```
    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
{

extern CFeature* CreateFDFeature();
extern CFeature* CreateEnergyFeature();
extern CFeature* CreateStdDevFeature();
extern CFeature* CreateGeoAvgFeature();
extern CFeature* CreatePitchFeature();
extern CFeature* CreateEnvelopeFeature();

int VitDebugInt=0;
extern FILE* pLogFile;

int CSpeechFeatureList::CreateListOfFeatureModules(CFeature** mpFeatures, int
MaxFeatures, OTA_FLIST_TYPE ListType)
{

    int NumFeatures=0;

    mpFeatures[NumFeatures++] = CreateEnergyFeature();
    mpFeatures[NumFeatures++] = CreateFDFeature();
    mpFeatures[NumFeatures++] = CreateEnergyFeature();
    mpFeatures[NumFeatures++] = CreateFDFeature();
    mpFeatures[NumFeatures++] = CreateStdDevFeature();
    mpFeatures[NumFeatures++] = CreateGeoAvgFeature();
    mpFeatures[NumFeatures++] = CreateEnvelopeFeature();

    if (ListType==OTA_FLTYPE_INITIAL_SEARCH || ListType==OTA_FLTYPE_COARSE_ALIGN)
        return 2;
    else
        return 1;

}

CTASignal* CSpeechFeatureList::GetCopyOfSignals(CTASignal* pSignals, int NumSignals)
{
    CAudioSignal* pNewSigs = new CAudioSignal[NumSignals];
    for (int s=0; s<NumSignals; s++)
        pNewSigs[s] = ((CAudioSignal*)pSignals)[s];
    return (CTASignal*)pNewSigs;
}

//Combine the information from all correlation matrices and
//potentially feature vectors into one matrix which is stored for feature 0, left
channel.
//We mainly take the energy into account to modify correlations which are based on
frames
//with very low energy in the degraded signal.
//Assumption: the energy feature was calculated as the feature #1
//NOTE: StartFrame is relative to the frame size used for the correlation matrix,
//which may differ from the frame size of the feature vectors!
bool CSpeechDelaySearch::CombineMatricesAndFeatures(int StartFrame, int DegStep,
CCAIntermediateResults* pCAIntermediate)
{

            return CombineMatricesAndFeaturesV1(StartFrame, DegStep, pCAIntermediate);

}

#pragma region FEATURE_SELECTION_V2x

#pragma endregion

#pragma region CombineMatricesAndFeaturesV1

bool CSpeechDelaySearch::CombineMatricesAndFeaturesV1(int StartFrame, int DegStep,
CCAIntermediateResults* pCAIntermediate)
{

    int* FrameWithLastValidDelay = pCAIntermediate->pFrameWithLastValidDelay;
    int* pActiveFrameFlags = pCAIntermediate->pActiveFrameFlags;
    long* DelayVec = pCAIntermediate->pDelayVec;

    OTA_FLOAT* pMaxCorrelations = pCAIntermediate->pMaxCorrelations;
    int* pMaxPositions = pCAIntermediate->pMaxPositions;
    int* pFeatureUsed = pCAIntermediate->pFeatureUsed;
    int* pSelectionMethodUsed = pCAIntermediate->pSelectionMethodUsed;

    unsigned int i;
    long DegFrames;
    int DelayFrames;
    int firstActiveFrameIdx = -1;
    OTA_FLOAT** ppMatrix = GetPointerToMatrix(0, 0, &DegFrames, &DelayFrames);

    int ZeroDelayOffset = -mProcessData.mMinLowVarDelay;
    assert(ZeroDelayOffset==DelayFrames / 2);
    int NumFeatures = mpFeatureList->mNumFeatures;
    int NumSignals = mProcessData.mNumSignals;
    for (i=0; i<(unsigned int)DegFrames; i++)
        FrameWithLastValidDelay[i] = i;

    OTA_FLOAT LowEnergyThreshold = 1e23;

    long Len1;
    int Len2;

    pSelectionMethodUsed[0] = -99;
    pFeatureUsed[0] = -99;
    OTA_FLOAT MaxCorrelationSum=0;
    OTA_FLOAT MaxCorrelationCount = 0.0;
    for (long Deg=1; Deg<DegFrames; Deg++)
    {
        bool Done = false;

        pSelectionMethodUsed[Deg] = -99;
        pFeatureUsed[Deg] = -99;

        int LastValidFrame = FrameWithLastValidDelay[Deg-1];

        int LastValidMaxPos;
        OTA_FLOAT LasValidMaxVal = matMaxExt(ppMatrix[LastValidFrame], DelayFrames,
&LastValidMaxPos);

        int LastValidDelay = DelayVec[LastValidFrame] + LastValidMaxPos -
ZeroDelayOffset;

        int OffsetForConstDelay = (((0) > ((((DelayFrames-1) <
(LastValidDelay-DelayVec[Deg]+ZeroDelayOffset)) ? (DelayFrames-1) :
(LastValidDelay-DelayVec[Deg]+ZeroDelayOffset)))) ? (0) : ((((DelayFrames-1) <
(LastValidDelay-DelayVec[Deg]+ZeroDelayOffset)) ? (DelayFrames-1) :
(LastValidDelay-DelayVec[Deg]+ZeroDelayOffset))));

        //Skip inactive frames
        if (1)
        {
            if (!pActiveFrameFlags[Deg])
            {
                DelayVec[Deg] = DelayVec[LastValidFrame];
                matbCopy(ppMatrix[LastValidFrame], ppMatrix[Deg], DelayFrames);
                if (Deg)
                    FrameWithLastValidDelay[Deg] = FrameWithLastValidDelay[Deg-1];
                Done = true;

            }
        }

        //Use the feature with the best correlation, if the difference between the
correlations is significant.
