```
    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
{

CSpeechActiveFrameDetection::CSpeechActiveFrameDetection()
{
    for (int s=0; s<2; s++)
    {
        for (int c=0; c<2; c++)
        {
            mppActiveFrameFlags[s][c]=0;
            mDataValidFlags[s][c]=0;
            mSignalLevels[s][c]=0;
            mNoiseLevels[s][c]=0;
            mNoiseThresholds[s][c]=0;
        }
    }
}

CSpeechActiveFrameDetection::~CSpeechActiveFrameDetection()
{
    Free();
}
```

```
void CSpeechActiveFrameDetection::Free()
{
    for (int s=0; s<2; s++)
    {
        for (int c=0; c<2; c++)
        {
            if(mppActiveFrameFlags[s][c]) delete[] mppActiveFrameFlags[s][c];
            mppActiveFrameFlags[s][c]=0;
            mDataValidFlags[s][c]=0;
            mSignalLevels[s][c]=0;
            mNoiseLevels[s][c]=0;
            mNoiseThresholds[s][c]=0;
        }
    }
}

bool CSpeechActiveFrameDetection::Init(CProcessData* pProcessData)
{
    bool rc=true;

    mProcessData = *pProcessData;
    return rc;
}

bool CSpeechActiveFrameDetection::Start(CTASignal** pSignals)
{
    bool rc=true;

    mProcessData.Init(0, 1);

    rc = mFeatureList.Create(pSignals, &mProcessData, OTA_FLTYPE_VAD);

    for (int i=0; rc && i<2; i++)
        mNumFeatureFrames[i] = mFeatureList.GetFVector(0, i, 0)->mSize;

    return rc;
}

void CSpeechActiveFrameDetection::GetNoiseThreshold(OTA_FLOAT* Vec, int VecLen,
OTA_FLOAT* pNoise, OTA_FLOAT* pSignal, OTA_FLOAT* pNoiseThreshold)
{
    OTA_FLOAT NoiseThreshold;
    OTA_FLOAT NoiseLevel;
    OTA_FLOAT StdDevOfNoisyPart;
    OTA_FLOAT SignalLevel;
    OTA_FLOAT MinLevel;

    NoiseThreshold = matMean(Vec, VecLen);
    MinLevel = matMax(Vec, VecLen);
    MinLevel = MinLevel >0 ? MinLevel * 1.0e-5 : 0.5;
    matbThresh1(Vec, VecLen, MinLevel, MAT_GT);

    for( int i = 0; i < 12; i++ )
    {
        //ToDo: Compute mean and StdDev of the noise power (StdDevOfNoisyPart)*/

        //... Code missing in public C code

        NoiseThreshold = NoiseLevel + 2.005 * StdDevOfNoisyPart;
        NoiseThreshold *= 1.001;
    }

    //* Compute the signal and noise levels */
    //* ToDo:
    //- Set NoiseLevel  to the mean of all samples <= NoiseThreshold
    //- Set SignalLevel to the mean of all samples > NoiseThreshold
    //- Limit the thresholds to 1e-7
    //

}

OTA_FLOAT CSpeechActiveFrameDetection::ModifyThreshold(OTA_FLOAT NoiseLevel, OTA_FLOAT
SignalLevel, OTA_FLOAT NoiseThreshold, bool IsRefSignal, int* MinSpeechLength)
{
    const OTA_FLOAT LowSNRdB=3;
```

```
    OTA_FLOAT SNRdB = 10*log10(SignalLevel/NoiseLevel);
    OTA_FLOAT NoisedB = 10*log10(NoiseLevel);

    if (SNRdB<12.0)
    {
        if (SNRdB<LowSNRdB)
        {
            *MinSpeechLength =4;
            NoiseThreshold = NoiseLevel + 0.03*(SignalLevel-NoiseLevel);
        }
    }

    else if (SNRdB>16.0 && SNRdB<35)
    {
        NoiseThreshold *= 5;
    }
    else if (SNRdB>=35)
    {
        NoiseThreshold *= 1.5;
    }

    return NoiseThreshold;
}

void CSpeechActiveFrameDetection::IdentifyActivity(OTA_FLOAT* Vec, OTA_FLOAT*
pPitchVec, int VecLen, OTA_FLOAT NoiseLevel, OTA_FLOAT SignalLevel, OTA_FLOAT
NoiseThreshold, bool IsRefSignal)
{
    const OTA_FLOAT LowSNRdB=3;
    OTA_FLOAT LevelMin=1;
    OTA_FLOAT g;
    int  count;
    int  iteration;
    int  start;
    int  finish;

    OTA_FLOAT SNRdB = 10*log10(SignalLevel/NoiseLevel);
    OTA_FLOAT NoisedB = 10*log10(NoiseLevel);

    int MaxDropoutLength = MSecondsToFrames(10);
    int MinSpeechLength = MSecondsToFrames(50);
    if (IsRefSignal) MinSpeechLength = MSecondsToFrames(35);
    int MinPauseLength  = MSecondsToFrames(200);
    int MinPauseLength2 = MSecondsToFrames(300);
    int MinPauseLength3 = MSecondsToFrames(500);
    int IsIsolatedDistance = MSecondsToFrames(20);

    NoiseThreshold = ModifyThreshold(NoiseLevel, SignalLevel, NoiseThreshold,
IsRefSignal, &MinSpeechLength);

    for( count = 0L; count < VecLen; count++ )
        if( Vec[count] <= NoiseThreshold )
