```
    typedef double XFLOAT
    typedef double OTA_FLOAT

    typedef double OTA_FLOAT
    typedef MAT_DCplx OTA_CPLX


{

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,  86,  34,  0,  0},
                        {-1,    -1,  -1,  15,  12,  0,  0}},
                    {16000,  64, 64,
                        {256,  512, 256, 128,  64,  0},
                        {-1,    -1,  -1,  63,  33,  0},
                        {-1,    -1,  -1,  13,  10,  0}},
                    {48000, 256, 256,
```

```
                        {512, 1024, 512, 512, 128,  0},
                        {-1,    -1,  -1, 115,  61,  0},
                        {-1,    -1,  -1,  17,  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
}
}


{

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
```

```
332
333             D = mP.mSpeechTAPara.ViterbiDistanceWeightFactorDist
334             D = D * mSamplerate / 1000
335             F =((float) log(1+0.5) / (D*D))
336             mP.mSpeechTAPara.ViterbiDistanceWeightFactor = F
337
338             return true
339         }
340
341         CParameters   mP
342 }
343
344 class SECTION
345 {
346     public:
347         int Start
348         int End
349         int Len() {return End-Start }
350         void CopyFrom(const SECTION &src)
351         {
352             this->Start = src.Start
353             this->End   = src.End
354         }
355 }
356
357 typedef struct OTA_RESULT
358 {
359     void CopyFrom(const OTA_RESULT* src)
360     {
361         mNumFrames           = src->mNumFrames
362         mStepsize            = src->mStepsize
363         mResolutionInSamples = src->mResolutionInSamples
364         if (src->mpDelay != NULL && mNumFrames > 0)
365         {
366             matFree(mpDelay)
367             mpDelay = (long*)matMalloc(mNumFrames * sizeof(long))
368             for (int i = 0  i < mNumFrames  i++)
369                 mpDelay[i] = src->mpDelay[i]
370         }
371         else
372         {
373             matFree(mpDelay)
374             mpDelay = NULL
375         }
376
377         if (src->mpReliability != NULL && mNumFrames > 0)
378         {
379             matFree(mpReliability)
380             mpReliability = (OTA_FLOAT*)matMalloc(mNumFrames * sizeof(OTA_FLOAT))
381             for (int i = 0  i < mNumFrames  i++)
382                 mpReliability[i] = src->mpReliability[i]
383         }
384         else
385         {
386             matFree(mpReliability)
387             mpReliability = NULL
388         }
389         mAvgReliability   = src->mAvgReliability
390         mRelSamplerateDev = src->mRelSamplerateDev
391
392         mNumUtterances = src->mNumUtterances
393         if (src->mpStartSampleUtterance != NULL && mNumUtterances > 0)
394         {
395             matFree(mpStartSampleUtterance)
396             mpStartSampleUtterance = (int*)matMalloc(mNumUtterances * sizeof(int))
397             for (int i = 0  i < mNumUtterances  i++)
398                 mpStartSampleUtterance[i] = src->mpStartSampleUtterance[i]
399         }
```

```
        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))
            mNumIngoreFlags = src->mNumIngoreFlags
            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
        }

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

        mAslFramelength = 0
        mAslFrames = 0
        mpAslActiveFrameFlags = NULL
        mAslFramesDeg = 0
        mpAslActiveFrameFlagsDeg = 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
        matFree(mpAslActiveFrameFlagsDeg)
        mpAslActiveFrameFlagsDeg = 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 mNumIngoreFlags
    int mAslFrames
    int mAslFramelength
    int *mpAslActiveFrameFlags
    int mAslFramesDeg
    int *mpAslActiveFrameFlagsDeg

    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,

}
```

```
669 ITempAlignment* CreateAlignment(AlignmentType Type)
670
671 }
672
673
674 {
675
676 extern CFeature* CreateFDFeature()
677 extern CFeature* CreateEnergyFeature()
678 extern CFeature* CreateStdDevFeature()
679 extern CFeature* CreateGeoAvgFeature()
680 extern CFeature* CreatePitchFeature()
681 extern CFeature* CreateEnvelopeFeature()
682 extern CFeature* CreateMFCCFeature()
683
684 int VitDebugInt=0
685 extern FILE* pLogFile
686