        OTA_FLOAT CurrentMaxVal=ppMatrix[Deg][0];
        int CurrentMaxPos = 0;
        if (1 && !Done && mProcessData.mStepSize>1)
        {
            for (int f=0; f<NumFeatures; f++)
            {
                int Channels = mpFeatureList->mpFeatures[f]->mChannels;
                for (int c=0; c<Channels; c++)
                {
                    int Pos;
                    OTA_FLOAT** ppTestMatrix = GetPointerToMatrix(f, c, &Len1, &Len2);
                    OTA_FLOAT MaxR = matMaxExt(ppTestMatrix[Deg], Len2, &Pos);
                    if (MaxR-CurrentMaxVal>0.0 && MaxR>0.7)
                    {
                        matbCopy(ppTestMatrix[Deg], ppMatrix[Deg], Len2);
                        CurrentMaxVal = MaxR;
                        CurrentMaxPos = Pos;
                    }

                }
            }

        }

        //If we reached the start of a new active section, the position of the peak
        //must be copied to 50% of the previous (inactive) frames as well.
        if (1 && !Done && pActiveFrameFlags[Deg] && !pActiveFrameFlags[Deg-1])
        {

            if(firstActiveFrameIdx == -1)
                firstActiveFrameIdx = Deg;

            int Start = Deg - (Deg-FrameWithLastValidDelay[Deg-1])/2;

            if (Start != Deg)
            {

                const int nrMaxAvgFrames = 10;

                matbZero(ppMatrix[Deg-1], DelayFrames);

                DelayVec[Deg-1] = 0;

                int actualNrAvgFrames = 0;
                for(int af = Deg; af < (((Deg + nrMaxAvgFrames) < (DegFrames)) ? (Deg + 
nrMaxAvgFrames) : (DegFrames)); af++)
                {
                    if(pActiveFrameFlags[af])
                    {
                        matbAdd2(ppMatrix[af], ppMatrix[Deg-1], DelayFrames);
                        DelayVec[Deg-1] += DelayVec[af];
                        actualNrAvgFrames++;
                    }
                }
                matbMpy1(1.0/(OTA_FLOAT)actualNrAvgFrames, ppMatrix[Deg-1], 
DelayFrames);
                DelayVec[Deg-1] /= actualNrAvgFrames;

                for (;Start<Deg-1; Start++)
                {
                    DelayVec[Start] = DelayVec[Deg];
                    matbCopy(ppMatrix[Deg-1], ppMatrix[Start], DelayFrames);
                }

            }
        }

    }

    if (0 && pLogFile)
    {
        ;
        for (int i=0; i<DegFrames; i++)
            ;
        ;
    }

    //Do not allow any delay changes before the start frame
    if (1)
    {

        int i;

        int StartMax;
        OTA_FLOAT MaxVal = matMaxExt(ppMatrix[StartFrame], DelayFrames, &StartMax);

        for (i=0; i<StartFrame && StartFrame<0.5*DegFrames; i++)
        {
            DelayVec[i] = DelayVec[StartFrame];
            matbCopy(ppMatrix[StartFrame], ppMatrix[i], DelayFrames);
        }

        for (int i=0; i<DegFrames; i++)
            pMaxCorrelations[i] = matMaxExt(ppMatrix[i], DelayFrames, pMaxPositions+i);
    }

    return true;
}

#pragma endregion

void CSpeechDelaySearch::CleanupPath(CCAIntermediateResults* pCAIntermediate, long* 
DelayVec, long DelayVecLen, int* FrameWithLastValidDelay, int DegStep)
{
    int i;

    //Eliminate strong sporadic delay changes which are reverted within a short period

    int LargeChange = MSecondsToSamples(2);

    for (i=1; i<DelayVecLen; i++)
    {
        int k;

        if (pCAIntermediate->pActiveFrameFlags[i-1] && abs((int)(DelayVec[i] - 
DelayVec[i-1])) > LargeChange)

        {
            int ChangeStart = i;
            bool Found = false;
            for (k=i; k<i+MSecondsToFrames(300)/DegStep && !Found && k<DelayVecLen; 
k++)

                if (abs((int)(DelayVec[i-1]-DelayVec[k]))<LargeChange/2)

                    Found=true;

            k--;

            int ChangeEnd = k;
            if (Found)
            {