            Vec[count] = -Vec[count];

    Vec[0] = -LevelMin;
    Vec[VecLen-1] = -LevelMin;

    int InactivityStart = 0;
    int InactivityFinish = 0;
    int ActivityStart = 0;
    int NextActivityStart = 0;
    int ActivityFinish = 0;
    for( count = 1; count < VecLen; count++ )
    {
        if( (Vec[count-1] > 0.0f) && (Vec[count] <= 0.0f) )
            ActivityFinish = InactivityStart = count;
        if( (Vec[count-1] <= 0.0f) && (Vec[count] > 0.0f) )
        {
            ActivityStart = NextActivityStart;
            NextActivityStart = InactivityFinish = count;
            if( (InactivityFinish - InactivityStart) <= MaxDropoutLength )
            {
                bool DoIt=false;
                if (ActivityFinish-ActivityStart<MinSpeechLength)
                {
```

```
                    for (iteration=InactivityFinish; iteration<VecLen &&
iteration<InactivityFinish+MinSpeechLength; iteration++)
                        if (!Vec[iteration])
                            break;
                    if (iteration>=InactivityFinish+3*MinSpeechLength)
                        DoIt=true;
                }
                else DoIt=true;

                if (DoIt)
                    for( iteration = InactivityStart; iteration < InactivityFinish;
iteration++ )
                        Vec[iteration] = LevelMin;
            }
        }
    }

    if (SNRdB<LowSNRdB)
        //Code missing: extend all active segmants by one frame at either end

    //Missing: Eliminate pauses shorter than MinPauseLength

    if( SignalLevel >= (NoiseLevel * 1000.0f) )
    {
        for( count = 1; count < VecLen; count++ )
        {
            if( (Vec[count] > 0.0f) && (Vec[count-1] <= 0.0f) )
                start = count;
            if( (Vec[count] <= 0.0f) && (Vec[count-1] > 0.0f) )
            {
                finish = count;
                g = 0.0f;
                for( iteration = start; iteration < finish; iteration++ )
                    g += Vec[iteration];
                if( g < 3.0f * NoiseThreshold * (finish - start) )
                    for( iteration = start; iteration < finish; iteration++ )
                        Vec[iteration] = -Vec[iteration];
            }
        }
    }

    //not available: - Join sections of speech that are separated by less than
MinPauseLength */
    //- Join sections of speech that are separated by less than MinPauseLength2 and
which exceed the threshold at least once */

    //Todo: Make sure that there is at least one active section

    for( count = 0L; count < VecLen; count++ )
        if( Vec[count] <= 0.0f ) Vec[count] = 0.0f;
        else                     Vec[count] = LevelMin;

    int PauseStart=Vec[0]>0 ? -1:0;
    int SpeechStart= Vec[0]>0 ? 0:-1;
    int PreviousPauseStart= -1;
    int PreviousSpeechStart= -1;
    for( count = 1L; count < VecLen; count++ )
    {
        if (Vec[count-1]>0 && Vec[count]<=0)
        {   PreviousPauseStart = PauseStart; PauseStart = count; }

        if (Vec[count-1]<=0 && Vec[count]>0 )
        {
            PreviousSpeechStart = SpeechStart;
            SpeechStart = count;

            if (   PauseStart-PreviousSpeechStart < MinSpeechLength*2
                && PreviousSpeechStart - PreviousPauseStart > MSecondsToFrames(400)
                && SpeechStart - PauseStart > MSecondsToFrames(400))
            {
                if (PreviousPauseStart>=0 && PreviousSpeechStart>=0 &&
PreviousPauseStart>=0 && PauseStart>=0)
                    for (int i=PreviousSpeechStart; i<PauseStart; i++)
                        Vec[i] = 0.0;
            }
            else if (   PauseStart-PreviousSpeechStart < MinSpeechLength*2
```

```
                && PreviousPauseStart == 0
                && SpeechStart - PauseStart > MSecondsToFrames(400))
            {
                if (PreviousPauseStart>=0 && PreviousSpeechStart>=0 &&
PreviousPauseStart>=0 && PauseStart>=0)
                    for (int i=PreviousSpeechStart; i<PauseStart; i++)