687 int CSpeechFeatureList::CreateListOfFeatureModules(CFeature** mpFeatures, int MaxFeatures,
    OTA_FLIST_TYPE ListType)
688 {
689
690     int NumFeatures=0
691
692     mpFeatures[NumFeatures++] = CreateEnergyFeature()
693     mpFeatures[NumFeatures++] = CreateFDFeature()
694     mpFeatures[NumFeatures++] = CreateEnergyFeature()
695     mpFeatures[NumFeatures++] = CreateFDFeature()
696     mpFeatures[NumFeatures++] = CreateStdDevFeature()
697     mpFeatures[NumFeatures++] = CreateGeoAvgFeature()
698     mpFeatures[NumFeatures++] = CreateEnvelopeFeature()
699
700     if (ListType==OTA_FLTYPE_INITIAL_SEARCH || ListType==OTA_FLTYPE_COARSE_ALIGN)
701         return 2
702     else
703         return 1
704
705 }
706
707 CTASignal* CSpeechFeatureList::GetCopyOfSignals(CTASignal* pSignals, int NumSignals)
708 {
709     CAudioSignal* pNewSigs = new CAudioSignal[NumSignals]
710     for (int s=0  s<NumSignals  s++)
711         pNewSigs[s] = ((CAudioSignal*)pSignals)[s]
712     return (CTASignal*)pNewSigs
713 }
714
715 //Combine the information from all correlation matrices and
716 //potentially feature vectors into one matrix which is stored for feature 0, left channel.
717 //We mainly take the energy into account to modify correlations which are based on frames
718 //with very low energy in the degraded signal.
719 //Assumption: the energy feature was calculated as the feature #1
720 //NOTE: StartFrame is relative to the frame size used for the correlation matrix,
721 //which may differ from the frame size of the feature vectors!
722 bool CSpeechDelaySearch::CombineMatricesAndFeatures(int StartFrame, int DegStep,
    CCAIntermediateResults* pCAIntermediate)
723 {
724
725             return CombineMatricesAndFeaturesV1(StartFrame, DegStep, pCAIntermediate)
726
727 }
728
729
730
731
732 bool CSpeechDelaySearch::CombineMatricesAndFeaturesV1(int StartFrame, int DegStep,
    CCAIntermediateResults* pCAIntermediate)
733 {
```

```
734
735     int* FrameWithLastValidDelay = pCAIntermediate->pFrameWithLastValidDelay
736     int* pActiveFrameFlags = pCAIntermediate->pActiveFrameFlags
737     long* DelayVec = pCAIntermediate->pDelayVec
738
739     OTA_FLOAT* pMaxCorrelations = pCAIntermediate->pMaxCorrelations
740     int* pMaxPositions = pCAIntermediate->pMaxPositions
741     int* pFeatureUsed = pCAIntermediate->pFeatureUsed
742     int* pSelectionMethodUsed = pCAIntermediate->pSelectionMethodUsed
743
744     unsigned int i
745     long DegFrames
746     int DelayFrames
747     int firstActiveFrameIdx = -1
748     OTA_FLOAT** ppMatrix = GetPointerToMatrix(0, 0, &DegFrames, &DelayFrames)
749
750     int ZeroDelayOffset = -mProcessData.mMinLowVarDelay
751     assert(ZeroDelayOffset==DelayFrames / 2)
752     int NumFeatures = mpFeatureList->mNumFeatures
753     int NumSignals = mProcessData.mNumSignals
754     for (i=0  i<(unsigned int)DegFrames  i++)
755         FrameWithLastValidDelay[i] = i
756
757     OTA_FLOAT LowEnergyThreshold = 1e23
758
759     long Len1
760     int Len2
761
762     pSelectionMethodUsed[0] = -99
763     pFeatureUsed[0] = -99
764
765     OTA_FLOAT MaxCorrelationSum=0
766     OTA_FLOAT MaxCorrelationCount = 0.0
767     for (long Deg=1  Deg<DegFrames  Deg++)
768     {
769         bool Done = false
770
771         pSelectionMethodUsed[Deg] = -99
772         pFeatureUsed[Deg] = -99
773
774         int LastValidFrame = FrameWithLastValidDelay[Deg-1]
775
776         int LastValidMaxPos
777         OTA_FLOAT LasValidMaxVal = matMaxExt(ppMatrix[LastValidFrame], DelayFrames,
    &LastValidMaxPos)
778
779         int LastValidDelay = DelayVec[LastValidFrame] + LastValidMaxPos - ZeroDelayOffset
780
781         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))))
782
783         //Skip inactive frames
784         if (1)
785         {
786             if (!pActiveFrameFlags[Deg])
787             {
788                 DelayVec[Deg] = DelayVec[LastValidFrame]
789                 matbCopy(ppMatrix[LastValidFrame], ppMatrix[Deg], DelayFrames)
790                 if (Deg)
791                     FrameWithLastValidDelay[Deg] = FrameWithLastValidDelay[Deg-1]
792                 Done = true
793
794             }
795         }
796
```

```
797         //Use the feature with the best correlation, if the difference between the correlations is
    significant.
