```
    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 CCorrelationMatrix::Free()
{
    if (mpCorrMatrix)
    {

    }

    if (mpNormMatrix)
    {

        mpNormMatrix = 0;
    }
}

OTA_FLOAT CorrelationWithHistogramAndWindow(OTA_FLOAT* pSearchThis, OTA_FLOAT*
pSearchHere, int Corrlen, int HistogramLen, int HistogramStep, OTA_FLOAT* pWindowed1,
OTA_FLOAT* pWindowed2, OTA_FLOAT* HannWin, OTA_FLOAT* pCorrNorm)
{
```

```
    OTA_FLOAT Histogram = 0;

    if(HannWin)
    {
        for (int i=0; i<HistogramLen; i++)
        {
            matbMpy3(HannWin, pSearchThis+i, pWindowed1, Corrlen);
            matbMpy3(HannWin, pSearchHere+i, pWindowed2, Corrlen);
            if (i==0)
                Histogram += matPearsonCorrelation2(pWindowed1, pWindowed2, Corrlen,
pCorrNorm);
            else
                Histogram += matPearsonCorrelation(pWindowed1, pWindowed2, Corrlen);
        }
    }
    else
    {
        for (int i=0; i<HistogramLen; i++)
        {
            matWinHann(pSearchThis+i, pWindowed1, Corrlen);
            matWinHann(pSearchHere+i, pWindowed2, Corrlen);
            if (i==0)
                Histogram += matPearsonCorrelation2(pWindowed1, pWindowed2, Corrlen,
pCorrNorm);
            else
                Histogram += matPearsonCorrelation(pWindowed1, pWindowed2, Corrlen);
        }
    }

    Histogram /= HistogramLen;

    if (Histogram>0.7)
    {
        OTA_FLOAT Factor = 1.5;
        Factor = Factor*Factor*Corrlen;
        Factor = 1000;
        OTA_FLOAT E1 = matDotProd(pWindowed1, pWindowed1, Corrlen);
        OTA_FLOAT E2 = matDotProd(pWindowed2, pWindowed2, Corrlen);
        if (E1>Factor*E2)
            Histogram = -0.001;
        if (E2>Factor*E1)
            Histogram = -0.001;
    }

    return Histogram;
}

typedef struct
{
    bool PlotThisFrame;
    int ThisFrameNum;
    int CurrentStepSize;
    int CurrentFeatureNum;
} PLOT_INFO;

double CalcCorrelationForDelayRange(OTA_FLOAT *mpCorrDelayVec, OTA_FLOAT
*mpCorrNormVec, OTA_FLOAT *pBuffer1, OTA_FLOAT *pBuffer2, OTA_FLOAT* HannWin,
CProcessData* pProcessData,
                                    OTA_FLOAT* pFVecRef, long ffFeatureVectorlengthRef,
OTA_FLOAT* pFVecDeg, long ffFeatureVectorlengthDeg,
                                    long ffLastRefStart, long ffNextRefStartIndex, long
ffLastDegStart, int ffDegIndex, int DelayLowIndex,
int DelayHighIndex, int &maxIndex, int LowLim, int
HighLim,
                                    PLOT_INFO* pPlotInfo)
{
    const int MaxHistogramLength = 16;
    double  MaxCorr = 0;
    int     MaxPos = 0;
    int     ffHistogramLen = 1;
    double  CorrNow;
    long    ffIndex;

    int PatternLength = pProcessData->mCoarseAlignCorrlen;

    for (int ffd = LowLim; ffd<HighLim; ffd++)
```

```
    {
        int ffHistogramOffset = 0;
        int ffHistogramStep = 1;

        CorrNow = 0;
        ffIndex = ffNextRefStartIndex + ffd;
        ffHistogramLen = (((ffLastRefStart-ffIndex) < (ffLastDegStart-ffDegIndex)) ?
(ffLastRefStart-ffIndex) : (ffLastDegStart-ffDegIndex));
        ffHistogramLen = (((ffHistogramLen) < (MaxHistogramLength*ffHistogramStep)) ?