                        Vec[i] = 0.0;
            }

        }
    }

    start = 0L;
    finish = 0L;
    for( count = 1; count < VecLen; count++ )
    {
        if( (Vec[count] > 0.0f) && (Vec[count-1] <= 0.0f) )
        {
            start = count;
            if( (finish > 0L) && ((start - finish) <= MinPauseLength3) )
                for( iteration = finish; iteration < start; iteration++ )
                    Vec[iteration] = LevelMin;
        }
        if( (Vec[count] <= 0.0f) && (Vec[count-1] > 0.0f) )
            finish = count;
    }

}

int CSpeechActiveFrameDetection::SkipConstLevel(OTA_FLOAT* Vec, int VecLen)
{
    int StartConstSection=0;
    int EndConstSection;
    {
        while(StartConstSection<VecLen && Vec[StartConstSection]<0)
StartConstSection++;

        int i;
        OTA_FLOAT AvgE=0;
        int NumFramesInWin = MSecondsToFrames(50);
        for (i=StartConstSection; i<VecLen && i<StartConstSection+NumFramesInWin; i++)
            AvgE += Vec[i];
        AvgE /= NumFramesInWin;

        EndConstSection = StartConstSection+1;
        while (EndConstSection<VecLen && fabs(Vec[EndConstSection])<20*AvgE &&
fabs(Vec[EndConstSection])>0.05*AvgE)
            EndConstSection++;

    }

    if (EndConstSection>VecLen-3)
        EndConstSection = 0;

    if (EndConstSection-StartConstSection>MSecondsToFrames(50))
        return EndConstSection;
    else return 0;
}

int CSpeechActiveFrameDetection::SearchStartFrame(int Signal, int Channel, int
EarliestStartFrame)
{
    int i, j;
    int* pVec = mppActiveFrameFlags[Signal][Channel];
    OTA_FLOAT* pEnergy = mFeatureList.GetFVector(0, Signal, Channel)->mpVector;
    int VecLen = (int)mFeatureList.GetFVector(0, Signal, Channel)->mSize;

    OTA_FLOAT SigLevel, NoiseLevel, Threshold;
    i=EarliestStartFrame; while (i>0 && pVec[i--]); while (i>0 && !pVec[i--]);
    j=EarliestStartFrame; while (j<VecLen && !pVec[j++]); while (j<VecLen &&
pVec[j++]);
    GetNoiseThreshold(pEnergy+i, j-i, &NoiseLevel, &SigLevel, &Threshold);
    OTA_FLOAT LocalSNRdB = 10 * log10(SigLevel/NoiseLevel);

    if (0 && LocalSNRdB>7)
```

```
    {
        int FramesToSkip = SkipConstLevel(pEnergy+EarliestStartFrame,
VecLen-EarliestStartFrame);
        EarliestStartFrame += FramesToSkip;
    }

    int NumFramesRequired = (int)(0.050F * (float)mProcessData.mSamplerate /
(float)mProcessData.mStepSize);
    int PotentialStartFrame = EarliestStartFrame;
    OTA_FLOAT AvgE = SigLevel;

    const OTA_FLOAT MaxSNRdB = 25;
    if (LocalSNRdB>MaxSNRdB)
    {
        AvgE = Threshold;
    }
    else
    {
        LocalSNRdB = (((MaxSNRdB) < (LocalSNRdB)) ? (MaxSNRdB) : (LocalSNRdB));
        AvgE = SigLevel-LocalSNRdB*(SigLevel-Threshold)/MaxSNRdB;
    }

    for (i=EarliestStartFrame; i<mNumFeatureFrames[Signal]-NumFramesRequired; i++)
    {
        if (pVec[i])
        {
            bool AvgEExceeded=false;
            for (j=1; j<NumFramesRequired; j++)
            {
                if (pEnergy[i+j]>AvgE)
                    AvgEExceeded = true;
                if (!pVec[i+j])
                    break;
            }
            if (j==NumFramesRequired && AvgEExceeded)
            {
                PotentialStartFrame=i;
                break;
            }
            else
            {
                i+=j;
            }
        }
    }

    if (PotentialStartFrame>EarliestStartFrame+MSecondsToFrames(64))
        PotentialStartFrame-=MSecondsToFrames(64);

    if (PotentialStartFrame>0.98*mNumFeatureFrames[Signal])
        PotentialStartFrame = EarliestStartFrame;

    return PotentialStartFrame;
}

OTA_FLOAT CSpeechActiveFrameDetection::GetSectionSnrIndB(int Signal, int Channel, int
EarliestStartFrame, OTA_FLOAT* pNoiseLevel, OTA_FLOAT* pSignalLevel, OTA_FLOAT*
pThreshold)
{
    int i, j;
    int* pVec = mppActiveFrameFlags[Signal][Channel];
    OTA_FLOAT* pEnergy = mFeatureList.GetFVector(0, Signal, Channel)->mpVector;
    int VecLen = (int)mFeatureList.GetFVector(0, Signal, Channel)->mSize;