                for (;i<k; i++)
                    DelayVec[i] = DelayVec[k];
                if (mProcessData.mpLogFile)
                    ;
            }
        }
    }

    //Eliminate major delay changes which occure for a period which is of the same 
magnitude as the delay change

    int LargeChange2 = MSecondsToFrames(50);
    int LargeChange3InMF = LargeChange2/DegStep;

    for (i=1; i<DelayVecLen; i++)
    {
        int k;
        int DelayDiff1 = DelayVec[i] - DelayVec[i-1];
        if (abs((int)(DelayDiff1)) > LargeChange2)
        {
            int ChangeStart = i;
            bool Found = false;

            bool ChangeIsPositive = ((int)(DelayVec[i] - DelayVec[i-1])) > 0;
            for (k=i+1; k<DelayVecLen && !Found; k++)
            {
                int DelayDiff2 = (int)(DelayVec[k] - DelayVec[k-1]);

                if (abs(DelayDiff1+DelayDiff2)<1)

                    Found=true;
            }

            int ChangeEnd = k;

            if (Found && (ChangeEnd-ChangeStart)<LargeChange3InMF)

            {
                if (mProcessData.mpLogFile)
                    ;
                for (k=i; k<ChangeEnd; k++)
                    DelayVec[k] = DelayVec[ChangeStart-1];
            }
        }
    }

    //For invalid segments use the delay following the segment for the
    //second half of the invalid segment
    for (i=1; i<DelayVecLen; i++)
    {
        int k=0;
        for (; i<DelayVecLen && FrameWithLastValidDelay[i]!=i; i++);

        if (FrameWithLastValidDelay[i-1]!=i-1)
        {
            if (i!=DelayVecLen)
            {
                int HalfSectionLen = (i-FrameWithLastValidDelay[i-1])/2;

                for (k=FrameWithLastValidDelay[i-1]+1; k<i-HalfSectionLen; k++)
                    DelayVec[k] = DelayVec[FrameWithLastValidDelay[i-1]];

                for (; k<i; k++)
                    DelayVec[k] = DelayVec[i];
            }
            else
            {
                for (k=FrameWithLastValidDelay[i-1]+1; k<i; k++)
                    DelayVec[k] = DelayVec[FrameWithLastValidDelay[i-1]];
            }
        }
    }

}

void CalcOneLineOfCorrMatrix(OTA_FLOAT **pCorrmatrix, const OTA_FLOAT *pRefSig, const
OTA_FLOAT *pShiftedDegSig, int RefStart, int RefEnd, int SearchStart, int SearchEnd,
int CurLine, int CorrLen, OTA_FLOAT *BufferRef, OTA_FLOAT *BufferDeg)
{
    int d, NextRefStart;

    OTA_FLOAT refStdDev, degStdDev;
    OTA_FLOAT refMean, degMean;

    degMean = matdMeanStdDev(pShiftedDegSig, CorrLen, &degStdDev);
    matbAdd4(pShiftedDegSig, -degMean, BufferDeg, CorrLen);

    if (degStdDev > 0)
    {
        for (d = SearchStart, NextRefStart = RefStart; d < SearchEnd && NextRefStart <
RefEnd; d++, NextRefStart++)
        {

            OTA_FLOAT const *pShiftedRefSig = pRefSig + NextRefStart;

            refMean = matdMeanStdDev(pShiftedRefSig, CorrLen, &refStdDev);
            matbAdd4(pShiftedRefSig, -refMean, BufferRef, CorrLen);

            matbMpy2(BufferDeg, BufferRef, CorrLen);
            OTA_FLOAT XY = matSum(BufferRef, CorrLen);

            if (refStdDev > 0.0)
                pCorrmatrix[CurLine][d] = XY / ((OTA_FLOAT)(CorrLen - 1)* refStdDev *
degStdDev);
            else
                pCorrmatrix[CurLine][d] = 0.0;

            pCorrmatrix[CurLine][d] = ((((((-1.0) > (pCorrmatrix[CurLine][d])) ? (-1.0)
: (pCorrmatrix[CurLine][d]))) < (1.0)) ? ((((-1.0) >
(pCorrmatrix[CurLine][d])) ? (-1.0) : (pCorrmatrix[CurLine][d]))) : (1.0));
        }
    }
    else
        for (d = SearchStart, NextRefStart = RefStart; d < SearchEnd && NextRefStart <
RefEnd; d++, NextRefStart++)
            pCorrmatrix[CurLine][d] = 0.0;

    if (d < SearchEnd)
        matbZero(pCorrmatrix[CurLine] + d, SearchEnd - d);
}

//This method is called by Run() after the iterative coarse alignment.
