```
    typedef double XFLOAT;
    typedef double OTA_FLOAT;

    typedef double OTA_FLOAT;
    typedef MAT_DCplx OTA_CPLX;

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= 0.2000000029802322;

        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
{

void GetNormalizedCCF(MAT_HANDLE mh, const OTA_FLOAT*  srcA,  int lenA, const
OTA_FLOAT*  srcB, int lenB, OTA_FLOAT*  dst, int dstLen);
void GetNormalizedCCFHistogram(MAT_HANDLE mh, const OTA_FLOAT*  srcA,  int lenA, const
OTA_FLOAT*  srcB, int lenB, OTA_FLOAT*  dst, int dstLen, int HistoLen);
void GetNormalizedCCFPeakHistogram(MAT_HANDLE mh, const OTA_FLOAT*  srcA,  int lenA,
const OTA_FLOAT*  srcB, int lenB, OTA_FLOAT*  dst, int dstLen, int HistoLen);

extern FILE* pLogFile;

inline void NaNTest2(const OTA_FLOAT* Vec, const int Len)
{
    int i;
    for (i=0; i<Len; i++)
        assert(Vec[i]==Vec[i]);
}

void
{
}
```

```
void SmoothHistogramTriangular(MAT_HANDLE mh, OTA_FLOAT* pHistogram, int HistogramLen,
int KernelWidth)
{
    int i;
    int Center = KernelWidth / 2;
    KernelWidth = 2*Center+1;
    OTA_FLOAT* pKernel = matxMalloc(KernelWidth);
    for (i=1; i<Center; i++)
    {
        OTA_FLOAT NextVal = (OTA_FLOAT)i/(OTA_FLOAT)Center;
        pKernel[i] = NextVal;
        pKernel[KernelWidth-i-1] = NextVal;
    }
    pKernel[0] = pKernel[KernelWidth-1] = 0;
    pKernel[Center] = 1;

    matRunFIRFilter(mh, pHistogram, pHistogram, HistogramLen, pKernel, KernelWidth,
MAT_FIRDelayComp);

    matFree(pKernel);

}

inline void GetNormalizedCCFCore(MAT_HANDLE mh, int FFTlen, int order, const OTA_FLOAT*
srcA, const OTA_FLOAT* srcB, OTA_FLOAT* dst, int lenA, int lenB, int dstLen, OTA_FLOAT*
tempinA, OTA_FLOAT* tempinB, OTA_FLOAT* tempinC, OTA_CPLX* tempoutA, OTA_CPLX*
tempoutB, OTA_CPLX* tempoutC)
{
    for (int d=0; d<dstLen; d++)
        dst[d] += matPearsonCorrelation((OTA_FLOAT*)srcA+d, (OTA_FLOAT*)srcB, lenB);

    NaNTest(dst, dstLen);
}

void GetNormalizedCCF(MAT_HANDLE mh, const OTA_FLOAT*  srcA,  int lenA, const
OTA_FLOAT*  srcB, int lenB, OTA_FLOAT*  dst, int dstLen)
{
    int order=0;
    OTA_FLOAT* tempinA=NULL;
    OTA_FLOAT* tempinB=NULL;
    OTA_FLOAT* tempinC=NULL;
    OTA_FLOAT* tempintest=NULL;
    OTA_CPLX* tempoutA=NULL;
    OTA_CPLX* tempoutB=NULL;
    OTA_CPLX* tempoutC=NULL;

    int MinLen = dstLen+lenB;
    while (1<<order <= MinLen)
        order++;
    order++;
    int FFTlen = 1<<order;
    lenA = (((lenA) < (FFTlen/2)) ? (lenA) : (FFTlen/2));

    tempinA=(OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*FFTlen);
    tempinB=(OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*FFTlen);
    tempinC=(OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*(FFTlen+2));

    tempoutA= (OTA_CPLX*)matMalloc (sizeof(OTA_CPLX)*(FFTlen/2+1));
    tempoutB= (OTA_CPLX*)matMalloc (sizeof(OTA_CPLX)*(FFTlen/2+1));
    tempoutC= (OTA_CPLX*)matMalloc (sizeof(OTA_CPLX)*(FFTlen/2+1));

    matbSet(0.0, dst, dstLen);
    GetNormalizedCCFCore(mh, FFTlen, order, srcA, srcB, dst, lenA, lenB, dstLen,