798         OTA_FLOAT CurrentMaxVal=ppMatrix[Deg][0]
799         int CurrentMaxPos = 0
800         if (1 && !Done && mProcessData.mStepSize>1)
801         {
802             for (int f=0  f<NumFeatures  f++)
803             {
804                 int Channels = mpFeatureList->mpFeatures[f]->mChannels
805                 for (int c=0  c<Channels  c++)
806                 {
807                     int Pos
808                     OTA_FLOAT** ppTestMatrix = GetPointerToMatrix(f, c, &Len1, &Len2)
809                     OTA_FLOAT MaxR = matMaxExt(ppTestMatrix[Deg], Len2, &Pos)
810                     if (MaxR-CurrentMaxVal>0.0 && MaxR>0.7)
811                     {
812                         matbCopy(ppTestMatrix[Deg], ppMatrix[Deg], Len2)
813                         CurrentMaxVal = MaxR
814                         CurrentMaxPos = Pos
815                     }
816
817                 }
818             }
819
820         }
821
822         //If we reached the start of a new active section, the position of the peak
823         //must be copied to 50% of the previous (inactive) frames as well.
824         if (1 && !Done && pActiveFrameFlags[Deg] && !pActiveFrameFlags[Deg-1])
825         {
826
827             if(firstActiveFrameIdx == -1)
828                 firstActiveFrameIdx = Deg
829
830             int Start = Deg - (Deg-FrameWithLastValidDelay[Deg-1])/2
831
832             if (Start != Deg)
833             {
834
835                 const int nrMaxAvgFrames = 10
836
837                 matbZero(ppMatrix[Deg-1], DelayFrames)
838
839                 DelayVec[Deg-1] = 0
840
841                 int actualNrAvgFrames = 0
842                 for(int af = Deg  af < (((Deg + nrMaxAvgFrames) < (DegFrames)) ? (Deg +
    nrMaxAvgFrames) : (DegFrames))  af++)
843                 {
844                     if(pActiveFrameFlags[af])
845                     {
846                         matbAdd2(ppMatrix[af], ppMatrix[Deg-1], DelayFrames)
847                         DelayVec[Deg-1] += DelayVec[af]
848                         actualNrAvgFrames++
849                     }
850                 }
851                 matbMpy1(1.0/(OTA_FLOAT)actualNrAvgFrames, ppMatrix[Deg-1], DelayFrames)
852                 DelayVec[Deg-1] /= actualNrAvgFrames
853
854                 for ( Start<Deg-1  Start++)
855                 {
856                     DelayVec[Start] = DelayVec[Deg]
857                     matbCopy(ppMatrix[Deg-1], ppMatrix[Start], DelayFrames)
858                 }
859
860             }
861         }
862
```

```
863     }
864
865     if (0 && pLogFile)
866     {
867
868         for (int i=0  i<DegFrames  i++)
869
870
871     }
872
873     //Do not allow any delay changes before the start frame
874     if (1)
875     {
876
877         int i
878
879         int StartMax
880         OTA_FLOAT MaxVal = matMaxExt(ppMatrix[StartFrame], DelayFrames, &StartMax)
881
882         for (i=0  i<StartFrame && StartFrame<0.5*DegFrames  i++)
883         {
884             DelayVec[i] = DelayVec[StartFrame]
885             matbCopy(ppMatrix[StartFrame], ppMatrix[i], DelayFrames)
886         }
887
888         for (int i=0  i<DegFrames  i++)
889             pMaxCorrelations[i] = matMaxExt(ppMatrix[i], DelayFrames, pMaxPositions+i)