//The delay vector mpDelayInSamplesPerFrame contains for each frame the delay with
//with an accuracy of +/-mProcessData.mMinStepsize
bool CSpeechDelaySearch::FineAlign(CFAIntermediateResults* pFAIntermediate, CTASignal
**pSignals, long *pNumFrames, int Stepsize, int CorrLen, unsigned long Flags)
{
    bool rc=true;
    int* pConstDelayMarker = pFAIntermediate->pConstDelayMarker;
    int* pOptOffset = pFAIntermediate->pOptOffset;
    int* pActiveFrameFlags = pFAIntermediate->pActiveFrameFlags;
    long* pDelayInSamplesPerFrame = pFAIntermediate->pDelayVec;
    OTA_FLOAT* pReliabilityPerFrame = pFAIntermediate->pReliabilityPerFrame;
    int* pSearchRangePerMacroFrameLow = pFAIntermediate->pSearchRangeLow;
    int* pSearchRangePerMacroFrameHigh = pFAIntermediate->pSearchRangeHigh;

    mProcessData.Init(1, 1.0);

    return rc;
}

bool CSpeechDelaySearch::FineAlign(CFAIntermediateResults* pFAIntermediate, CTASignal
**pSignals, CActiveFrameDetection* pActiveFrameDetection, long*
pDelayInSamplesPerFrame, OTA_FLOAT* pReliabilityPerFrame, long *pNumFrames, int
Stepsize, int SearchRange, int CorrLen, unsigned long Flags)
{
    bool rc=true;
    int f, i;

    int NumFrames = *pNumFrames;

    ;
    //Calculate a framewise short (+/-SearchRange) CCF between the input waveforms
(left channels only),
    //and search the maximum. The position of the maximum is the required lag around
    OTA_FLOAT* pRefSigRaw = ((CAudioSignal*)pSignals[0])->mpData[0];
    OTA_FLOAT* pDegSigRaw = ((CAudioSignal*)pSignals[1])->mpData[0];
    OTA_FLOAT* pRefSig =
(OTA_FLOAT*)matMalloc(((CAudioSignal*)pSignals[0])->mSignalLength *
sizeof(OTA_FLOAT));
    OTA_FLOAT* pDegSig =
(OTA_FLOAT*)matMalloc(((CAudioSignal*)pSignals[1])->mSignalLength *
sizeof(OTA_FLOAT));
    matbCopy(pRefSigRaw, pRefSig, ((CAudioSignal*)pSignals[0])->mSignalLength);
    matbCopy(pDegSigRaw, pDegSig, ((CAudioSignal*)pSignals[1])->mSignalLength);

    ;
    matbSqr1(pRefSig, ((CAudioSignal*)pSignals[0])->mSignalLength);
    matbSqr1(pDegSig, ((CAudioSignal*)pSignals[1])->mSignalLength);

    int Next=0;
    int Step = 2*SearchRange+1;

    OTA_FLOAT *tempBuffer1 = (OTA_FLOAT*)matMalloc(CorrLen * sizeof(OTA_FLOAT));
    OTA_FLOAT *tempBuffer2 = (OTA_FLOAT*)matMalloc(CorrLen * sizeof(OTA_FLOAT));

    ;

    OTA_FLOAT** pCorrmatrix = (OTA_FLOAT**)matMalloc2D(NumFrames, Step *
sizeof(OTA_FLOAT));

    OTA_FLOAT* pCenterEnergy = (OTA_FLOAT*)matMalloc(NumFrames * sizeof(OTA_FLOAT));
    int* pOptOffset = (int*)matMalloc(NumFrames * sizeof(int));

    long LastStartRef = ((CAudioSignal*)pSignals[0])->mSignalLength-CorrLen;
    int NumDegFrames = (((NumFrames) <
((((CAudioSignal*)pSignals[1])->mSignalLength-CorrLen) / Stepsize)) ? (NumFrames) :
((((CAudioSignal*)pSignals[1])->mSignalLength-CorrLen) / Stepsize));

    ;

    int LastValidFrame = 0;
    int LastValidSectionStart = 0;
    bool IsValidSection = false;
    int* pActiveFrameFlags = new int[NumFrames];
    pActiveFrameDetection->GetActiveFrameFlags(1, 0, Stepsize, pActiveFrameFlags,
NumFrames);

    for (f=0; f<NumFrames; f++)
    {
        if (f < NumDegFrames)
        {
            int NextRefStart = f*Stepsize+pDelayInSamplesPerFrame[f]-SearchRange;
            int d = 0;
            if (NextRefStart < 0)
            {
                d = (((-NextRefStart) < (2*SearchRange+1)) ? (-NextRefStart) :
```

```
(2*SearchRange+1));
                matbZero(pCorrmatrix[f], d);
                NextRefStart = 0;
            }

            //Skip inactive frames

            if (!pActiveFrameFlags[f] && LastValidFrame>0)
            {

                OTA_FLOAT MaxRs[7+1];
                int MaxPositions[7+1];
                int BestFrame;
                int FirstFrameSearched = (((LastValidFrame-7) >
(LastValidSectionStart)) ? (LastValidFrame-7) :
(LastValidSectionStart));
                int FramesSearched = LastValidFrame-FirstFrameSearched+1;
                for (i=FirstFrameSearched; i<=LastValidFrame; i++)
                    MaxRs[i-FirstFrameSearched] = matMaxExt(pCorrmatrix[i],
2*SearchRange+1, MaxPositions+i-FirstFrameSearched);
                OTA_FLOAT RequiredR = matMaxExt(MaxRs, FramesSearched, &BestFrame);
                BestFrame += FirstFrameSearched;

                CalcOneLineOfCorrMatrix(pCorrmatrix, pRefSig, pDegSig + f*Stepsize,