tempinA, tempinB, tempinC, tempoutA, tempoutB, tempoutC);

    matFree(tempoutC);
    matFree(tempoutB);
    matFree(tempoutA);
    matFree(tempinB);
    matFree(tempinA);
    matFree(tempinC);

}

void GetNormalizedCCFHistogram(MAT_HANDLE mh, const OTA_FLOAT*  srcA,  int lenA, const
```

```
OTA_FLOAT*  srcB, int lenB, OTA_FLOAT*  dst, int dstLen, int HistoLen)
{
    int order=0;
    OTA_FLOAT* tempinA=NULL;
    OTA_FLOAT* tempinB=NULL;
    OTA_FLOAT* tempinC=NULL;
    OTA_FLOAT* tempintest=NULL;
    OTA_CPLX* tempoutA=NULL;
    OTA_CPLX* tempoutB=NULL;
    OTA_CPLX* tempoutC=NULL;

    int i;

    int MinLen = dstLen+lenB;
    while (1<<order <= MinLen)
        order++;
    order++;
    int FFTlen = 1<<order;
    lenA = (((lenA) < (FFTlen/2)) ? (lenA) : (FFTlen/2));

    tempinA=(OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*FFTlen);
    tempinB=(OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*FFTlen);
    tempinC=(OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*FFTlen+2);

    tempoutA= (OTA_CPLX*)matMalloc (sizeof(OTA_CPLX)*(FFTlen/2+1));
    tempoutB= (OTA_CPLX*)matMalloc (sizeof(OTA_CPLX)*(FFTlen/2+1));
    tempoutC= (OTA_CPLX*)matMalloc (sizeof(OTA_CPLX)*(FFTlen/2+1));

    matbSet(0.0, dst, dstLen);
    for (i=0; i<HistoLen; i++)
        GetNormalizedCCFCore(mh, FFTlen, order, srcA+i, srcB+i, dst, lenA, lenB,
dstLen, tempinA, tempinB, tempinC, tempoutA, tempoutB, tempoutC);

    for (i=0; i<dstLen; i++)
        dst[i] /= HistoLen;

    matFree(tempoutC);
    matFree(tempoutB);
    matFree(tempoutA);
    matFree(tempinB);
    matFree(tempinA);
    matFree(tempinC);

}

void GetNormalizedCCFPeakHistogram(MAT_HANDLE mh, const OTA_FLOAT*  srcA,  int lenA,
const OTA_FLOAT*  srcB, int lenB, OTA_FLOAT*  dst, int dstLen, int HistoLen)
{
    int order=0;
    OTA_FLOAT* Correl=NULL;
    OTA_FLOAT* tempinA=NULL;
    OTA_FLOAT* tempinB=NULL;
    OTA_FLOAT* tempinC=NULL;
    OTA_FLOAT* tempintest=NULL;
    OTA_CPLX* tempoutA=NULL;
    OTA_CPLX* tempoutB=NULL;
    OTA_CPLX* tempoutC=NULL;

    int i;

    int MinLen = dstLen+lenB;
    while (1<<order <= MinLen)
        order++;
    order++;
    int FFTlen = 1<<order;
    lenA = (((lenA) < (FFTlen/2)) ? (lenA) : (FFTlen/2));

    Correl=matxMalloc(dstLen);
    tempinA=(OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*FFTlen);
    tempinB=(OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*FFTlen);
    tempinC=(OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*FFTlen+2);

    tempoutA= (OTA_CPLX*)matMalloc (sizeof(OTA_CPLX)*(FFTlen/2+1));
    tempoutB= (OTA_CPLX*)matMalloc (sizeof(OTA_CPLX)*(FFTlen/2+1));
    tempoutC= (OTA_CPLX*)matMalloc (sizeof(OTA_CPLX)*(FFTlen/2+1));
```

```
    int MaxIndex=-1;
    for (i=0; i<HistoLen; i++)
    {
        matbSet(0.0, dst, dstLen);
        GetNormalizedCCFCore(mh, FFTlen, order, srcA+i, srcB+i, Correl, lenA, lenB,
dstLen, tempinA, tempinB, tempinC, tempoutA, tempoutB, tempoutC);