(ffHistogramLen) : (MaxHistogramLength*ffHistogramStep));

        ffHistogramOffset = -ffHistogramLen/2;
        ffHistogramOffset = (((ffHistogramOffset) > (-ffDegIndex)) ?
(ffHistogramOffset) : (-ffDegIndex));
        ffHistogramOffset = (((ffHistogramOffset) > (-ffIndex)) ? (ffHistogramOffset) :
(-ffIndex));

        if
(ffDegIndex+ffHistogramLen+ffHistogramOffset+PatternLength>ffFeatureVectorlengt
hDeg ||
ffIndex+ffHistogramLen+ffHistogramOffset+PatternLength>ffFeatureVectorlengthRef
)
            OutputDebugString("DelaySeach.cpp: CCorrelationMatrix::CreateMatrix(),
unexpected condition!\n");

        ffHistogramLen /= ffHistogramStep;

        if (ffHistogramLen==0) ffHistogramLen=1;

        mpCorrDelayVec[ffd] = CorrNow =
CorrelationWithHistogramAndWindow(&pFVecDeg[ffDegIndex+ffHistogramOffset],
&pFVecRef[ffIndex+ffHistogramOffset], PatternLength, ffHistogramLen,
ffHistogramStep, pBuffer1, pBuffer2, HannWin, &mpCorrNormVec[ffd]);

    }

    for (int k=LowLim; k<DelayLowIndex; k++)
        mpCorrDelayVec[k] =0;
    for (int k=DelayHighIndex; k<HighLim; k++)
        mpCorrDelayVec[k] =0;
    MaxCorr = -1;
    for (int k=DelayLowIndex; k<DelayHighIndex; k++)
    {
        if (MaxCorr<mpCorrDelayVec[k])
        {
            MaxCorr = mpCorrDelayVec[k];
            MaxPos = k;
        }
    }

    maxIndex = MaxPos;
    return MaxCorr;
}

//Create the correlation matrix for one feature pair (one signal pair, one channel, one
feature)
//The structure is as follows:
//
// - This vector contains for each element of the underlying feature vectors
//   one vector with the correlation of all possible delay lags between mMinLowVarDelay
and mMaxHighVarDelay.
//
//   M[DegFrameIndex][RefFrameIndex] = Correlation(&FeatureVecDeg[DegFrameIndex],
&FeatureVecRef[RefFrameIndex], mProcessData.mCoarseAlignCorrlen)
//      with: DegFrameIndex being the index of any frame [0;FeatureVectorlengthDeg-1]
and
//            RefFrameIndex being the index of any frame [0;-mMinLowVarDelay +
mMaxHighVarDelay]
//               around the frame at the average delay which is found at index
-mMinLowVarDelay.
//        RefFrameIndex can also be interpreted as the delay for frame DegFrameIndex
relative to the average delay.
//
//
//Data are stored for each DegStep frames of the degraded signal only.