NextRefStart, LastStartRef, d, 2*SearchRange+1, f, CorrLen,
tempBuffer1, tempBuffer2);
                OTA_FLOAT MaxR = matMax(pCorrmatrix[f], 2*SearchRange+1);

                if (RequiredR<0.4 || RequiredR>MaxR+0.1)
                {
                    pDelayInSamplesPerFrame[f] = pDelayInSamplesPerFrame[BestFrame];
                    matbCopy(pCorrmatrix[BestFrame], pCorrmatrix[f], 2*SearchRange+1);
                }

                IsValidSection=false;

            }
            else if (!pActiveFrameFlags[f] && LastValidFrame<=0)
            {
                matbSet(0.0, pCorrmatrix[f], 2*SearchRange+1);
                IsValidSection=false;

            }
            else
            {
                CalcOneLineOfCorrMatrix(pCorrmatrix, pRefSig, pDegSig + f*Stepsize,
NextRefStart, LastStartRef, d, 2*SearchRange+1, f, CorrLen,
tempBuffer1, tempBuffer2);
                LastValidFrame = f;
                if (!IsValidSection)
                {
                    IsValidSection=true;
                    LastValidSectionStart = f;
                }
            }

        }
        else
            matbZero(pCorrmatrix[f], 2*SearchRange+1);

        pCenterEnergy[f] = matSum(pDegSig+f*Stepsize, CorrLen/2) / CorrLen/2;

    }

    ;

    bool IsLeadIn=true;
    for (f=1; f<NumFrames; f++)
    {
        if (pActiveFrameFlags[f] && !pActiveFrameFlags[f-1])
        {
            int BestFrame = f;
            OTA_FLOAT BestR = matMax(pCorrmatrix[f], 2*SearchRange+1);
            for (int i=f+1; i<f+4 && i<NumFrames; i++)
            {
                if (pActiveFrameFlags[i])
                {
                    OTA_FLOAT MaxR = matMax(pCorrmatrix[i], 2*SearchRange+1);
                    if (MaxR>BestR) {BestR = MaxR; BestFrame=i;}
                }
            }

            int Start=f-1;
            if (!IsLeadIn)
            {
                for (; Start>=0 && !pActiveFrameFlags[Start]; Start--);
                Start = f - (f-Start)/2;
            }
            else Start = 0;

            for (int i=Start; i<=f; i++)
            {
                {
                    pDelayInSamplesPerFrame[i] = pDelayInSamplesPerFrame[BestFrame];
                    matbCopy(pCorrmatrix[BestFrame], pCorrmatrix[i], 2*SearchRange+1);
                }
            }

            IsLeadIn = false;
        }
    }

    matFree(tempBuffer1);
    matFree(tempBuffer2);

    OTA_FLOAT* PenaltyWeightFactor = (OTA_FLOAT*)matMalloc(NumFrames * 
sizeof(OTA_FLOAT));
    matbSet(1.0, PenaltyWeightFactor, NumFrames);

    //Filter out drops of the correlation
    if (1)
    {
        int Len2 = 2*SearchRange+1;
        OTA_FLOAT LastBestR=0;
        OTA_FLOAT SecondLastBestR=0;
        for (long Deg=1; Deg<NumFrames; Deg++)
        {
            OTA_FLOAT** ppMatrix = pCorrmatrix;
            OTA_FLOAT BestR = matMax(ppMatrix[Deg], Len2);
            OTA_FLOAT CurrentBestR = BestR;

            if (BestR<mProcessData.mP.mSpeechTAPara.FineAlignLowEnergyCorrel && 
pCenterEnergy[Deg]<mProcessData.mP.mSpeechTAPara.FineAlignLowEnergyThresh)
            {
                matbZero(ppMatrix[Deg], Len2);
                ppMatrix[Deg][Len2/2] = 1.0;
                BestR = LastBestR;

            }

            if (BestR<mProcessData.mP.mSpeechTAPara.FineAlignShortDropOfCorrelR && 
LastBestR>mProcessData.mP.mSpeechTAPara.FineAlignShortDropOfCorrelRLastBest
)
            {
                matbZero(ppMatrix[Deg], Len2);
                ppMatrix[Deg][Len2/2] = 1.0;
                BestR = LastBestR;

            }

            if (1 && LastBestR<SecondLastBestR-0.2 &&  LastBestR<BestR-0.2)
            {
                if (Deg>1)
                {
                    matbZero(ppMatrix[Deg], Len2);
                    ppMatrix[Deg][Len2/2] = 1.0;
                    BestR = LastBestR;

                }
            }

            if (1)
            {

                if (pActiveFrameFlags[Deg] && CurrentBestR > 0.95)

                {
                    PenaltyWeightFactor[Deg]=0.25;

                }
            }

            SecondLastBestR = LastBestR;
            LastBestR = BestR;
        }
    }

    //Get a vector with the relative delay of each frame to the previous one based on 
the
    //results from the last delay calculation.
