

```

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 mSNRRRefdB, 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
{
class CPitchFeature : public CFeature, CPitchBase
{
public:
    CPitchFeature()
    {
        for (int i=0; i<2*2; i++)
            mpFeatureVectors[i] = 0;
        mNumFVectors = 0;
    };

    virtual ~CPitchFeature() {Free();};
    void Free();
    virtual bool CalcVector(CTASignal** pSignals, CProcessData* pProcessData,
SEGMENT* pSegments, int* pActiveFrameFlags);
    virtual CFeatureVector* GetFVector(int IndexOfVector);

private:
    bool CalcFeatureForOneChannel(int FVecIndex, CAudioSignal* Sig, int Channel,
SEGMENT* pSegment, int* pActiveFrameFlags);

```

```

    CFeatureVector *mpFeatureVectors[2*2];
    CProcessData mProcessData;

    unsigned int mStepSize;
    unsigned long mpFLength[2*2];

    int mNumFVectors;
};

void CPitchFeature::Free()
{
    for (int i=0; i<2*2; i++)
    {
        mpFeatureVectors[i] = 0;
        mpFLength[i] = 0;
    }
    mNumFVectors = 0;
    mChannels = 0;
}

CFeature* CreatePitchFeature()
{
    return (CFeature*) new CPitchFeature;
}

bool CPitchFeature::CalcFeatureForOneChannel(int FVecIndex, CAudioSignal* Sig, int
Channel, SEGMENT* pSegment, int* pActiveFrameFlags)
{
    bool rc = true;

    unsigned long StartSample=pSegment->Start;
    unsigned long FrameCount=0;

    OTA_FLOAT* pSamples = Sig->mpData[Channel+2];

    int Framesize = mProcessData.mWindowSize;
    int Overlap = mProcessData.mOverlap;
    unsigned long LastStart=((pSegment->End-StartSample-Framesize+1) / mStepSize) *
mStepSize+StartSample;
    unsigned long SF = Sig->mSampleRate;

    mpFLength[FVecIndex] = (pSegment->End-StartSample-Framesize+1) / mStepSize;
    mpFeatureVectors[FVecIndex] = new CFeatureVector(mpFLength[FVecIndex]);
    if (!mpFeatureVectors[FVecIndex]) rc = false;

    OTA_FLOAT* pPitchVector = 0;
    int PitchVecLen = 0;
    int PitchVecFrameSize = 0;
    OTA_FLOAT AvgPitchFreq;
    int PitchStart = 0;
    GetPitchVector(&mProcessData, pSamples, StartSample, pSegment->End, &pPitchVector,
&PitchVecLen, &PitchVecFrameSize, &AvgPitchFreq, &PitchStart);

    if(pActiveFrameFlags)
    {
        while (StartSample<LastStart)
        {
            if(pActiveFrameFlags[FrameCount])
                mpFeatureVectors[FVecIndex]->mpVector[FrameCount] =
pPitchVector[max(0,min(PitchVecLen-1,
((int)StartSample-PitchStart+PitchVecFrameSize/2)/PitchVecFrameSize));]
            else

                ;//Set the element to 0!

            StartSample += mStepSize;
            FrameCount++;
        }
    }
    else
    {
        while (StartSample<LastStart)
        {
            mpFeatureVectors[FVecIndex]->mpVector[FrameCount] =
pPitchVector[max(0,min(PitchVecLen-1,

```

```

((int)StartSample-PitchStart+PitchVecFrameSize/2)/PitchVecFrameSize));
    StartSample += mStepSize;
    FrameCount++;
}
}

matFree(pPitchVector);

mNumFVectors++;

return rc;
}

bool CPitchFeature::CalcVector(CTASignal** pSignalsIn, CProcessData* pProcessData,
SEGMENT* pSegments, int* pActiveFrameFlags)
{
    bool rc = true;

    CAudioSignal *pSignals[2];
    pSignals[0] = (CAudioSignal*)(pSignalsIn[0]);
    pSignals[1] = (CAudioSignal*)(pSignalsIn[1]);

    Free();

    mProcessData = *pProcessData;
    mChannels = min(pSignals[0]->mNumChannels, pSignals[1]->mNumChannels);
    mStepSize = mProcessData.mWindowSize - mProcessData.mOverlap;

    if (rc)
    {
        int VecIndex=0;
        for (int s=0; rc && s<mProcessData.mNumSignals; s++)
        {
            if (pSegments[s].Start<0) rc=false;
            if (pSegments[s].End<0) rc=false;
            if (pSegments[s].End>pSignals[s]->mSignalLength) rc=false;
            if (pSegments[s].Start>pSignals[s]->mSignalLength-1) rc=false;
            if (pSegments[s].End-pSegments[s].Start<mProcessData.mWindowSize) rc
=false;

            for (int c=0; rc && c<mChannels; c++)
            {
                rc = CalcFeatureForOneChannel(VecIndexFromChannel(s, c), pSignals[s], c,
&pSegments[s], s == 1 ? pActiveFrameFlags : 0);
            }
        }
        else Free();

        return rc;
    }
}

CFeatureVector* CPitchFeature::GetFVector(int IndexOfVector)
{
    return (mpFeatureVectors[IndexOfVector]);
}
}

```