```

```
//Degstep is also used for reading pAvgDelayInFrames (which contains one value for
every DegStep frames) and
//the length of the result vector
//NOTE: local variables starting with "ff" are related to feature frames and others are
related to TA frames
bool CCorrelationMatrix::CreateMatrix(CFeature* pFeature, int Channel, int*
pActiveFrameFlags, CProcessData* pProcessData, int NumMacroFrames, int*
pSearchRangeLow, int* pSearchRangeHigh, OTA_FLOAT* pPitchVec, int DegStep, long*
pAvgDelayInFrames, int CurrentFeatureIndex)
{
    bool rc = true;

    mProcessData = *pProcessData;
    mpFeature = pFeature;
    mMacroFrameSize = mProcessData.mStepSize * DegStep;

    long ffFeatureVectorlengthRef = mpFeature->GetFVector(0, Channel)->mSize;
    long ffFeatureVectorlengthDeg = mpFeature->GetFVector(1, Channel)->mSize;

    long ffMaxFeatureVectorlength = (((ffFeatureVectorlengthRef) >
(ffFeatureVectorlengthDeg)) ? (ffFeatureVectorlengthRef) :
(ffFeatureVectorlengthDeg));
    mNumMacroFrames = NumMacroFrames;
    mMinLowVarDelay   = mProcessData.mMinLowVarDelay;
    mMaxHighVarDelay  = mProcessData.mMaxHighVarDelay;
    mCorrelationVectorlength = -mMinLowVarDelay + mMaxHighVarDelay + 1;

    if (mProcessData.mpLogFile)
    {
        fprintf(mProcessData.mpLogFile,
"CCorrelationMatrix::CreateMatrix(ffFeatureVectorlengthRef=%ld,
ffFeatureVectorlengthDeg=%ld, DegStep=%d)\n", ffFeatureVectorlengthRef,
ffFeatureVectorlengthDeg, DegStep);
        fprintf(mProcessData.mpLogFile, "\tAllocating correlation matrix of size %d x
%d\n", mNumMacroFrames, mCorrelationVectorlength);
        fflush(mProcessData.mpLogFile);
    }

    if (mNumMacroFrames<=0)
    {
        OutputDebugString("CCorrelationMatrix::CreateMatrix(): ERROR, FeatureVectorLen
is <= 0!\n");
        exit(1);
    }

    mpCorrMatrix = (OTA_FLOAT**)matMalloc2D(mNumMacroFrames, mCorrelationVectorlength *
sizeof(OTA_FLOAT));
    mpNormMatrix = (OTA_FLOAT**)matMalloc2D(mNumMacroFrames, mCorrelationVectorlength *
sizeof(OTA_FLOAT));
    rc = (mpCorrMatrix!=0);

    if (!rc || !mpNormMatrix)
    {
        OutputDebugString("CCorrelationMatrix::CreateMatrix(): ERROR, problem with
memory allocation for mpCorrMatrix!\n");
        Free();
    }

    if (rc)
    {
        long ffLastRefStart = ffFeatureVectorlengthRef -
mProcessData.mCoarseAlignCorrlen-1;
        long ffLastDegStart = ffFeatureVectorlengthDeg -
mProcessData.mCoarseAlignCorrlen-1;
        OTA_FLOAT* pFVecRef=0;
        OTA_FLOAT* pFVecDeg=0;

        if (mpFeature->GetNumSets()>1)
        {
            pFVecRef = mpFeature->GetFVector(0, Channel, 0)->mpVector;
            pFVecDeg = mpFeature->GetFVector(1, Channel, 0)->mpVector;
        }
        else
        {
            pFVecRef = mpFeature->GetFVector(0, Channel, 0)->mpVector;
            pFVecDeg = mpFeature->GetFVector(1, Channel, 0)->mpVector;
```

```
        }

        int ffWindowOffset = mProcessData.mCoarseAlignCorrlen / 2;

        OTA_FLOAT* pBuffer1 = (OTA_FLOAT*)matMalloc(mProcessData.mCoarseAlignCorrlen *
sizeof(OTA_FLOAT));
        OTA_FLOAT* pBuffer2 = (OTA_FLOAT*)matMalloc(mProcessData.mCoarseAlignCorrlen *
sizeof(OTA_FLOAT));

        OTA_FLOAT* HannWin = (OTA_FLOAT*)matMalloc(mProcessData.mCoarseAlignCorrlen *
sizeof(OTA_FLOAT));

        matbSet((OTA_FLOAT)1.0, pBuffer1, mProcessData.mCoarseAlignCorrlen);
        matWinHann(pBuffer1, HannWin, mProcessData.mCoarseAlignCorrlen);

        int MaxOptDelayIndex=mCorrelationVectorlength-1;
        int MinOptDelayIndex=0;

        static int PlotFrameNum = -275;
        PLOT_INFO PlotInfo;
        PlotInfo.PlotThisFrame = false;
        PlotInfo.ThisFrameNum = 0;
        PlotInfo.CurrentStepSize = mProcessData.mStepSize;
        PlotInfo.CurrentFeatureNum = CurrentFeatureIndex;

        OTA_FLOAT MaxCorr = 0;
        int MaxPos = 0;
        int SecondLastMaxPos = 0;
        for (int f=0; f<mNumMacroFrames; f++)
        {

            {
                int ffDegIndex = f*DegStep-ffWindowOffset;

                if (ffDegIndex<0) ffDegIndex += ffWindowOffset;

                long ffRefIndex = -9999999;
                int DelayIndexHigh = -9999999;
                int DelayIndexLow = -9999999;

                if (ffDegIndex<ffLastDegStart)
                {
                    //Calculate the indices for the feature frame vectors
                    //All vectors are allocated with the maximum length of the feature,
but not all data may be valid!