    int* pRelativeDelayPerFrame = (int*)matMalloc(NumFrames * sizeof(int));
    pRelativeDelayPerFrame[0] = 0;
    for (f=1; f<NumFrames; f++)
        pRelativeDelayPerFrame[f] = pDelayInSamplesPerFrame[f] - 
pDelayInSamplesPerFrame[f-1];

    VITERBI_PARA VP;
    VP.ViterbiDistanceWeightFactor = 
mProcessData.mP.mSpeechTAPara.ViterbiDistanceWeightFactor;
    VP.UseRelDistance = true;

    Viterbi(pCorrmatrix, pRelativeDelayPerFrame, PenaltyWeightFactor, pOptOffset, 
pReliabilityPerFrame, NumFrames, 2*SearchRange+1, &VP);

    matFree(pRelativeDelayPerFrame);

    for (i=0; i<3; i++)
    {
        for (f=1; f<NumFrames-1; f++)
        {
            //Check if the previous frame had a better correlation than this frame
            if (pCorrmatrix[f][pOptOffset[f]]<pCorrmatrix[f-1][pOptOffset[f-1]]-0.2)
            {
                if (pCorrmatrix[f][pOptOffset[f-1]] > pCorrmatrix[f][pOptOffset[f]])
                {
                    pOptOffset          [f] = pOptOffset[f-1];
                    pReliabilityPerFrame[f] = pCorrmatrix[f][pOptOffset[f-1]];
                }
            }
            //Check if the next frame has a better correlation than this frame
            else if 
(pCorrmatrix[f][pOptOffset[f]]<pCorrmatrix[f+1][pOptOffset[f+1]]-0.2)
            {
                if (pCorrmatrix[f][pOptOffset[f+1]] > pCorrmatrix[f][pOptOffset[f]])
                {
                    pOptOffset          [f] = pOptOffset[f+1];
                    pReliabilityPerFrame[f] = pCorrmatrix[f][pOptOffset[f+1]];
                }
            }
        }
    }

    for (f=0; f<NumFrames; f++)
        pDelayInSamplesPerFrame[f] += pOptOffset[f] - SearchRange;

        if (mProcessData.mpLogFile)
        {
            double Avg=0;
            double AvgCnt=0;
            ;
            ;
            for (f=0; f<NumFrames; f++)
                ;
            ;
        }

    matFree2D((void**)pCorrmatrix);
    pCorrmatrix = 0;
    if (pCenterEnergy)
        matFree(pCenterEnergy);
    if (pRefSig)
        matFree(pRefSig);
    if (pDegSig)
        matFree(pDegSig);
    if (pOptOffset)
        matFree(pOptOffset);
    if(PenaltyWeightFactor)
        matFree(PenaltyWeightFactor);

    delete[] pActiveFrameFlags;

    return rc;
}

bool CSpeechTempAlignment::Init(CProcessData* pProcessData)
{
    bool rc = true;
    int i=0;

    SPEECH_WINDOW_PARA* pPara=pProcessData->mP.mSpeechTAPara.Win;
    while(pPara->Samplerate<pProcessData->mSamplerate)
        pPara++;

    pProcessData->mDelayFineAlignCorrlen = pPara->mDelayFineAlignCorrlen;
    pProcessData->mSRDetectFineAlignCorrlen = pPara->mSRDetectFineAlignCorrlen;
    for (i=0; i<8; i++)
    {
        pProcessData->mpCoarseAlignCorrlen[i] = pPara->CoarseAlignCorrlen[i];
        pProcessData->mpWindowSize[i]      = pPara->WindowSize[i];
        pProcessData->mpOverlap[i]      = pPara->WindowSize[i] / 2;
        pProcessData->mpViterbiDistanceWeightFactor[i] =
pPara->pViterbiDistanceWeightFactor[i];
    }

    pProcessData->mpOverlap[0]      = 0;
    pProcessData->mpOverlap[1]      = 0;
    pProcessData->mpOverlap[2]      = 0;
    pProcessData->Init(0, 1.0);

    rc = CTempAlignment::Init(pProcessData, new CSpeechDelaySearch, new
CSpeechActiveFrameDetection);

    if (rc)
        for (i=0; i<2; i++)
            mppSignals[i] = new CAudioSignal;

    mpFeatureList  = new CSpeechFeatureList;
    mpFeatureList2 = new CSpeechFeatureList;

    return rc;
}

bool CSpeechTempAlignment::Run(unsigned long Control, OTA_RESULT* pResult, int
TArunIndex)
{
    bool rc=true;

    if (rc) rc = CTempAlignment::Run(Control, pResult, TArunIndex);

    return rc;
};

//Determine some reasonable limits for the delay searches.
//All results are measured in ms and describe the delay of the ref signal.
//- The ref file has at least 40% activity and consists of two sentences.
//- The total amount of silence is split into at least two sections (typically three)
//- Assume that no more than 50% of the silence fall before the start or after the end
of the file
//- Assume that the speech part is not cut off at either end due to the delay
//- The ref file may by z samples longer than the ref file. These may all be  silence
//Flen,r = length of the ref file
//This results in a max delay of the ref signal of:
//  D1 = (Flen,ref*0.4*0.5)  or, more exact for the first utterance: D1 = RefStart
```

```
//and:
//  D2 = -(Flen,ref*0.4 + z) * 0.5
void CSpeechTempAlignment::GetDelayLimits(int RefStartSample, int* pMaxDelayPos, int*
MaxDelayNeg)
{
    *pMaxDelayPos = (((SamplesToMSeconds(RefStartSample)) <
(mProcessData.mMaxStaticDelayInMs)) ? (SamplesToMSeconds(RefStartSample)) :
(mProcessData.mMaxStaticDelayInMs));
    int z = mppSignals[1]->mSignalLength-mppSignals[0]->mSignalLength;
    *MaxDelayNeg = (((SamplesToMSeconds(-(mppSignals[0]->mSignalLength*0.2 + z) * 0.8))
> (mProcessData.mMinStaticDelayInMs)) ?