890     }
891
892     return true
893 }
894
895
896 void CSpeechDelaySearch::CleanupPath(CCAIntermediateResults* pCAIntermediate, long* DelayVec, long
    DelayVecLen, int* FrameWithLastValidDelay, int DegStep)
897 {
898     int i
899
900     //Eliminate strong sporadic delay changes which are reverted within a short period
901
902     int LargeChange = MSecondsToSamples(2)
903
904     for (i=1  i<DelayVecLen  i++)
905     {
906         int k
907
908         if (pCAIntermediate->pActiveFrameFlags[i-1] && abs((int)(DelayVec[i] - DelayVec[i-1])) >
    LargeChange)
909
910         {
911             int ChangeStart = i
912             bool Found = false
913             for (k=i  k<i+MSecondsToFrames(300)/DegStep && !Found && k<DelayVecLen  k++)
914
915                 if (abs((int)(DelayVec[i-1]-DelayVec[k]))<LargeChange/2)
916
917                     Found=true
918
919             k--
920
921             int ChangeEnd = k
922             if (Found)
923             {
924                 for ( i<k  i++)
925                     DelayVec[i] = DelayVec[k]
926                 if (mProcessData.mpLogFile)
927
928             }
```

```
        }
    }

    //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
}
```

```
1056 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)
1057 {
1058     bool rc=true
1059     int f, i
1060
1061     int NumFrames = *pNumFrames
1062
1063
1064     //Calculate a framewise short (+/-SearchRange) CCF between the input waveforms (left channels
     only),
1065     //and search the maximum. The position of the maximum is the required lag around
1066     OTA_FLOAT* pRefSigRaw = ((CAudioSignal*)pSignals[0])->mpData[0]
1067     OTA_FLOAT* pDegSigRaw = ((CAudioSignal*)pSignals[1])->mpData[0]
1068     OTA_FLOAT* pRefSig = (OTA_FLOAT*)matMalloc(((CAudioSignal*)pSignals[0])->mSignalLength *
     sizeof(OTA_FLOAT))
1069     OTA_FLOAT* pDegSig = (OTA_FLOAT*)matMalloc(((CAudioSignal*)pSignals[1])->mSignalLength *
     sizeof(OTA_FLOAT))
1070     matbCopy(pRefSigRaw, pRefSig, ((CAudioSignal*)pSignals[0])->mSignalLength)
1071     matbCopy(pDegSigRaw, pDegSig, ((CAudioSignal*)pSignals[1])->mSignalLength)
1072
1073
1074     matbSqr1(pRefSig, ((CAudioSignal*)pSignals[0])->mSignalLength)
1075     matbSqr1(pDegSig, ((CAudioSignal*)pSignals[1])->mSignalLength)
1076
1077     int Next=0
1078     int Step = 2*SearchRange+1
1079
1080     OTA_FLOAT *tempBuffer1 = (OTA_FLOAT*)matMalloc(CorrLen * sizeof(OTA_FLOAT))
1081     OTA_FLOAT *tempBuffer2 = (OTA_FLOAT*)matMalloc(CorrLen * sizeof(OTA_FLOAT))
1082
1083
1084
1085     OTA_FLOAT** pCorrmatrix = (OTA_FLOAT**)matMalloc2D(NumFrames, Step * sizeof(OTA_FLOAT))
1086
1087     OTA_FLOAT* pCenterEnergy = (OTA_FLOAT*)matMalloc(NumFrames * sizeof(OTA_FLOAT))
1088     int* pOptOffset = (int*)matMalloc(NumFrames * sizeof(int))
1089
1090     long LastStartRef = ((CAudioSignal*)pSignals[0])->mSignalLength-CorrLen
1091     int NumDegFrames = (((NumFrames) < ((((CAudioSignal*)pSignals[1])->mSignalLength-CorrLen) /
     Stepsize)) ? (NumFrames) : ((((CAudioSignal*)pSignals[1])->mSignalLength-CorrLen) / Stepsize))
1092
1093
1094
1095     int LastValidFrame = 0
1096     int LastValidSectionStart = 0
1097     bool IsValidSection = false
