```
    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 void CPitchBase::GetPitchVector(CProcessData *pProcessData, OTA_FLOAT* pSamples, int StartSample,
    int LastSample, OTA_FLOAT** ppPitchVector, int* PitchVecLen, int* PitchVecFrameSize, OTA_FLOAT*
    AvgPitchFreq, int *PitchStartOffset)
677 {
678     int i
679     OTA_FLOAT AvgPitch=0
680     OTA_FLOAT FloatingAvgPitchVals[10]
681     int FloatingAvgPitchCount
682
683     mpProcessData = pProcessData
684
685     int Framesize = (int)(0.032*mpProcessData->mSamplerate)
686
687     int Order = (int)(log((OTA_FLOAT)Framesize)/log(2.0))
688     int F1 = (int)(pow(2.0, Order))
689     int F2 = (int)(pow(2.0, Order+1))
690     if (Framesize-F1<F2-Framesize)
691         Framesize = F1
692     else
693         Framesize = F2
694
695     OTA_FLOAT FrameDuration = (OTA_FLOAT)Framesize / mpProcessData->mSamplerate * 1000.0
696
697     int FirstVoicedSample = 0
698     FirstVoicedSample = FirstVoicedSample % Framesize
699     *PitchStartOffset = FirstVoicedSample
700
701     int FrameCount = 0
702     int MaxFrames = (LastSample - StartSample - FirstVoicedSample) / Framesize
703     OTA_FLOAT* pPitchVector = matxMalloc(MaxFrames)
704     OTA_FLOAT* pCorrVector = matxMalloc(MaxFrames)
705
706     OTA_FLOAT* HammingWnd = (OTA_FLOAT*)matMalloc(Framesize * sizeof(OTA_FLOAT))
707
708     matbSet((OTA_FLOAT)1.0, HammingWnd, Framesize)
709     matWinHamming(HammingWnd, HammingWnd, Framesize)
710
711     FloatingAvgPitchCount = 0
712     OTA_FLOAT FltAvg=0
713     bool FltAvgValid=false
714     while (FrameCount < MaxFrames)
715     {
716         if (FltAvgValid)
717         {
718             FltAvg = 0
719             for (i = 0  i < 10  i++)
720                 FltAvg += FloatingAvgPitchVals[i]
721             FltAvg /= (OTA_FLOAT)10
722         }
723         else FltAvg = 0
724
725         pPitchVector[FrameCount] = Pitch(&pSamples[StartSample + FirstVoicedSample], Framesize,
    &pCorrVector[FrameCount], FltAvg, HammingWnd)
726         StartSample += Framesize
727
728         if (pPitchVector[FrameCount]>0)
729         {
730             FloatingAvgPitchVals[FloatingAvgPitchCount++] = pPitchVector[FrameCount]
731             if (FloatingAvgPitchCount>=10)
732             {
733                 FloatingAvgPitchCount = 0
```

```
                FltAvgValid = true
            }
        }
        FrameCount++
    }

    for (i=1  i<MaxFrames-1  i++)
    {
        if (pPitchVector[i]>0)
        {
            XFLOAT MaxPitchChange= 30
            XFLOAT Diff1 = fabs(fabs(pPitchVector[i-1]) - pPitchVector[i])
            XFLOAT Diff2 = fabs(fabs(pPitchVector[i+1]) - pPitchVector[i])

            if (Diff1>MaxPitchChange && Diff2>MaxPitchChange)
                pPitchVector[i] = -pPitchVector[i]
        }
    }

    OTA_FLOAT BinWidth=25.0
    int MaxBins = (int)(1000.0/BinWidth)
    int* PDF = new int[MaxBins]
    for (i=0  i<MaxBins  i++) PDF[i] = 0
    for (i=1  i<MaxFrames-1  i++)
        if (pPitchVector[i]>0.0 && pPitchVector[i]<1000.0)
            PDF[(int)(pPitchVector[i]/BinWidth+0.5)]++

    int MaxOfPDF = -1
    int IndexOfMaxOfPDF=-1
    for (i=0  i<MaxBins  i++)
        if (MaxOfPDF<PDF[i])
            {MaxOfPDF = PDF[i]  IndexOfMaxOfPDF = i }
    OTA_FLOAT MinPitchFreq=0.0
    OTA_FLOAT MaxPitchFreq=1000.0
    if (IndexOfMaxOfPDF>0)
    {

            MinPitchFreq = (IndexOfMaxOfPDF-1)*BinWidth-0.5*BinWidth
            if (MinPitchFreq<0) MinPitchFreq = 0
            MaxPitchFreq = (IndexOfMaxOfPDF+1)*BinWidth+0.5*BinWidth

