabilityPerFrame[i] =
PolqaResults->m_DelayReliabilityPerFrame[i-OffsetInFrames];

                        PolqaResults->m_DelayPerFrame[i] =
PolqaResults->m_DelayPerFrame[i-OffsetInFrames] -
RequiredOffset;
                    }
                    PolqaResults->m_NumberOfFrames+=OffsetInFrames;
                }
            }
            else
            {
                RequiredOffset = -pTaResult->mpDelayUtterance[0];
                if (pDelayPara->LogFile)
                    fprintf(pDelayPara->LogFile, "\n\n***** Shifting original signal by
%ld samples left\n\n", RequiredOffset);
                unsigned long
NewLen=pDelayPara->OriginalNumberOfSamples+RequiredOffset;
                pDelayPara->OriginalNumberOfSamples = NewLen;
                for (i=0; i<NewLen; i++)
                    pRefSig[i] = pRefSig[i-RequiredOffset];
                for (i=0; i<pTaResult->mNumUtterances; i++)
                    (*pDelayPara->pDelayUtterance).m_pData[i] += RequiredOffset;

                for (i=0; i<pDelayPara->MaxModelFrames; i++)
                    PolqaResults->m_DelayPerFrame[i] += RequiredOffset;

                int OffsetInFrames = (RequiredOffset+pDelayPara->Framesize/2) /
pDelayPara->Framesize;
                pDelayPara->MaxModelFrames += OffsetInFrames;
            }
            TACheckTimeMatEval(mh, 1, &ClockCycles, &TimeDiff);
            if (pDelayPara->LogFile)
                fprintf(pDelayPara->LogFile, "\nTime required to fix very long offsets:
%.3lfs\n", TimeDiff);
        }

        if (rc && Done)
        {
            ShowProgress(0.0, "... Calculating the average delay");
            XFLOAT AvgDelay=0;
            for (f=0; f<pTaResult->mNumFrames; f++)
                AvgDelay += -pTaResult->mpDelay[f];
            AvgDelay /= (XFLOAT)(pTaResult->mNumFrames);

            PolqaResults->m_MinDelay=100000000.0f;
            PolqaResults->m_MaxDelay=-100000000.0f;
            for (f=0; f<pTaResult->mNumFrames; f++)
            {
                XFLOAT val=-pTaResult->mpDelay[f];
                if(val>PolqaResults->m_MaxDelay)PolqaResults->m_MaxDelay=val;
                if(val<PolqaResults->m_MinDelay)PolqaResults->m_MinDelay=val;
            }

            PolqaResults->m_GlobalDelay = AvgDelay/PolqaResults->m_SampleFrequencyHz;
            PolqaResults->m_CrudeDelay = AvgDelay;
            PolqaResults->m_SNRDegdB = pTaResult->mSNRDegdB;
            PolqaResults->m_SNRRefdB = pTaResult->mSNRRefdB;
            PolqaResults->m_SignalLevelRef = pTaResult->mSignalLevelRef;
            PolqaResults->m_SignalLevelDeg = pTaResult->mSignalLevelDeg;
```

```
            PolqaResults->m_NoiseLevelRef = pTaResult->mNoiseLevelRef;
            PolqaResults->m_NoiseLevelDeg = pTaResult->mNoiseLevelDeg;
            PolqaResults->m_NoiseThresholdRef = pTaResult->mNoiseThresholdRef;
            PolqaResults->m_NoiseThresholdDeg = pTaResult->mNoiseThresholdDeg;
        }

        pTA->Free();

    }   while (!Done && rc);

    if (rc) DumpAlignedFile(pDelayPara, "d:\\temp\\Aligned.OptInterface.pcm");

    delete pTaResult;
    pTA->Destroy();

    ShowProgress(0.0, "... Done with temporal alignment");

    ;
    return rc;
}

}
```