    OTA_FLOAT SigLevel, NoiseLevel, Threshold;
    i=EarliestStartFrame; while (i>0 && pVec[i--]); while (i>0 && !pVec[i--]); i+=2;
    j=EarliestStartFrame; while (j<VecLen && !pVec[j++]); while (j<VecLen &&
pVec[j++]); j-=2;
    GetNoiseThreshold(pEnergy+i, j-i, &NoiseLevel, &SigLevel, &Threshold);
    *pNoiseLevel = 10 * log10(NoiseLevel);
    *pSignalLevel = 10 * log10(SigLevel);
    *pThreshold = 10 * log10(Threshold);
    OTA_FLOAT LocalSNRdB = 10 * log10(SigLevel/NoiseLevel);

    return LocalSNRdB;
}
```

```
void CSpeechActiveFrameDetection::GetClassificationMeasure(OTA_FLOAT* Vec, int VecLen,
int* NumActiveSections, int* AvgActiveSectionLen, int* TotalActiveSectionLen,
OTA_FLOAT* ActiveInactiveRatio)
{
    int i;
    *NumActiveSections=0;
    *AvgActiveSectionLen=0;
    *TotalActiveSectionLen=0;
    int AStart=0;
    int AEnd=0;
    for( i = 1L; i < VecLen; i++ )
    {
        if (Vec[i]>0 && Vec[i-1]<=0)
            AStart = i;

        if (Vec[i]<=0 && Vec[i-1]>0 || (i==VecLen-1 && Vec[i]>0))
        {
            AEnd = i;
            *TotalActiveSectionLen += AEnd-AStart;
            (*NumActiveSections)++;
        }
    }
    if (*NumActiveSections)
    {
        *AvgActiveSectionLen = *TotalActiveSectionLen / *NumActiveSections;
        *ActiveInactiveRatio = (OTA_FLOAT)(*TotalActiveSectionLen) /
(OTA_FLOAT)(VecLen-*TotalActiveSectionLen);
    }
    else
    {
        *AvgActiveSectionLen = 0;
        *NumActiveSections = 0,
        *ActiveInactiveRatio = 0.0;
        *TotalActiveSectionLen = 1.0E-15;
    }
}

void CSpeechActiveFrameDetection::ClassifyDistortion(int Channel, OTA_FLOAT*
ClickLevelOfDeg)
{
    int i, VecLen;
    int Dummy;
    OTA_FLOAT pNoiseLevel[2];
    OTA_FLOAT pSignalLevel[2];
    OTA_FLOAT pNoiseThreshold[2];
    int NumActiveSections[2];
    int AvgActiveSectionLen[2];
    int TotalActiveSectionLen[2];
    OTA_FLOAT ActiveInactiveRatio[2];

    VecLen = ((((int)mFeatureList.GetFVector(0, 0, Channel)->mSize) <
((int)mFeatureList.GetFVector(0, 1, Channel)->mSize)) ?
((int)mFeatureList.GetFVector(0, 0, Channel)->mSize) :
((int)mFeatureList.GetFVector(0, 1, Channel)->mSize));

    if (1 || VecLen)
    {
        OTA_FLOAT* Vec = (OTA_FLOAT*)matMalloc(VecLen * sizeof(OTA_FLOAT));
        matbCopy(mFeatureList.GetFVector(0, 0, Channel)->mpVector, Vec, VecLen);
        GetNoiseThreshold(Vec, VecLen,  pNoiseLevel+0,  pSignalLevel+0,
pNoiseThreshold+0);
        pNoiseThreshold[0] = ModifyThreshold(pNoiseLevel[0], pSignalLevel[0],
pNoiseThreshold[0], true, &Dummy);

        for( i = 0L; i < VecLen; i++ )
            if( Vec[i] <= pNoiseThreshold[0] )
                Vec[i] = -Vec[i];
        Vec[0] = Vec[VecLen-1] = -1.0;
        GetClassificationMeasure(Vec, VecLen, NumActiveSections+0,
AvgActiveSectionLen+0, TotalActiveSectionLen+0, ActiveInactiveRatio+0);

        matbCopy(mFeatureList.GetFVector(0, 1, Channel)->mpVector, Vec, VecLen);
        GetNoiseThreshold(Vec, VecLen,  pNoiseLevel+1,  pSignalLevel+1,
pNoiseThreshold+1);
        pNoiseThreshold[1] = ModifyThreshold(pNoiseLevel[1], pSignalLevel[1],
```

```
pNoiseThreshold[1], false, &Dummy);

        for( i = 0L; i < VecLen; i++ )
            if( Vec[i] <= pNoiseThreshold[1] )
                Vec[i] = -Vec[i];
        Vec[0] = Vec[VecLen-1] = -1.0;
        GetClassificationMeasure(Vec, VecLen, NumActiveSections+1,
AvgActiveSectionLen+1, TotalActiveSectionLen+1, ActiveInactiveRatio+1);

        if (mProcessData.mpLogFile)
        {
            fprintf(mProcessData.mpLogFile, "\tClassification:\n");
            fprintf(mProcessData.mpLogFile,
"\tSignal\tNumActiveSections\tAvgActiveSectionLen\tTotalActiveSectionLen\tA
ctiveInactiveRatio\n");
            for (i=0; i<2; i++)
                fprintf(mProcessData.mpLogFile, "\t%d\t%d\t%d\t%d\t%.2f\n", i,
NumActiveSections[i], AvgActiveSectionLen[i], TotalActiveSectionLen[i],
(float)ActiveInactiveRatio[i]);
            fprintf(mProcessData.mpLogFile, "\tRatio Deg / Ratio Ref:\t%.3f\n",