    }

    int PitchCount=0
    AvgPitch = 0
    for (i=1  i<MaxFrames-1  i++)
        if (pPitchVector[i]>MinPitchFreq && pPitchVector[i]<MaxPitchFreq)
            {AvgPitch+=pPitchVector[i]  PitchCount++ }
    if (PitchCount)
        AvgPitch /= PitchCount

    *AvgPitchFreq = AvgPitch
    if (ppPitchVector)
    {
        *ppPitchVector = pPitchVector
        if (PitchVecLen) *PitchVecLen = MaxFrames
        if (PitchVecFrameSize) *PitchVecFrameSize = Framesize
    }
    else matFree(pPitchVector)
    matFree(pCorrVector)
    matFree(HammingWnd)
    delete[] PDF
}

int CPitchBase::GetFirstVoicedSample(OTA_FLOAT* pSamples, int StartSample, int LastSample)
{
    exit(1)

```

```
802     OTA_FLOAT* AvgPitchFreq
803
804     int i
805     OTA_FLOAT AvgPitch=0
806     OTA_FLOAT FloatingAvgPitchVals[10]
807     int FloatingAvgPitchCount
808
809     int Framesize = (int)(0.004*mpProcessData->mSamplerate)
810     int Order = (int)(log((OTA_FLOAT)Framesize)/log(2.0))
811     int F1 = (int)(pow(2.0, Order))
812     int F2 = (int)(pow(2.0, Order+1))
813     if (Framesize-F1<F2-Framesize)
814         Framesize = F1
815     else
816         Framesize = F2
817
818     OTA_FLOAT FrameDuration = (OTA_FLOAT)Framesize / mpProcessData->mSamplerate * 1000.0
819
820     int FrameCount = 0
821     int MaxFrames = (LastSample-StartSample) / Framesize
822     OTA_FLOAT CurrentPitch=0
823     OTA_FLOAT Correlation=0
824
825     OTA_FLOAT* HammingWnd = (OTA_FLOAT*)matMalloc(Framesize * sizeof(OTA_FLOAT))
826
827     matbSet((OTA_FLOAT)1.0, HammingWnd, Framesize)
828     matWinHamming(HammingWnd, HammingWnd, Framesize)
829
830     FloatingAvgPitchCount = 0
831     OTA_FLOAT FltAvg=0
832     bool FltAvgValid=false
833     while (FrameCount<MaxFrames && CurrentPitch<=0)
834     {
835         if (FltAvgValid)
836         {
837             FltAvg = 0
838             for (i=0  i<10  i++)
839                 FltAvg += FloatingAvgPitchVals[i]
840             FltAvg /= (OTA_FLOAT)10
841         }
842         else FltAvg = 0
843
844         CurrentPitch = Pitch(&pSamples[StartSample], Framesize, &Correlation, FltAvg, HammingWnd)
845         StartSample += Framesize
846
847         if (CurrentPitch>0)
848         {
849             FloatingAvgPitchVals[FloatingAvgPitchCount++] = CurrentPitch
850             if (FloatingAvgPitchCount>=10)
851             {
852                 FloatingAvgPitchCount = 0
853                 FltAvgValid = true
854             }
855         }
856         FrameCount++
857     }
858     matFree(HammingWnd)
859     return (FrameCount-1) * Framesize - Framesize>>1
860 }
861
862 OTA_FLOAT CPitchBase::Pitch( OTA_FLOAT* data, unsigned long frameLen, OTA_FLOAT* pCorr, OTA_FLOAT
    AvgPitch, OTA_FLOAT *HammingWnd)
863 {
864
865     OTA_FLOAT pitchFrequency = 0
866
867     OTA_FLOAT* pSamples=data
868     OTA_FLOAT* ClippedData = 0
```

```
869
870     if (2000.0 < Power(pSamples, frameLen))
871     {
872
873         const int order = matFFTOrder(frameLen*4)
874
875         const long fftBufferLen = 1<<order
876
877         const unsigned long SF = mpProcessData->mSamplerate
878         const OTA_FLOAT frequencyBinSize = SF/(OTA_FLOAT)fftBufferLen
879
880         SmartBuffer SB_data_temp(mpSmartBufferPool, (2 + fftBufferLen) * sizeof(OTA_FLOAT))
881         OTA_FLOAT* data_temp = SB_data_temp.Buffer
882
883         SmartBuffer SB_cmplxSpec(mpSmartBufferPool, (2 + fftBufferLen) * sizeof(OTA_FLOAT))
884         OTA_FLOAT* cmplxSpec = SB_cmplxSpec.Buffer