(SamplesToMSeconds(-(mppSignals[0]->mSignalLength*0.2 + z) * 0.8)) :
(mProcessData.mMinStaticDelayInMs));

    ;
}

//Calculate a noise switching indicator

OTA_FLOAT CSpeechTempAlignment::EvaluateNoiseOfOneSection(OTA_FLOAT* NoiseLevelSpeech,
OTA_FLOAT* NoiseLevelSilence, OTA_FLOAT* NoiseLevelAfter, int NumChecks, SECTION* Sec,
SECTION* SecNext, int Signal)
{
    OTA_FLOAT Switching = 0.0;
    int i;
    mProcessData.Init(1, 1);
    mpActiveFrameDetection->Init(&mProcessData);
    mpActiveFrameDetection->Start(mppSignals);

    SEGMENT Segment;

    Segment.Start = Sec->Start;
    Segment.End = Sec->End;
    *NoiseLevelSpeech = mpActiveFrameDetection->GetLevelBelowThreshold(&Segment,
Signal, 0);

    Segment.Start = Sec->End;
    Segment.End = SecNext->Start;
    *NoiseLevelSilence = mpActiveFrameDetection->GetLevelBelowThreshold(&Segment,
Signal, 0);

    int Offset=-50;
    for (i=0; i<NumChecks; i++)
    {
        Segment.Start = Sec->End + MSecondsToSamples(Offset+i*10);
        Segment.End = Segment.Start + MSecondsToSamples(Offset+i*10+10);
        NoiseLevelAfter[i] = mpActiveFrameDetection->GetLevelBelowThreshold(&Segment,
Signal, 0);
    }

    *NoiseLevelSpeech = 10*log10(*NoiseLevelSpeech+0.0000001);
    *NoiseLevelSilence = 10*log10(*NoiseLevelSilence+0.0000001);
    for (i=0; i<NumChecks; i++)
        NoiseLevelAfter[i] = 10*log10(NoiseLevelAfter[i]+0.0000001);

    OTA_FLOAT AvgE=0;
    OTA_FLOAT AvgESilence=0;
    int CountSilence=0;
    for (i=0; i<NumChecks; i++)
        AvgE += NoiseLevelAfter[i];
    AvgE /= (OTA_FLOAT)NumChecks;
    AvgE = 0.9* AvgE;
    for (i=0; i<NumChecks; i++)
    {
        if (NoiseLevelAfter[i]>AvgE)
        {
            NoiseLevelAfter[i] = AvgE;
        }
        else
        {
            AvgESilence += NoiseLevelAfter[i];
            CountSilence++;
        }
    }
    AvgESilence /= ((OTA_FLOAT)CountSilence +0.0000001);
    if (!CountSilence) AvgESilence = AvgE;
    if (AvgESilence<AvgE &&
        NoiseLevelAfter[0] >= AvgE-0.001 &&
        NoiseLevelAfter[NumChecks-1] >= AvgE-0.001 &&
        *NoiseLevelSilence-*NoiseLevelSpeech>0.0)
        Switching = AvgE-AvgESilence;

    return Switching;
}

/*  • All following operations are performed on the downsampled envelopes of the
degraded signal.
    • NoiseLevelSpeech:  Based on the noise threshold determined by the VAD, all frames
during an utterance which fall below the threshold are averaged.
    • NoiseLevelSilence:  Based on the noise threshold determined by the VAD, all
frames between two utterances which fall below the threshold are averaged.
    • Create a vector NoiseLevel which does the above averaging for 30 10ms intervals
starting from 50ms before the end of an utterance.
    • Get the average of NoiseLevel  (=Avg1).
    • Set all elements of NoiseLevel  which exceed 0.9*the average to the value of the
value of the average.
    • Get the average of all other elements of NoiseLevel  (=Avg2)
    • Get 10*log10() of all the above averages and energies.