(float)(ActiveInactiveRatio[1] / ActiveInactiveRatio[0]));
        }

        if (pNoiseLevel[1]<1000.0 && ActiveInactiveRatio[1] / ActiveInactiveRatio[0] >
1.7)
            *ClickLevelOfDeg = 1.0;
        else
            *ClickLevelOfDeg = 0.0;

        matFree(Vec);
    }
}

int CSpeechActiveFrameDetection::CalculateActivityFlags(int Signal, int Channel, int*
pFlagBuffer, int VecLen)
{
    int i;
    int PotentialStartFrame=0;

    OTA_FLOAT* pNoiseLevel     = &mNoiseLevels[Signal][Channel];
    OTA_FLOAT* pSignalLevel    = &mSignalLevels[Signal][Channel];
    OTA_FLOAT* pNoiseThreshold = &mNoiseThresholds[Signal][Channel];
    OTA_FLOAT ClicklevelOfDeg=0;

    if (Signal==1)
    {
        ClassifyDistortion(Channel, &ClicklevelOfDeg);
        if (mProcessData.mpLogFile)
            fprintf(mProcessData.mpLogFile, "\tDetected click level: %0.2f\n",
(float)ClicklevelOfDeg);
    }

    VecLen = (((VecLen) < ((int)mFeatureList.GetFVector(0, Signal, Channel)->mSize)) ?
(VecLen) : ((int)mFeatureList.GetFVector(0, Signal, Channel)->mSize));

    OTA_FLOAT* Vec = (OTA_FLOAT*)matMalloc(VecLen * sizeof(OTA_FLOAT));
    matbCopy(mFeatureList.GetFVector(0, Signal, Channel)->mpVector, Vec, VecLen);

    OTA_FLOAT* PitchVec=0;

    GetNoiseThreshold(Vec+PotentialStartFrame, VecLen-PotentialStartFrame,
pNoiseLevel,  pSignalLevel,  pNoiseThreshold);
    if (ClicklevelOfDeg) *pNoiseThreshold += (*pSignalLevel-*pNoiseThreshold) / 40.0;
    IdentifyActivity (Vec, PitchVec, VecLen, *pNoiseLevel, *pSignalLevel,
*pNoiseThreshold, Signal==0);

    for (i=0; i<VecLen; i++)
    {
        if (Vec[i]>0)
            pFlagBuffer[i] = 1;
        else
            pFlagBuffer[i] = 0;
    }

    if (mProcessData.mpLogFile)
        fprintf(mProcessData.mpLogFile, "\tInitialy measured
```

```
levels:\tSignal=%.1fdB,\tNoise=%.1fdB,\tThreshold=%.1fdB\n",
(float)(10.0*log10(*pSignalLevel)), (float)(10.0*log10(*pNoiseLevel)),
(float)(10.0*log10(*pNoiseThreshold)));

    i=VecLen-1;
    while ( i>0 && !pFlagBuffer[i]) i--;
    if (VecLen / (VecLen-i) <= 3)
    {
        matbCopy(mFeatureList.GetFVector(0, Signal, Channel)->mpVector, Vec,
VecLen-PotentialStartFrame);
        GetNoiseThreshold(Vec+PotentialStartFrame, i-PotentialStartFrame,  pNoiseLevel,
 pSignalLevel,  pNoiseThreshold);
        if (ClicklevelOfDeg) *pNoiseThreshold += (*pSignalLevel-*pNoiseThreshold) /
40.0;
        IdentifyActivity (Vec, PitchVec, VecLen, *pNoiseLevel, *pSignalLevel,
*pNoiseThreshold, Signal==0);

        for (i=0; i<VecLen; i++)
        {
            if (Vec[i]>0)
                pFlagBuffer[i] = 1;
            else
                pFlagBuffer[i] = 0;
        }
        if (mProcessData.mpLogFile)
            fprintf(mProcessData.mpLogFile, "\tAfter requalification:
\tSignal=%.1fdB,\tNoise=%.1fdB,\tThreshold=%.1fdB\n",
(float)(10.0*log10(*pSignalLevel)), (float)(10.0*log10(*pNoiseLevel)),
(float)(10.0*log10(*pNoiseThreshold)));
    }

    OTA_FLOAT SnrPerSection[100];
    OTA_FLOAT Noiselevel[100];
    OTA_FLOAT SignalLevel[100];
    OTA_FLOAT Threshold[100];
    int NumSections=0;
    for (i=1; i<VecLen; i++)
        if ( pFlagBuffer[i] && !pFlagBuffer[i-1] && NumSections<100)
            SnrPerSection[NumSections++] = GetSectionSnrIndB(Signal, Channel, i,
Noiselevel+NumSections, SignalLevel+NumSections, Threshold+NumSections);

    bool SNRVariationDetected = false;
    OTA_FLOAT AvgSnr=0;
    for (i=0; i<NumSections; i++)
        AvgSnr += SnrPerSection[i];
    AvgSnr = NumSections > 0 ? AvgSnr/NumSections : (OTA_FLOAT)0.0;
    for (i=1; i<NumSections; i++)
        if (fabs(SnrPerSection[i]-AvgSnr) > 7.0)
            SNRVariationDetected = true;

    OTA_FLOAT MaxThreshold = Threshold[0];
    OTA_FLOAT MinThreshold = Threshold[0];
    for (i=1; i<NumSections; i++)
    {
        MaxThreshold = (((MaxThreshold) > (Threshold[i])) ? (MaxThreshold) :
(Threshold[i]));
        MinThreshold = (((MinThreshold) < (Threshold[i])) ? (MinThreshold) :
(Threshold[i]));
    }