885
886         *pCorr = 0
887         MAT_HANDLE mh = (MAT_HANDLE) mpProcessData->mpMathlibHandle
888
889         OTA_FLOAT pitchPower = 0
890
891         matbMpy3(pSamples, HammingWnd, cmplxSpec, frameLen)
892         matbZero(cmplxSpec + frameLen, fftBufferLen - frameLen)
893
894         matRealFft(mh, cmplxSpec, order, MAT_Forw)
895
896         AbsoluteSpectrum(cmplxSpec, data_temp, fftBufferLen)
897
898         SB_cmplxSpec.Free()
899         cmplxSpec = 0
900
901         const OTA_FLOAT cutOffFrequency = 1250.0
902         int cutOffPoint
903         CutOffFrequency(data_temp, fftBufferLen, cutOffFrequency, &cutOffPoint)
904
905         Smooth(data_temp, fftBufferLen, cutOffPoint)
906
907         Subharmonics(data_temp, fftBufferLen/2, AvgPitch, &pitchPower, &pitchFrequency, cutOffPoint)
908
909         Voiced(data, frameLen, &pitchFrequency, pCorr)
910
911         if (ClippedData) matFree(ClippedData)
912     }
913
914     return pitchFrequency
915 }
916
917 void matdSpline_linearXAxis(double const *xk, double const *yk, int length,
918                             double y1k_left, double y1k_right, double *y2k,
919                             int cutOffPoint, double *u)
920 {
921     int i
922     double a, q_right, h_ratio, u_right
923     double linearXDiff = xk[1] - xk[0]
924
925     if (y1k_left > 0.99e30)
926         y2k[0] = u[0] = 0.0
927     else
928     {
929         y2k[0] = -0.5
930         u[0] = (3.0/(xk[1] - xk[0]))*((yk[1] - yk[0])/(xk[1] - xk[0]) - y1k_left)
931     }
932
933     for (i=1  i < length-1 && (i < cutOffPoint || fabs(u[i-1]) > 1e-99)  i++)
934
935     {
936         h_ratio = 0.5
```

```
        a = h_ratio*y2k[i-1] + 2.0
        y2k[i] = (h_ratio - 1.0)/a
        u[i] = ((yk[i+1] - yk[i]) - (yk[i] - yk[i-1])) / linearXDiff
        u[i] = (6.0*u[i]/(2*linearXDiff) - h_ratio*u[i-1])/a
    }
    for (  i < length-1  i++)
    {
        h_ratio = 0.5
        a = h_ratio*y2k[i-1] + 2.0
        y2k[i] = (h_ratio - 1.0)/a
        u[i] = -u[i-1]
    }
    if (y1k_right > 0.99e30)
        q_right = u_right = 0.0
    else
    {
        q_right = 0.5
        u_right =
(3.0/(xk[length-1]-xk[length-2]))*(y1k_right-(yk[length-1]-yk[length-2])/(xk[length-1]-xk[le
ngth-2]))
    }
    y2k[length-1] = (u_right - q_right*u[length-2])/(q_right*y2k[length-2] + 1.0)
    for (i = length-2  i >= 0  i--)
        y2k[i] = y2k[i]*y2k[i+1] + u[i]
}

void CPitchBase::Subharmonics(OTA_FLOAT* data, int dataLen, OTA_FLOAT AvgPitch, OTA_FLOAT*
maxPitchPower, OTA_FLOAT* maxPitchFrequency,
                              int cutOffPoint)
{

    //Compute the pitch of the dataLen long signal section in data.
    //The algorithm to be used is described in Dik J. Hermes, Measurement of pitch by subharmonic
summation, Acoust.Soc.Am 83(1), January 1988

}

void CPitchBase::Smooth(OTA_FLOAT* data, int dataLen, int cutOffPoint)
{
    int i,j

    int lastPeak =-2

    dataLen/=2

    data[0]=0
    data[dataLen-1]=0
    for(i=1  i<dataLen-1  i++)
    {
        if((data[i]>data[i-1]) && (data[i]>=data[i+1]))
        {

            for(j=lastPeak+3 j<i-2 j++)

            data[j]=0

            lastPeak = i
        }
    }

    for(i=1  i < dataLen-1 && (i < cutOffPoint || fabs(data[i-1]) > 1e-99)  i++)

    {

        data[i]=0.14*data[i-1]+0.72*data[i]+0.14*data[i+1]

    }
}
```

```
1001
1002 void CPitchBase::AbsoluteSpectrum(OTA_FLOAT* cmplxSpec, OTA_FLOAT *absSpec, int bufferLen)
1003 {
1004
1005     for (int i = 0  i < bufferLen/2  i++)