                    //The "real" best found delay for frame f should be between index
mMinLowVarDelay and mMaxHighVarDelay+1
                    //in the correlation vector of frame f, in the middle of the
vector.

                    ffRefIndex = ffDegIndex+mMinLowVarDelay+pAvgDelayInFrames[f];

                    int SearchRange = -mMinLowVarDelay + mMaxHighVarDelay + 1;

                    int AbsMaxDelayIndex = (((ffLastRefStart-ffRefIndex) <
(SearchRange)) ? (ffLastRefStart-ffRefIndex) : (SearchRange));

                    int AbsMinDelayIndex = (((0) > (-ffRefIndex)) ? (0) :
(-ffRefIndex));

                    int MaxDelayIndex = pSearchRangeHigh[f]-mMinLowVarDelay+1;

                    MaxDelayIndex = (((AbsMaxDelayIndex) < (MaxDelayIndex)) ?
(AbsMaxDelayIndex) : (MaxDelayIndex));
                    MaxDelayIndex = (((AbsMinDelayIndex) > (MaxDelayIndex)) ?
(AbsMinDelayIndex) : (MaxDelayIndex));

                    int MinDelayIndex = pSearchRangeLow[f]-mMinLowVarDelay;
                    MinDelayIndex = (((AbsMaxDelayIndex) < (MinDelayIndex)) ?
(AbsMaxDelayIndex) : (MinDelayIndex));
                    MinDelayIndex = (((AbsMinDelayIndex) > (MinDelayIndex)) ?
(AbsMinDelayIndex) : (MinDelayIndex));

                    DelayIndexHigh = (((MaxDelayIndex) < (MaxOptDelayIndex)) ?
(MaxDelayIndex) : (MaxOptDelayIndex));
                    DelayIndexHigh = (((AbsMaxDelayIndex) < (DelayIndexHigh)) ?
```

```
(AbsMaxDelayIndex) : (DelayIndexHigh));
                    DelayIndexHigh = (((AbsMinDelayIndex) > (DelayIndexHigh)) ?
(AbsMinDelayIndex) : (DelayIndexHigh));

                    DelayIndexLow = (((MinDelayIndex) > (MinOptDelayIndex)) ?
(MinDelayIndex) : (MinOptDelayIndex));
                    DelayIndexLow = (((AbsMaxDelayIndex) < (DelayIndexLow)) ?
(AbsMaxDelayIndex) : (DelayIndexLow));
                    DelayIndexLow = (((AbsMinDelayIndex) > (DelayIndexLow)) ?
(AbsMinDelayIndex) : (DelayIndexLow));

                    if (DelayIndexLow>DelayIndexHigh)
                        DelayIndexLow = DelayIndexHigh;

                    if (DelayIndexLow==DelayIndexHigh)
                    {
                        DelayIndexHigh++;

                        //Check the bounds. If those are exceeded, extend the search
range towards the lower end.