    if ((Signal!=0 || AvgSnr<25) && (MaxThreshold-MinThreshold>3 ||
SNRVariationDetected))
    {
        if (mProcessData.mpLogFile)
            fprintf(mProcessData.mpLogFile, "\tRequalifiyng thresholds and levels per
section\n");

        for (i=1; i<VecLen; i++)
        {
            if (pFlagBuffer[i] && !pFlagBuffer[i-1])
            {
                PotentialStartFrame = SearchStartFrame(Signal, Channel, i);
                if (PotentialStartFrame>i)
                {
                    for (i--; i<PotentialStartFrame; i++)
                        pFlagBuffer[i] = false;
                }
```

```
                else
                {
                    for (int j=PotentialStartFrame; j<i; j++)
                        pFlagBuffer[j] = true;
                }
            }
        }
    }

    PotentialStartFrame = SearchStartFrame(Signal, Channel, 0);

    matFree(Vec);

    return FramesToSamples(PotentialStartFrame);
}

int CSpeechActiveFrameDetection::GetMaxFrames(int Signal, int Channel)
{
    return mNumFeatureFrames[Signal];
}

void CSpeechActiveFrameDetection::GetLevels(int Signal, int Channel, int Downsampling,
OTA_FLOAT* pNoiseLevel, OTA_FLOAT* pSignalLevel, OTA_FLOAT* pNoiseThreshold, SEGMENT*
pSegment)
{

    mppActiveFrameFlags[Signal][Channel] = DEBUG_NEW int[mNumFeatureFrames[Signal]];
    mStartSamples[Signal][Channel] = CalculateActivityFlags(Signal, Channel,
mppActiveFrameFlags[Signal][Channel], mNumFeatureFrames[Signal]);
    mDataValidFlags[Signal][Channel] = true;

    if (!pSegment)
    {
        *pNoiseLevel     =
mNoiseLevels[Signal][Channel]*((OTA_FLOAT)Downsampling/(OTA_FLOAT)mProcessData.
mStepSize);
        *pSignalLevel    =
mSignalLevels[Signal][Channel]*((OTA_FLOAT)Downsampling/(OTA_FLOAT)mProcessData
.mStepSize);
        *pNoiseThreshold =
mNoiseThresholds[Signal][Channel]*((OTA_FLOAT)Downsampling/(OTA_FLOAT)mProcessD
ata.mStepSize);
    }
    else
    {
        int VecStart = pSegment->Start / mProcessData.mStepSize;
        int VecEnd = pSegment->End / mProcessData.mStepSize;
        int VecLen = mFeatureList.GetFVector(0, Signal, Channel)->mSize;
        if (VecEnd<=VecLen)
        {
            VecLen = VecEnd-VecStart;
            OTA_FLOAT* Vec = matxMalloc(VecLen);
            matbCopy(mFeatureList.GetFVector(0, Signal, Channel)->mpVector+VecStart,
Vec, VecLen);
            GetNoiseThreshold(Vec, VecLen, pNoiseLevel, pSignalLevel, pNoiseThreshold);
            matFree(Vec);
        }

    }
}

OTA_FLOAT CSpeechActiveFrameDetection::GetLevelBelowThreshold(SEGMENT* pSegment, int
Signal, int Channel)
{

    mppActiveFrameFlags[Signal][Channel] = DEBUG_NEW int[mNumFeatureFrames[Signal]];
    mStartSamples[Signal][Channel] = CalculateActivityFlags(Signal, Channel,
mppActiveFrameFlags[Signal][Channel], mNumFeatureFrames[Signal]);
    mDataValidFlags[Signal][Channel] = true;

    OTA_FLOAT NoiseThreshold = mNoiseThresholds[Signal][Channel];

    OTA_FLOAT  AverageEnergy=0;
    int        AverageCount=0;
    int VecStart = pSegment->Start / mProcessData.mStepSize;
    int VecEnd = pSegment->End / mProcessData.mStepSize;
```

```
    int VecLen = mFeatureList.GetFVector(0, Signal, Channel)->mSize;
    if (VecEnd<=VecLen)
    {
        VecLen = VecEnd-VecStart;
        OTA_FLOAT* Vec = mFeatureList.GetFVector(0, Signal, Channel)->mpVector;

        for (int i=VecStart; i<VecEnd; i++)
            if (Vec[i]<NoiseThreshold)
                {AverageEnergy+=Vec[i]; AverageCount++;}
        if (AverageCount)
            AverageEnergy /= AverageCount;
        else
            AverageEnergy = NoiseThreshold;
    }
    return AverageEnergy;
}

int CSpeechActiveFrameDetection::GetStartSample(int Signal, int Channel, int
EarliestSamples)
{

    mppActiveFrameFlags[Signal][Channel] = DEBUG_NEW int[mNumFeatureFrames[Signal]];
    mStartSamples[Signal][Channel] = CalculateActivityFlags(Signal, Channel,
mppActiveFrameFlags[Signal][Channel], mNumFeatureFrames[Signal]);
    mDataValidFlags[Signal][Channel] = true;