1098     int* pActiveFrameFlags = new int[NumFrames]
1099     pActiveFrameDetection->GetActiveFrameFlags(1, 0, Stepsize, pActiveFrameFlags, NumFrames)
1100
1101     for (f=0  f<NumFrames  f++)
1102     {
1103         if (f < NumDegFrames)
1104         {
1105             int NextRefStart = f*Stepsize+pDelayInSamplesPerFrame[f]-SearchRange
1106             int d = 0
1107             if (NextRefStart < 0)
1108             {
1109                 d = (((-NextRefStart) < (2*SearchRange+1)) ? (-NextRefStart) : (2*SearchRange+1))
1110                 matbZero(pCorrmatrix[f], d)
1111                 NextRefStart = 0
1112             }
1113
1114             //Skip inactive frames
1115
1116             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]) || (f == 1 && pActiveFrameFlags[f]))

        {
            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
```

```
1247
1248                }
1249            }
1250
1251            if (1)
1252            {
1253
1254                if (pActiveFrameFlags[Deg] && CurrentBestR > 0.95)
1255
1256                {
1257                    PenaltyWeightFactor[Deg]=0.25
1258
1259                }
1260            }
1261
1262            SecondLastBestR = LastBestR
1263            LastBestR = BestR
1264        }
1265    }
1266
1267    //Get a vector with the relative delay of each frame to the previous one based on the
1268    //results from the last delay calculation.
1269    int* pRelativeDelayPerFrame = (int*)matMalloc(NumFrames * sizeof(int))
1270    pRelativeDelayPerFrame[0] = 0
1271    for (f=1  f<NumFrames  f++)
1272        pRelativeDelayPerFrame[f] = pDelayInSamplesPerFrame[f] - pDelayInSamplesPerFrame[f-1]
1273
1274    VITERBI_PARA VP
1275    VP.ViterbiDistanceWeightFactor = mProcessData.mP.mSpeechTAPara.ViterbiDistanceWeightFactor
1276    VP.UseRelDistance = true
1277
1278    Viterbi(pCorrmatrix, pRelativeDelayPerFrame, PenaltyWeightFactor, pOptOffset,
    pReliabilityPerFrame, NumFrames, 2*SearchRange+1, &VP)
1279
1280    matFree(pRelativeDelayPerFrame)
1281
1282    for (i=0  i<3  i++)
1283    {
1284        for (f=1  f<NumFrames-1  f++)
1285        {
1286            //Check if the previous frame had a better correlation than this frame
1287            if (pCorrmatrix[f][pOptOffset[f]]<pCorrmatrix[f-1][pOptOffset[f-1]]-0.2)
1288            {
1289                if (pCorrmatrix[f][pOptOffset[f-1]] > pCorrmatrix[f][pOptOffset[f]])
1290                {
1291                    pOptOffset          [f] = pOptOffset[f-1]
1292                    pReliabilityPerFrame[f] = pCorrmatrix[f][pOptOffset[f-1]]
1293                }
1294            }
1295            //Check if the next frame has a better correlation than this frame
1296            else if (pCorrmatrix[f][pOptOffset[f]]<pCorrmatrix[f+1][pOptOffset[f+1]]-0.2)
1297            {
1298                if (pCorrmatrix[f][pOptOffset[f+1]] > pCorrmatrix[f][pOptOffset[f]])
1299                {
1300                    pOptOffset          [f] = pOptOffset[f+1]
1301                    pReliabilityPerFrame[f] = pCorrmatrix[f][pOptOffset[f+1]]
1302                }
1303            }
1304        }
1305    }
1306
1307    for (f=0  f<NumFrames  f++)
1308        pDelayInSamplesPerFrame[f] += pOptOffset[f] - SearchRange
1309