        OTA_FLOAT RMax=matMaxExt(Correl, dstLen, &MaxIndex);
        dst[MaxIndex] += RMax;
    }
    for (i=0; i<dstLen; i++)
        dst[i] /= HistoLen;

    matFree(tempoutC);
    matFree(tempoutB);
    matFree(tempoutA);
    matFree(tempinB);
    matFree(tempinA);
    matFree(tempinC);
    matFree(Correl);

}

int FindDelay(MAT_HANDLE mh, OTA_FLOAT* pA, int LenA, OTA_FLOAT* pB, int LenB, int
HistoLen, int HistoShift, int MaxDelay, OTA_FLOAT* pPearsonCorrelation, bool PlotMe)
{

    int FoundDelay = -1;

    if(LenA > 0 && LenB > 0)
    {
        OTA_FLOAT Correl=-1;
        if (MaxDelay==0) MaxDelay = (((0) > (LenB-LenA-(HistoLen*HistoShift))) ? (0) :
(LenB-LenA-(HistoLen*HistoShift)));
        int KernelWidth = (((8) < (MaxDelay/2)) ? (8) : (MaxDelay/2));
        OTA_FLOAT* pDest = (OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*MaxDelay);
        OTA_FLOAT* pHistogramRel = (OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*MaxDelay);

        int Offset = 0;
        matbSet(0.0, pHistogramRel, MaxDelay);

        for (int i=0; i<HistoLen; i++)
        {
            int MaxDelayUsed = MaxDelay;
            if (MaxDelayUsed>0)
            {

                for (int p=0; p<MaxDelay; p++)
                    pDest[p] = matPearsonCorrelation(pA, pB+p, LenA);

                Correl = matMaxExt(pDest, MaxDelayUsed, &FoundDelay);

                if(FoundDelay >= 0)
                    pHistogramRel[FoundDelay]+= Correl;
            }

            Offset += HistoShift;
        }
        if (HistoLen>1)
            SmoothHistogramTriangular(mh, pHistogramRel, MaxDelay, KernelWidth);

        matMaxExt(pHistogramRel, MaxDelay, &FoundDelay);

        if (pPearsonCorrelation)
        {
            int Len = (((LenA) < (LenB-FoundDelay)) ? (LenA) : (LenB-FoundDelay));
            if (Len>0)
                *pPearsonCorrelation = Correl = matPearsonCorrelation(pA,
pB+FoundDelay, Len);
            else
                *pPearsonCorrelation = 0;

        }

        matFree(pDest);
        matFree(pHistogramRel);
```

```
    }
    else
    {
        FoundDelay = 0;
        *pPearsonCorrelation = 0;
    }
    return FoundDelay;
}

int FindDelayStrict(MAT_HANDLE mh, OTA_FLOAT* pA, int LenA, OTA_FLOAT* pB, int LenB,
int HistoLen, int HistoShift, int MaxDelay, OTA_FLOAT* pPearsonCorrelation)
{
    int FoundDelay = -1;

    if(LenA > 0 && LenB > 0)
    {
        OTA_FLOAT Correl=-1;
        if (MaxDelay==0) MaxDelay = (((0) > (LenB-LenA)) ? (0) : (LenB-LenA));
        int KernelWidth = (((8) < (MaxDelay/2)) ? (8) : (MaxDelay/2));
        OTA_FLOAT* pDest = (OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*MaxDelay);
        OTA_FLOAT* pHistogramRel = (OTA_FLOAT*)matMalloc (sizeof (OTA_FLOAT)*MaxDelay);

        int Offset = 0;
        matbSet(0.0, pHistogramRel, MaxDelay);

        for (int i=0; i<HistoLen; i++)
        {
            int MaxDelayUsed = (((0) > ((((MaxDelay) < (LenB-LenA-Offset)) ? (MaxDelay)
: (LenB-LenA-Offset)))) ? (0) : ((((MaxDelay) < (LenB-LenA-Offset)) ?