    return SearchStartFrame(Signal, Channel, EarliestSamples/mProcessData.mStepSize) *
mProcessData.mStepSize;
}

int CSpeechActiveFrameDetection::GetStartFrame(int Signal, int Channel, int
Downsampling, int EarliestSamples)
{

    mppActiveFrameFlags[Signal][Channel] = DEBUG_NEW int[mNumFeatureFrames[Signal]];
    mStartSamples[Signal][Channel] = CalculateActivityFlags(Signal, Channel,
mppActiveFrameFlags[Signal][Channel], mNumFeatureFrames[Signal]);
    mDataValidFlags[Signal][Channel] = true;

    return (int)((OTA_FLOAT)SearchStartFrame(Signal, Channel,
EarliestSamples/mProcessData.mStepSize) /
((OTA_FLOAT)Downsampling/(OTA_FLOAT)mProcessData.mStepSize));
}

int CSpeechActiveFrameDetection::GetLastActiveFrame(int Signal, int Channel, int
Downsampling, int EarliestSamples)
{
    if (!mDataValidFlags[Signal][Channel])
    {
        mppActiveFrameFlags[Signal][Channel] = DEBUG_NEW
int[mNumFeatureFrames[Signal]];
        mStartSamples[Signal][Channel] = CalculateActivityFlags(Signal, Channel,
mppActiveFrameFlags[Signal][Channel], mNumFeatureFrames[Signal]);
        mDataValidFlags[Signal][Channel] = true;
    }

    int LastActiveFrame = mNumFeatureFrames[Signal]-1;
    while(!mppActiveFrameFlags[Signal][Channel][LastActiveFrame--]);
    LastActiveFrame++;

    return (int)(
(OTA_FLOAT)LastActiveFrame*(OTA_FLOAT)mProcessData.mStepSize/(OTA_FLOAT)Downsamplin
g );
}

int CSpeechActiveFrameDetection::GetActiveFrameFlags(int Signal, int Channel, int
Downsampling, int* pFlags, int MaxNumFlags, int EarliestSample)
{
    if (!mDataValidFlags[Signal][Channel])
    {
        mppActiveFrameFlags[Signal][Channel] = DEBUG_NEW
int[mNumFeatureFrames[Signal]];
        mStartSamples[Signal][Channel] = CalculateActivityFlags(Signal, Channel,
mppActiveFrameFlags[Signal][Channel], mNumFeatureFrames[Signal]);
        mDataValidFlags[Signal][Channel] = true;
    }
```

```
    int* pActiveFrameFlags=0;
    pActiveFrameFlags = mppActiveFrameFlags[Signal][Channel];

    if (pActiveFrameFlags)
    {
        int i;
        int EarliestFrame = EarliestSample / mProcessData.mStepSize;

        if (Downsampling<mProcessData.mStepSize)
        {
            OTA_FLOAT fDownsampling = (OTA_FLOAT)Downsampling /
(OTA_FLOAT)mProcessData.mStepSize;
            int Hangover = mProcessData.mStepSize / Downsampling /2;
            bool LastFrameWasActive=false;
            for (i=EarliestFrame; i<MaxNumFlags; i++)
            {
                int ff = (int)((OTA_FLOAT)i * fDownsampling + (OTA_FLOAT)0.5);
                if (ff<mNumFeatureFrames[Signal])
                {
                    if (LastFrameWasActive&&!pActiveFrameFlags[ff])
                    {
                        int Start = (((EarliestFrame) > (i-Hangover)) ? (EarliestFrame)
: (i-Hangover));
                        for (int j=Start; j<=i; j++)
                            pFlags[j] = 0;
                        LastFrameWasActive = false;
                    }
                    else if (!LastFrameWasActive&&pActiveFrameFlags[ff])
                    {
                        int End = i+Hangover;
                        for (;i<MaxNumFlags && i<End; i++)
                            pFlags[i] = 0;
                        i++;
                        if (i<MaxNumFlags) pFlags[i] = 1;
                        LastFrameWasActive = true;
                    }
                    else LastFrameWasActive = pFlags[i] = pActiveFrameFlags[ff];
                }
                else pFlags[i] = 0;
            }

            return (((MaxNumFlags) < ((int)(mNumFeatureFrames[Signal]* fDownsampling)))
? (MaxNumFlags) : ((int)(mNumFeatureFrames[Signal]* fDownsampling)));

        }
        else
        {
            Downsampling /= mProcessData.mStepSize;

            int CenterOffset = Downsampling / 2 - 1;
            if (Downsampling==1) CenterOffset = 0;

            pFlags[0] = 0;

            int LastFrame = (((MaxNumFlags) <
((mNumFeatureFrames[Signal]-EarliestFrame-CenterOffset-1)/Downsampling)) ?