    • if (NoiseLevel [0]>=Avg1  &&  NoiseLevel [29]>=Avg1  &&
NoiselevelSilence-NoiseLevelSpeech>15)
        SwitchingIndicator = Avg1 -Avg2
    else
        SwitchingIndicator = 0;

    returns: SwitchingIndicator (0 if no switching detected)
    Sets: NoiseLevelSpeech, NoiseLevelSilence (-1 if not set)
*/

void CSpeechTempAlignment::GetNoiseSwitching(OTA_FLOAT* pBGNSwitchingLevel, OTA_FLOAT*
pNoiseLevelSpeech, OTA_FLOAT* pNoiseLevelSilence)
{
    int i;
    OTA_FLOAT IndicatorRef=0;
    OTA_FLOAT IndicatorDeg=0;
    OTA_FLOAT NoiseLevelSpeechRef=-1;
    OTA_FLOAT NoiseLevelSilenceRef=-1;
    OTA_FLOAT NoiseLevelSpeechDeg=-1;
    OTA_FLOAT NoiseLevelSilenceDeg=-1;
    OTA_FLOAT NoiseLevelAfterRef[30];
    OTA_FLOAT NoiseLevelAfterDeg[30];

    if (mNumReparsePoints>1)
    {
        IndicatorRef = EvaluateNoiseOfOneSection(&NoiseLevelSpeechRef,
&NoiseLevelSilenceRef, NoiseLevelAfterRef, 30, &mpReparsePoints[0].Ref,
&mpReparsePoints[1].Ref, 0);
        IndicatorDeg = EvaluateNoiseOfOneSection(&NoiseLevelSpeechDeg,
&NoiseLevelSilenceDeg, NoiseLevelAfterDeg, 30, &mpReparsePoints[0].Deg,
&mpReparsePoints[1].Deg, 1);

    }

    pBGNSwitchingLevel[0] = IndicatorRef;
    pBGNSwitchingLevel[1] = IndicatorDeg;
    pNoiseLevelSpeech[0] = NoiseLevelSpeechRef;
    pNoiseLevelSpeech[1] = NoiseLevelSpeechDeg;
    pNoiseLevelSilence[0] = NoiseLevelSilenceRef;
    pNoiseLevelSilence[1] = NoiseLevelSilenceDeg;
}

//Calculate the average pitch frequency of of one channel.
OTA_FLOAT CSpeechTempAlignment::GetPitchFreq(CTASignal **pSignals, int Signal, int
Channel)
{
    OTA_FLOAT AvgPitch=0;
    int PitchStart=0;
    CPitchBase Pitch(mpSmartBufferPool);
    Pitch.GetPitchVector(&mProcessData,
((CAudioSignal*)pSignals[Signal])->mpData[Channel], 0,
((CAudioSignal*)pSignals[Signal])->mSignalLength, 0, 0, 0, &AvgPitch, &PitchStart);
```

```
    return (float)AvgPitch;
}

//Get the individual pitch values for all frames on a given frame scale. Values <=0
mark unvoiced frames.
OTA_FLOAT CSpeechTempAlignment::GetPitchVector(CTASignal **pSignals, int Signal, int
Channel, OTA_FLOAT* pVector, int NumFrames, int SamplesPerFrame)
{
    OTA_FLOAT AvgPitch=0;
    CPitchBase Pitch(mpSmartBufferPool);

    OTA_FLOAT* pPitchVec;
    int NumPitchFrames;
    int PitchStartOffset=0;
    Pitch.GetPitchVector(&mProcessData,
((CAudioSignal*)pSignals[Signal])->mpData[Channel], 0,
((CAudioSignal*)pSignals[Signal])->mSignalLength, &pPitchVec, &NumPitchFrames,
&mpResults->mPitchFrameSize, &AvgPitch, &PitchStartOffset);

    int StartOffsetExternal = 0;
    int StartFrameTA = 0;

    if (Signal==1 && mStartOffset<0)
    {
        StartOffsetExternal = (-mStartOffset+SamplesPerFrame/2) / SamplesPerFrame;
        StartOffsetExternal = (((0) > (StartOffsetExternal)) ? (0) :
(StartOffsetExternal));
        StartFrameTA = (((0) >
((-mStartOffset+mpResults->mPitchFrameSize/2)/mpResults->mPitchFrameSize)) ?
(0) :
((-mStartOffset+mpResults->mPitchFrameSize/2)/mpResults->mPitchFrameSize));
    }
    else if (Signal==0 && mStartOffset>0)
    {
        StartOffsetExternal = (mStartOffset+SamplesPerFrame/2) / SamplesPerFrame;
        StartFrameTA = (mStartOffset+mpResults->mPitchFrameSize,
mpResults->mPitchFrameSize/2)/mpResults->mPitchFrameSize;
    }

    int FrameCount = 0;
    int StartSample = SamplesPerFrame/2;
    int LastStart = (NumFrames-2-StartOffsetExternal)*SamplesPerFrame;
    while (StartSample<LastStart)
    {
        pVector[FrameCount+StartOffsetExternal] = pPitchVec[(((0) >
((((NumPitchFrames-1) < ((StartSample+mpResults->mPitchFrameSize/2 -
PitchStartOffset)/mpResults->mPitchFrameSize)) ? (NumPitchFrames-1) :
((StartSample+mpResults->mPitchFrameSize/2 -
PitchStartOffset)/mpResults->mPitchFrameSize)))) ? (0) : ((((NumPitchFrames-1)
< ((StartSample+mpResults->mPitchFrameSize/2 -
PitchStartOffset)/mpResults->mPitchFrameSize)) ? (NumPitchFrames-1) :
((StartSample+mpResults->mPitchFrameSize/2 -
PitchStartOffset)/mpResults->mPitchFrameSize))))];
        StartSample += SamplesPerFrame;
        FrameCount++;
    }
    pVector[NumFrames-2]=0;
    pVector[NumFrames-1]=0;

    OTA_FLOAT StartPitch = 0;
    for (int i=0; i<StartOffsetExternal && i<NumFrames; i++)
        pVector[i] = StartPitch;

    matFree(pPitchVec);

    return (float)AvgPitch;
}

void CSpeechTempAlignment::GetFilterCharacteristics(FilteringParameters *FilterParams)
{

    FilterParams->disturbedEnergyQuotient = 1.0;

}

}
```