(MaxDelay) : (LenB-LenA-Offset))));
            if (MaxDelayUsed>0)
            {
                matCrossCorr(mh, pA+Offset, LenA, pB+Offset, LenB, pDest, MaxDelayUsed,
0);
                Correl = matMaxExt(pDest, MaxDelayUsed, &FoundDelay);

                if(FoundDelay >= 0)
                    pHistogramRel[FoundDelay]+= Correl;
            }

            Offset += HistoShift;
        }

        if (HistoLen>1)
            SmoothHistogramTriangular(mh, pHistogramRel, MaxDelay, KernelWidth);

        matMaxExt(pHistogramRel, MaxDelay, &FoundDelay);

        if (pPearsonCorrelation)
        {
            int Len = (((LenA) < (LenB-FoundDelay)) ? (LenA) : (LenB-FoundDelay));
            if (Len>0)
                *pPearsonCorrelation = Correl = matPearsonCorrelation(pA,
pB+FoundDelay, Len);
            else
                *pPearsonCorrelation = 0;
        }

        matFree(pDest);
        matFree(pHistogramRel);

    }
    else
    {
        FoundDelay = 0;
        *pPearsonCorrelation = 0;
    }
    return FoundDelay;
}

void FindMaxCorrelation(MAT_HANDLE mh, OTA_FLOAT* data1, unsigned long length1,
OTA_FLOAT* data2, unsigned long length2, int low_lag, int high_lag, int* index,
OTA_FLOAT* maximum, OTA_FLOAT* CorrelationBuffer)
{
    unsigned long CorrLen = high_lag-low_lag;
```

```
    matbSet(0, CorrelationBuffer, CorrLen);
    GetNormalizedCCF(mh, data1, length1, data2, length2, CorrelationBuffer, CorrLen);
    *maximum=matMaxExt(CorrelationBuffer, CorrLen, index);

    *maximum = (((1.0) < (*maximum)) ? (1.0) : (*maximum));
    *maximum = (((-1.0) > (*maximum)) ? (-1.0) : (*maximum));

}

#pragma optimize( "", off )
inline OTA_FLOAT A(int RFrom, int RTo, int MaxDistance, OTA_FLOAT* FromCorrelation,
OTA_FLOAT* ToCorrelation, OTA_FLOAT Factor1, OTA_FLOAT Factor2)
{
    OTA_FLOAT Distance = abs((RTo)-(RFrom));
    if (Distance*Distance*Factor1>13.8)
        return -1.0e6;
    else
        return 1-exp(Distance*Distance*Factor1);
}

void ComputePenaltyTable(long Size, long Center, OTA_FLOAT WeightFactor, OTA_FLOAT*
pPenaltyTable)
{
    int d;
    for (d=0; d<Size; d++)
    {
        pPenaltyTable[d] = d-Center;
        if (d>Center)   pPenaltyTable[d]*=1;
        else            pPenaltyTable[d]*=1;
    }
    matbAbs1(pPenaltyTable, Size);
    matbSqr1(pPenaltyTable, Size);
    matbMpy1(WeightFactor, pPenaltyTable, Size);
    matbThresh1(pPenaltyTable, Size, 9.21034037197618273607196581873 75, MAT_GT);
    matbExp1(pPenaltyTable, Size);
    for (d=0; d<Size; d++)
    {
        pPenaltyTable[d] = 1e10*(1.0-pPenaltyTable[d]);
        pPenaltyTable[d] = floor(pPenaltyTable[d])/1e10;
    }

}

#pragma optimize( "", on )

bool Viterbi(OTA_FLOAT** pMatrix, int* pOffsetPerFrame, OTA_FLOAT* PenaltyWeightFactor,
int *pOptOffset, OTA_FLOAT* pReliability, long NumDegradedFrames, long NumRefFrames,
VITERBI_PARA* pPara)
{
    bool rc = true;

    OTA_FLOAT** P;
    int**   L;

    ;

    L = (int**)matMalloc2D(NumDegradedFrames, NumRefFrames * sizeof(int));
    P = (OTA_FLOAT**)matMalloc2D(NumDegradedFrames, NumRefFrames * sizeof(OTA_FLOAT));

    if (L && P)
    {
        for (int i=0; rc && i<NumDegradedFrames; i++)
        {
            if (!P[i]) rc = false;
            if (!L[i]) rc = false;
        }
    }
    else rc = false;

    OTA_FLOAT* pPenaltyTable = (OTA_FLOAT*)matMalloc((2*NumRefFrames+1) *
sizeof(OTA_FLOAT));
    ComputePenaltyTable(2*NumRefFrames+1, NumRefFrames,
pPara->ViterbiDistanceWeightFactor, pPenaltyTable);