(MaxNumFlags) :
((mNumFeatureFrames[Signal]-EarliestFrame-CenterOffset-1)/Downsampling));
            pFlags[0] = 0;
            for (int mf=0; mf<LastFrame; mf++)
            {
                int Sum = 0;
                int StartFeatureFrame = (((0) > (mf * Downsampling - CenterOffset)) ?
(0) : (mf * Downsampling - CenterOffset));
                for (int s=0; s<Downsampling; s++)
                    Sum += pActiveFrameFlags[StartFeatureFrame+s+EarliestFrame];
                pFlags[mf] = Sum>CenterOffset ? 1 : 0;
            }

            for (i=LastFrame; i<MaxNumFlags; i++)
                pFlags[i] = 0;

            return LastFrame-1;
        }
    }
```

```
    else
    {
        for (int i=0; i<MaxNumFlags; i++)
            pFlags[i] = 1;

        return MaxNumFlags;
    }
}

void CSpeechActiveFrameDetection::ImproveSegments(SEGMENT* pSegments)
{
    for (int Signal=0; Signal<2; Signal++)
    {
        int Channel = 0;
        if (!mDataValidFlags[Signal][Channel])
        {
            mppActiveFrameFlags[Signal][Channel] = DEBUG_NEW
int[mNumFeatureFrames[Signal]];
            mStartSamples[Signal][Channel] = CalculateActivityFlags(Signal, Channel,
mppActiveFrameFlags[Signal][Channel], mNumFeatureFrames[Signal]);
            mDataValidFlags[Signal][Channel] = true;
        }
    }

    const OTA_FLOAT LowSNRdB=3;
    OTA_FLOAT SNRdB = 10*log10(mSignalLevels[1][0]/mNoiseLevels[1][0]);

    if (SNRdB>8 && SNRdB<20)
    {
        OTA_FLOAT* pVecRef = mFeatureList.GetFVector(0, 0, 0)->mpVector;
        OTA_FLOAT* pVecDeg = mFeatureList.GetFVector(0, 1, 0)->mpVector;

        OTA_FLOAT TriggerLevelRef = mSignalLevels[0][0]*2.0;
        OTA_FLOAT TriggerLevelDeg = mSignalLevels[1][0]*2.0;

        int TriggerPointRef;
        int TriggerEndRef   = (((SamplesToFrames(pSegments[0].End)) <
(mFeatureList.GetFVector(0, 0, 0)->mSize)) ?
(SamplesToFrames(pSegments[0].End)) : (mFeatureList.GetFVector(0, 0,
0)->mSize));
        do
        {
            TriggerPointRef = (((SamplesToFrames(pSegments[0].Start)) <
(mFeatureList.GetFVector(0, 0, 0)->mSize)) ?
(SamplesToFrames(pSegments[0].Start)) : (mFeatureList.GetFVector(0, 0,
0)->mSize));
            while(TriggerPointRef<TriggerEndRef &&
pVecRef[TriggerPointRef]<TriggerLevelRef) TriggerPointRef++;
            TriggerLevelRef *= 0.8;
        } while (TriggerPointRef>=TriggerEndRef);

        int TriggerPointDeg;
        int TriggerEndDeg   = (((SamplesToFrames(pSegments[0].End)) <
(mFeatureList.GetFVector(0, 0, 0)->mSize)) ?
(SamplesToFrames(pSegments[0].End)) : (mFeatureList.GetFVector(0, 0,
0)->mSize));
        do
        {
            TriggerPointDeg = (((SamplesToFrames(pSegments[0].Start)) <
(mFeatureList.GetFVector(0, 0, 0)->mSize)) ?
(SamplesToFrames(pSegments[0].Start)) : (mFeatureList.GetFVector(0, 0,
0)->mSize));
            while(TriggerPointDeg<TriggerEndDeg &&
pVecDeg[TriggerPointDeg]<TriggerLevelDeg) TriggerPointDeg++;
            TriggerLevelDeg *= 0.8;
        } while (TriggerPointDeg>=TriggerEndDeg);

        int DelayBetweenSegments = FramesToSamples(TriggerPointRef-TriggerPointDeg);
        int Correction = -pSegments[1].Start + pSegments[0].Start -
DelayBetweenSegments;

        if (mProcessData.mpLogFile)
            fprintf(mProcessData.mpLogFile, "---> Shifting deg segment by %d samples
(%dms)\n", Correction, SamplesToMSeconds(Correction) );

        pSegments[1].Start += Correction;
```

```
        pSegments[1].End   += Correction;
    }
}
}
```