1310        if (mProcessData.mpLogFile)
1311        {
1312            double Avg=0
1313            double AvgCnt=0
```

```
1314
1315
1316            for (f=0  f<NumFrames  f++)
1317
1318
1319        }
1320
1321    matFree2D((void**)pCorrmatrix)
1322    pCorrmatrix = 0
1323
1324    if (pCenterEnergy)
1325        matFree(pCenterEnergy)
1326    if (pRefSig)
1327        matFree(pRefSig)
1328    if (pDegSig)
1329        matFree(pDegSig)
1330    if (pOptOffset)
1331        matFree(pOptOffset)
1332    if(PenaltyWeightFactor)
1333        matFree(PenaltyWeightFactor)
1334
1335    delete[] pActiveFrameFlags
1336
1337    return rc
1338 }
1339
1340 bool CSpeechTempAlignment::Init(CProcessData* pProcessData)
1341 {
1342    bool rc = true
1343    int i=0
1344
1345    SPEECH_WINDOW_PARA* pPara=pProcessData->mP.mSpeechTAPara.Win
1346    while(pPara->Samplerate<pProcessData->mSamplerate)
1347        pPara++
1348
1349    pProcessData->mDelayFineAlignCorrlen = pPara->mDelayFineAlignCorrlen
1350    pProcessData->mSRDetectFineAlignCorrlen = pPara->mSRDetectFineAlignCorrlen
1351    for (i=0  i<8  i++)
1352    {
1353        pProcessData->mpCoarseAlignCorrlen[i] = pPara->CoarseAlignCorrlen[i]
1354        pProcessData->mpWindowSize[i]      = pPara->WindowSize[i]
1355        pProcessData->mpOverlap[i]      = pPara->WindowSize[i] / 2
1356        pProcessData->mpViterbiDistanceWeightFactor[i] = pPara->pViterbiDistanceWeightFactor[i]
1357    }
1358
1359    pProcessData->mpOverlap[0]      = 0
1360    pProcessData->mpOverlap[1]      = 0
1361    pProcessData->mpOverlap[2]      = 0
1362    pProcessData->Init(0, 1.0)
1363
1364    rc = CTempAlignment::Init(pProcessData, new CSpeechDelaySearch, new CSpeechActiveFrameDetection)
1365
1366    if (rc)
1367        for (i=0  i<2  i++)
1368            mppSignals[i] = new CAudioSignal
1369
1370    mpFeatureList  = new CSpeechFeatureList
1371    mpFeatureList2 = new CSpeechFeatureList
1372
1373    return rc
1374 }
1375
1376 bool CSpeechTempAlignment::Run(unsigned long Control, OTA_RESULT* pResult, int TArunIndex)
1377 {
1378    bool rc=true
1379
1380    if (rc) rc = CTempAlignment::Run(Control, pResult, TArunIndex)
1381
```

```
1382      return rc
1383  }
1384
1385  //Determine some reasonable limits for the delay searches.
1386  //All results are measured in ms and describe the delay of the ref signal.
1387  //- The ref file has at least 40% activity and consists of two sentences.
1388  //- The total amount of silence is split into at least two sections (typically three)
1389  //- Assume that no more than 50% of the silence fall before the start or after the end of the file
1390  //- Assume that the speech part is not cut off at either end due to the delay
1391  //- The ref file may by z samples longer than the ref file. These may all be  silence
1392  //Flen,r = length of the ref file
1393  //This results in a max delay of the ref signal of:
1394  //  D1 = (Flen,ref*0.4*0.5)  or, more exact for the first utterance: D1 = RefStart
1395  //and:
1396  //  D2 = -(Flen,ref*0.4 + z) * 0.5
1397  void CSpeechTempAlignment::GetDelayLimits(int RefStartSample, int* pMaxDelayPos, int* MaxDelayNeg)
1398  {
1399      *pMaxDelayPos = (((SamplesToMSeconds(RefStartSample)) < (mProcessData.mMaxStaticDelayInMs)) ?