1006         absSpec[i] = cmplxSpec[2*i]*cmplxSpec[2*i] + cmplxSpec[i*2+1]*cmplxSpec[i*2+1]
1007     matbSqrt1(absSpec, bufferLen/2)
1008     matbZero(absSpec + bufferLen/2, bufferLen - bufferLen/2)
1009
1010 }
1011
1012 void CPitchBase::CutOffFrequency(OTA_FLOAT* data,int dataLen, OTA_FLOAT cutOffFrequency,
1013                                  int *cutOffPoint)
1014 {
1015     OTA_FLOAT SF = mpProcessData->mSamplerate
1016     *cutOffPoint = (int)(cutOffFrequency / (SF/(OTA_FLOAT)dataLen))
1017     matbZero(data + *cutOffPoint, dataLen/2 - *cutOffPoint)
1018
1019 }
1020
1021 void CPitchBase::AbsoluteRealSpectrum(OTA_FLOAT* data, int dataLen)
1022 {
1023     int i
1024
1025     for(i=0 i<dataLen i++)
1026     {
1027         if (i<dataLen/2)
1028             data[i]=fabs(data[2*i])
1029         else
1030             data[i]=0
1031     }
1032
1033 }
1034
1035 OTA_FLOAT CPitchBase::Power(OTA_FLOAT* data, long dataLen)
1036 {
1037     OTA_FLOAT power = 0
1038
1039     power = matbNormL2(data, dataLen)
1040     power = power*power
1041 if (dataLen>0)
1042     return power/dataLen
1043 else
1044     return 0
1045 }
1046
1047 void CPitchBase::Voiced(OTA_FLOAT* data, long dataLen, OTA_FLOAT* pitchFrequency, OTA_FLOAT* corr)
1048 {
1049
1050     unsigned long SF = mpProcessData->mSamplerate
1051
1052     long T = (long) min (dataLen/3.0, SF/(max(1.0,*pitchFrequency)))
1053
1054     OTA_FLOAT delay =0
1055
1056     if (DelayEstimate(data, &data[2*T], 2*T, T, &delay, corr, 0) != 0)
1057         *corr = (OTA_FLOAT)-1.0
1058     else
1059         *corr = max((OTA_FLOAT)0.0, *corr)
1060
1061     if (*pitchFrequency < 0){
1062         *pitchFrequency *= -1
1063     }
1064
1065     if (*corr >= (OTA_FLOAT)0.0 && *corr < (OTA_FLOAT)0.52) *pitchFrequency = 0
1066     if (*pitchFrequency != 0){
1067         *pitchFrequency = *pitchFrequency
1068     }
```

```
    if (*pitchFrequency > 500.0) *pitchFrequency = 0
    if (*pitchFrequency < 60.0) *pitchFrequency = 0
}

long CPitchBase::Order(long number)
{
    long powerOf2 = 1
    long order = 0
    while (powerOf2 < number)
    {
        powerOf2 *= 2
        order++
    }
    return order
}

int CPitchBase::DelayEstimate( OTA_FLOAT * ref, OTA_FLOAT * test, long ref_len, long test_len,
OTA_FLOAT* delay, OTA_FLOAT* corr, int win)
{
    const long Y_len = ref_len+test_len-1

    OTA_FLOAT * Y = (OTA_FLOAT*)matCalloc(Y_len, sizeof(OTA_FLOAT))
    OTA_FLOAT *ref_temp  = (OTA_FLOAT*)matCalloc(ref_len, sizeof(OTA_FLOAT))
    OTA_FLOAT *test_temp  = (OTA_FLOAT*)matCalloc(test_len, sizeof(OTA_FLOAT))

    matbCopy(ref, ref_temp, ref_len)
    matbCopy(test, test_temp, test_len)

    OTA_FLOAT mean_ref = matMean(ref, ref_len)
    OTA_FLOAT mean_test = matMean(test, test_len)

    matbAdd1(-mean_ref, ref_temp, ref_len)
    matbAdd1(-mean_test, test_temp, test_len)

    OTA_FLOAT stdDev_ref = matStdDev(ref_temp, ref_len)
    OTA_FLOAT stdDev_test = matStdDev(test_temp, test_len)

    int I_maxInt = 0
    int retVal   = 0
    if (stdDev_ref > (OTA_FLOAT)0.0 && stdDev_test > (OTA_FLOAT)0.0 && Y_len > 1)
    {

        matCrossCorr((MAT_HANDLE)(mpProcessData->mpMathlibHandle), ref_temp, ref_len, test_temp,
test_len, Y, Y_len, -ref_len)
        matbMpy1(1/(stdDev_ref*stdDev_test*(Y_len-1)/2), Y, Y_len)
        *corr = matMaxExt (Y, Y_len, &I_maxInt)

    }
    else
    {
        *corr    = (OTA_FLOAT)0.0
        I_maxInt = 0
        retVal   = -1
    }

    *delay = I_maxInt-ref_len

    return retVal
}
}
```