                    }

                    //Preset the entire vector with zero

                    int UsedLimitLow = DelayIndexLow;
                    int UsedLimitHigh = DelayIndexHigh;
                    if (DelayIndexLow<DelayIndexHigh && ffRefIndex+DelayIndexLow>=0)
                    {
                        PlotInfo.PlotThisFrame = (f==PlotFrameNum &&
mProcessData.mCurrentIteration==3);
                        PlotInfo.ThisFrameNum = f;
                        MaxCorr = CalcCorrelationForDelayRange(mpCorrMatrix[f],
mpNormMatrix[f], pBuffer1, pBuffer2, HannWin, pProcessData,
                                               pFVecRef, ffFeatureVectorlengthRef, pFVecDeg,
ffFeatureVectorlengthDeg,
                                               ffLastRefStart, ffRefIndex, ffLastDegStart,
ffDegIndex, DelayIndexLow, DelayIndexHigh,
MaxPos, AbsMinDelayIndex, AbsMaxDelayIndex,
&PlotInfo);

                        {

                            if (MaxCorr > 0.98 && SecondLastMaxPos==MaxPos)

                            {
                                MinOptDelayIndex = (((0) > (MaxPos-10)) ? (0) :
(MaxPos-10));
                                MaxOptDelayIndex = (((mCorrelationVectorlength) <
(MaxPos+10)) ? (mCorrelationVectorlength) :
(MaxPos+10));
                            }
                            else if (DelayIndexLow > MinDelayIndex || DelayIndexHigh <
MaxDelayIndex)
                            {
                                int TempMaxPos;
                                double TempMaxCorr;
                                TempMaxCorr =
CalcCorrelationForDelayRange(mpCorrMatrix[f],
mpNormMatrix[f], pBuffer1, pBuffer2, HannWin,
pProcessData,
                                                      pFVecRef, ffFeatureVectorlengthRef,
pFVecDeg, ffFeatureVectorlengthDeg,

                                                      ffLastRefStart, ffRefIndex, ffLastDegStart,
ffDegIndex, MinDelayIndex, DelayIndexHigh,
TempMaxPos, DelayIndexHigh, MaxDelayIndex,
&PlotInfo);

                                UsedLimitLow = MinDelayIndex;

                                if (TempMaxCorr>MaxCorr)
                                {
                                    MaxCorr = TempMaxCorr;
                                    MaxPos = TempMaxPos;
                                }
```

```
                                TempMaxCorr =
CalcCorrelationForDelayRange(mpCorrMatrix[f],
mpNormMatrix[f], pBuffer1, pBuffer2, HannWin,
pProcessData,
                                            pFVecRef, ffFeatureVectorlengthRef,
pFVecDeg, ffFeatureVectorlengthDeg,
                                            ffLastRefStart, ffRefIndex, ffLastDegStart,
ffDegIndex, DelayIndexHigh, MaxDelayIndex,
TempMaxPos, DelayIndexHigh, MaxDelayIndex,
&PlotInfo);
                                UsedLimitHigh = MaxDelayIndex;

                                if (TempMaxCorr>MaxCorr)
                                {
                                    MaxCorr = TempMaxCorr;
                                    MaxPos = TempMaxPos;
                                }

                                //Reset search range for next frame

                            }
                        }
                        SecondLastMaxPos = MaxPos;
                    }

                }
                else
                {
                    //Set inactive frames to all zero

                }

            }

            if (!pActiveFrameFlags[f])

            {
                matbSet(0.0, mpCorrMatrix[f], mCorrelationVectorlength);
                matbSet(1.0, mpNormMatrix[f], mCorrelationVectorlength);

                MinOptDelayIndex = 0;
                MaxOptDelayIndex = mCorrelationVectorlength;
            }
        }
        matFree(pBuffer1);
        matFree(pBuffer2);
        matFree(HannWin);

    }

    mpSearchRangeLow = pSearchRangeLow;
    mpSearchRangeHigh = pSearchRangeHigh;
    return rc;
}

void CCorrelationMatrix::Print(FILE* pLogFile)
{

}

}
```