      (SamplesToMSeconds(RefStartSample)) : (mProcessData.mMaxStaticDelayInMs))
1400      int z = mppSignals[1]->mSignalLength-mppSignals[0]->mSignalLength
1401      *MaxDelayNeg = (((SamplesToMSeconds(-(mppSignals[0]->mSignalLength*0.2 + z) * 0.8)) >
      (mProcessData.mMinStaticDelayInMs)) ? (SamplesToMSeconds(-(mppSignals[0]->mSignalLength*0.2 + z)
      * 0.8)) : (mProcessData.mMinStaticDelayInMs))
1402
1403
1404  }
1405
1406  //Calculate a noise switching indicator
1407
1408  OTA_FLOAT CSpeechTempAlignment::EvaluateNoiseOfOneSection(OTA_FLOAT* NoiseLevelSpeech, OTA_FLOAT*
      NoiseLevelSilence, OTA_FLOAT* NoiseLevelAfter, int NumChecks, SECTION* Sec, SECTION* SecNext, int
      Signal)
1409  {
1410      OTA_FLOAT Switching = 0.0
1411      int i
1412      mProcessData.Init(1, 1)
1413      mpActiveFrameDetection->Init(&mProcessData)
1414      mpActiveFrameDetection->Start(mppSignals)
1415
1416      SEGMENT Segment
1417
1418      Segment.Start = Sec->Start
1419      Segment.End = Sec->End
1420      *NoiseLevelSpeech = mpActiveFrameDetection->GetLevelBelowThreshold(&Segment, Signal, 0)
1421
1422      Segment.Start = Sec->End
1423      Segment.End = SecNext->Start
1424      *NoiseLevelSilence = mpActiveFrameDetection->GetLevelBelowThreshold(&Segment, Signal, 0)
1425
1426      int Offset=-50
1427      int StepInMs = 10
1428      for (i=0  i<NumChecks  i++)
1429      {
1430          Segment.Start = Sec->End + MSecondsToSamples(Offset+i*StepInMs)
1431
1432          Segment.End = Segment.Start + mProcessData.mStepSize
1433          if (Segment.End<SecNext->Start)
1434              NoiseLevelAfter[i] = mpActiveFrameDetection->GetLevelBelowThreshold(&Segment, Signal, 0)
1435          else
1436              NoiseLevelAfter[i] = NoiseLevelAfter[(((i-1) > (0)) ? (i-1) : (0))]
1437
1438      }
1439
1440      if (*NoiseLevelSilence<0.125)
1441          *NoiseLevelSilence = 0.125
1442
1443      *NoiseLevelSpeech = 10*log10(*NoiseLevelSpeech+0.0000001)
1444      *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]
```

```
1507      OTA_FLOAT NoiseLevelAfterDeg[30]
1508
1509      for (int i = 0  i<30  i++)
1510      {
1511          NoiseLevelAfterRef[i] = 0.0
1512          NoiseLevelAfterDeg[i] = 0.0
1513      }
1514
1515      if (mNumReparsePoints>1)
1516      {
1517          int RPToBeUsed=0
1518
1519          while (RPToBeUsed<mNumReparsePoints &&
    mpReparsePoints[RPToBeUsed+1].Ref.Start-mpReparsePoints[RPToBeUsed].Ref.End <
    MSecondsToSamples(200))
1520              RPToBeUsed++
1521
1522          IndicatorRef = EvaluateNoiseOfOneSection(&NoiseLevelSpeechRef, &NoiseLevelSilenceRef,
    NoiseLevelAfterRef, 30, &mpReparsePoints[RPToBeUsed].Ref,
    &mpReparsePoints[RPToBeUsed+1].Ref, 0)
1523          IndicatorDeg = EvaluateNoiseOfOneSection(&NoiseLevelSpeechDeg, &NoiseLevelSilenceDeg,
    NoiseLevelAfterDeg, 30, &mpReparsePoints[RPToBeUsed].Deg,
    &mpReparsePoints[RPToBeUsed+1].Deg, 1)
1524
1525      }
1526
1527      pBGNSwitchingLevel[0] = IndicatorRef
1528      pBGNSwitchingLevel[1] = IndicatorDeg
1529      pNoiseLevelSpeech[0] = NoiseLevelSpeechRef
1530      pNoiseLevelSpeech[1] = NoiseLevelSpeechDeg
1531      pNoiseLevelSilence[0] = NoiseLevelSilenceRef
1532      pNoiseLevelSilence[1] = NoiseLevelSilenceDeg
1533  }
1534
1535  //Calculate the average pitch frequency of of one channel.