    OTA_FLOAT *PathProb = (OTA_FLOAT*)matMalloc(NumRefFrames * sizeof(OTA_FLOAT));
```

```
    if (rc && PathProb && pPenaltyTable)
    {
        int d;

        for (d=0; d<NumDegradedFrames; d++)
        {
            int i;

            matbCopy(pMatrix[d], P[d], NumRefFrames);
            matbThresh1(P[d], NumRefFrames, 0.0, MAT_LT);
            matbThresh1(P[d], NumRefFrames, 0.999, MAT_GT);
            for (i=0; i<NumRefFrames; i++)
                P[d][i] = (1-P[d][i]);
            for (i=0; i<NumRefFrames; i++)
                P[d][i] = -log10(P[d][i]);

            OTA_FLOAT PMin = matMin(P[d], NumRefFrames);
            OTA_FLOAT PMax = matMax(P[d], NumRefFrames);
            if (PMin==PMax)
                P[d][NumRefFrames/2] = PMax + 0.1;

        }

        for (d=1; d<NumDegradedFrames; d++)
        {

            int LastMaxPos;
            OTA_FLOAT LastMax;

            if (pPara->UseRelDistance)
            {
                for (int r=0; r<NumRefFrames; r++)
                {

                    matbCopy(P[d-1], PathProb, NumRefFrames);
                    for (int rr=0; rr<NumRefFrames; rr++)
                    {
                        int TableIndex = ((((((r + pOffsetPerFrame[d]-rr+NumRefFrames)
< (2*NumRefFrames)) ? (r + pOffsetPerFrame[d]-rr+NumRefFrames)
: (2*NumRefFrames))) > (0)) ? ((((r +
pOffsetPerFrame[d]-rr+NumRefFrames) < (2*NumRefFrames)) ? (r +
pOffsetPerFrame[d]-rr+NumRefFrames) : (2*NumRefFrames))) :
(0));
                        PathProb[rr] = PathProb[rr] + pPenaltyTable[TableIndex] *
PenaltyWeightFactor[d];
                    }

                    OTA_FLOAT LastMax = matMaxExt(PathProb, NumRefFrames, &LastMaxPos);

                    L[d][r] = LastMaxPos;

                    P[d][r] = P[d][r] + LastMax;
                }
            }
            else
            {
                for (int r=0; r<NumRefFrames; r++)
                {

                    matbCopy(P[d-1], PathProb, NumRefFrames);
                    for (int rr=0; rr<NumRefFrames; rr++)
                    {
                        int TableIndex = ((((((r -rr+NumRefFrames) < (2*NumRefFrames))
? (r -rr+NumRefFrames) : (2*NumRefFrames))) > (0)) ? ((((r
-rr+NumRefFrames) < (2*NumRefFrames)) ? (r -rr+NumRefFrames) :
(2*NumRefFrames))) : (0));
                        PathProb[rr] = PathProb[rr] + pPenaltyTable[TableIndex] *
PenaltyWeightFactor[d];
                    }

                    LastMax = matMaxExt(PathProb, NumRefFrames, &LastMaxPos);

                    L[d][r] = LastMaxPos;

                    P[d][r] = P[d][r] + LastMax;
                }
```

```
            }
        }

        matMaxExt(P[NumDegradedFrames-1], NumRefFrames,
pOptOffset+NumDegradedFrames-1);

        for (d=NumDegradedFrames-1; d>0; d--)
        {
            int LastOpt = pOptOffset[d];
            pOptOffset[d-1] = L[d][LastOpt];
        }

        for (d=0; d<NumDegradedFrames; d++)
            pReliability[d] = pMatrix[d][pOptOffset[d]];
    }

    if (PathProb)
        matFree(PathProb);
    if (pPenaltyTable)
        matFree(pPenaltyTable);

    matFree2D((void**)P);
    matFree2D((void**)L);

    return rc;
}

OTA_FLOAT* matxMalloc(int len)
{
    if (sizeof(OTA_FLOAT)==sizeof(float))
        return (OTA_FLOAT*)matsMalloc(len);
    else
        return (OTA_FLOAT*)matdMalloc(len);
}

OTA_CPLX* matCplxMalloc(int len)
{
    if (sizeof(OTA_CPLX)==sizeof(MAT_SCplx))
        return (OTA_CPLX*)matcMalloc(len);
    else
        return (OTA_CPLX*)matzMalloc(len);
}

}
```