1536  OTA_FLOAT CSpeechTempAlignment::GetPitchFreq(CTASignal **pSignals, int Signal, int Channel)
1537  {
1538      OTA_FLOAT AvgPitch=0
1539      int PitchStart=0
1540      CPitchBase Pitch(mpSmartBufferPool)
1541      Pitch.GetPitchVector(&mProcessData, ((CAudioSignal*)pSignals[Signal])->mpData[Channel], 0,
    ((CAudioSignal*)pSignals[Signal])->mSignalLength, 0, 0, 0, &AvgPitch, &PitchStart)
1542      return (float)AvgPitch
1543  }
1544
1545  //Get the individual pitch values for all frames on a given frame scale. Values <=0 mark unvoiced
    frames.
1546  OTA_FLOAT CSpeechTempAlignment::GetPitchVector(CTASignal **pSignals, int Signal, int Channel,
    OTA_FLOAT* pVector, int NumFrames, int SamplesPerFrame)
1547  {
1548      OTA_FLOAT AvgPitch=0
1549      CPitchBase Pitch(mpSmartBufferPool)
1550
1551      OTA_FLOAT* pPitchVec
1552      int NumPitchFrames
1553      int PitchStartOffset=0
1554      Pitch.GetPitchVector(&mProcessData, ((CAudioSignal*)pSignals[Signal])->mpData[Channel], 0,
    ((CAudioSignal*)pSignals[Signal])->mSignalLength, &pPitchVec, &NumPitchFrames,
    &mpResults->mPitchFrameSize, &AvgPitch, &PitchStartOffset)
1555
1556      int StartOffsetExternal = 0
1557      int StartFrameTA = 0
1558
1559      if (Signal==1 && mStartOffset<0)
1560      {
1561          StartOffsetExternal = (-mStartOffset+SamplesPerFrame/2) / SamplesPerFrame
1562          StartOffsetExternal = (((0) > (StartOffsetExternal)) ? (0) : (StartOffsetExternal))
1563          StartFrameTA = (((0) >
```

```
    ((-mStartOffset+mpResults->mPitchFrameSize/2)/mpResults->mPitchFrameSize)) ? (0) :
    ((-mStartOffset+mpResults->mPitchFrameSize/2)/mpResults->mPitchFrameSize))
1564        }
1565        else if (Signal==0 && mStartOffset>0)
1566        {
1567            StartOffsetExternal = (mStartOffset+SamplesPerFrame/2) / SamplesPerFrame
1568            StartFrameTA = (mStartOffset+mpResults->mPitchFrameSize,
    mpResults->mPitchFrameSize/2)/mpResults->mPitchFrameSize
1569        }
1570
1571        int FrameCount = 0
1572        int StartSample = SamplesPerFrame/2
1573        int LastStart = (NumFrames-2-StartOffsetExternal)*SamplesPerFrame
1574        while (StartSample<LastStart)
1575        {
1576            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))))]
1577            StartSample += SamplesPerFrame
1578            FrameCount++
1579        }
1580        pVector[NumFrames-2]=0
1581        pVector[NumFrames-1]=0
1582
1583        OTA_FLOAT StartPitch = 0
1584        for (int i=0  i<StartOffsetExternal && i<NumFrames  i++)
1585            pVector[i] = StartPitch
1586
1587        matFree(pPitchVec)
1588
1589        return (float)AvgPitch
1590    }
1591
1592    void CSpeechTempAlignment::GetFilterCharacteristics(FilteringParameters *FilterParams)
1593    {
1594
1595        FilterParams->disturbedEnergyQuotient = 1.0
1596
1597    }
1598
1599    }
1600
1601
```