```
    typedef double XFLOAT;
    typedef double OTA_FLOAT;

using namespace std;

typedef std::vector<XFLOAT>FVector;

/*****************************************************************************/

static XFLOAT const TA_OVERLAPFAC     = (XFLOAT)0.5;
static int   const TA_WINLEN_SIGNAL   =
    213 * TA_SAMPLING_RATE / 8000;
static int   const TA_MIN_SEGLEN      = 512;
static int   const TA_SHIFT_TOLERANCE =
    round(0.005f * TA_SAMPLING_RATE);
static XFLOAT const TA_INFINITE_CORR  = (XFLOAT)100.0;

namespace SQFUNCS_POLQA_INTERNAL
{
/*****************************************************************************/

SQTA_ResampResult::SQTA_ResampResult ()
{
    type = kRESNone;
    fMeanResamplingFac = 1.0f;
    fResamplingFactors = NULL;
    iNumSentences = 0;
    iNumTA = 0;
}

SQTA_ResampResult::SQTA_ResampResult (SQTA_ResampResult const *toCopy)
{
    type = toCopy->type;
    fMeanResamplingFac = toCopy->fMeanResamplingFac;
    iNumSentences = toCopy->iNumSentences;
    iNumTA = toCopy->iNumTA;
    if (iNumSentences)
    {
        fResamplingFactors = (XFLOAT*)matMalloc(iNumSentences * sizeof(XFLOAT));
        for (int i = 0; i < iNumSentences; i++)
        {
            fResamplingFactors[i] = toCopy->fResamplingFactors[i];
        }
    }
    else
    {
        fResamplingFactors = NULL;
    }
}

SQTA_ResampResult::~SQTA_ResampResult ()
{
    if(fResamplingFactors)
        matFree(fResamplingFactors);
}

void SQTA_ResampResult::assign (SQTA_ResampResult const *toCopy)
{
    type = toCopy->type;
    fMeanResamplingFac = toCopy->fMeanResamplingFac;
    iNumSentences = toCopy->iNumSentences;
    matFree(fResamplingFactors);
    if (iNumSentences)
    {
        fResamplingFactors = (XFLOAT*)matMalloc(iNumSentences * sizeof(XFLOAT));
        for (int i = 0; i < iNumSentences; i++)
        {
            fResamplingFactors[i] = toCopy->fResamplingFactors[i];
        }
    }
    else
    {
        fResamplingFactors = NULL;
    }
```

```
}

/*****************************************************************************/

SQTimeAlignment::SQTimeAlignment (SQTimeAlignment const &inputTA, bool copySignals,
MAT_HANDLE inMatHandle, FILE* pLogFile)
:   mRef                (NULL),
    mDeg                (NULL),
    mSNRDeg             ((XFLOAT)40.0),
    mMatchQuality       (0.0),
    mExtremeMatchFound  (false),
    mMinDelay           (0),
    mMaxDelay           (0),
    mRefLen             (-1),
    mDegLen             (-1),
    mTargetLen          (-1),
    mTargetRate         (-1),
    mTargetBitRes       (-1),
    mActSpeechThr       ((XFLOAT)-45.0),
    mNumSentences       (-1),
    mCrudeDelay         (0),
    mCurResolution      (-1),
    mCurSegmentRate     (-1),
    mSegments           (NULL),
    mMergedSegments     (NULL),
    mUnusedDegSegments  (NULL),
    mMaxSigLenBuff      (NULL),
    matHandle           (inMatHandle),
    mpLogFile           (pLogFile)
{
    OPTTRY
    {
        mSNRDeg             = inputTA.SNRDeg();
        mMatchQuality       = inputTA.MatchQuality();
        mExtremeMatchFound  = inputTA.ExtremeMatchFound();
        mMinDelay           = inputTA.MinDelay();
        mMaxDelay           = inputTA.MaxDelay();
        mTargetLen          = inputTA.TargetLen();
        mTargetRate         = inputTA.SamplingRate();
        mTargetBitRes       = inputTA.TargetBitRes();
        mActSpeechThr       = inputTA.ActiveSpeechThr();
        mCrudeDelay         = inputTA.CrudeDelay();
        mCurResolution      = inputTA.CurResolution();
        mRefLen             = inputTA.RefLen();
        mDegLen             = inputTA.DegLen();
        mCurSegmentRate     = inputTA.CurSegmentRate();
        mNumSentences       = inputTA.NumSentences();

        if (copySignals)
        {
            mRef = new SQSignal (*inputTA.refSignal(),
inputTA.refSignal()->SamplingFreq(), inputTA.refSignal()->BitResolution());
            mDeg = new SQSignal (*inputTA.degSignal(),
inputTA.degSignal()->SamplingFreq(), inputTA.degSignal()->BitResolution());
            mRef->SetNumPause (inputTA.refSignal()->NumPause());
            mRef->SetPauseCenter (inputTA.refSignal()->PauseCenter(),
inputTA.refSignal()->SamplingFreq());
            mDeg->SetNumPause (inputTA.degSignal()->NumPause());
            mDeg->SetPauseCenter (inputTA.degSignal()->PauseCenter(),
inputTA.degSignal()->SamplingFreq());
        }

        mSegments = new TA_SegList();
        mSegments->assign (inputTA.Segments()->begin(), inputTA.Segments()->end());

        mMergedSegments = new TA_SegList();
        mMergedSegments->assign (inputTA.MergedSegments()->begin(),
inputTA.MergedSegments()->end());

        mUnusedDegSegments = new TA_SegList();
        mUnusedDegSegments->assign (inputTA.UnusedDegSegments()->begin(),
inputTA.UnusedDegSegments()->end());

    }
    OPTCATCH((string errorMsg))
    {
```

```
        delete mRef;
        delete mDeg;
        delete mSegments;
        delete mMergedSegments;
        delete mUnusedDegSegments;
        OPTTHROW(( string("ERROR in SQTimeAlignment::SQTimeAlignment: " + errorMsg +
"\n")));
    }
    OPTCATCH( (...))
    {
        delete mRef;
        delete mDeg;
        delete mSegments;
        delete mMergedSegments;
        delete mUnusedDegSegments;
        OPTTHROW ((string("Unspecified error in SQTimeAlignment::SQTimeAlignment.")));
    }
}
SQTimeAlignment::SQTimeAlignment (SQSignal const &sigRef, SQSignal const &sigDeg,
                                  XFLOAT * const maxSigLenBuff, XFLOAT degResampFac,
MAT_HANDLE inMatHandle, FILE* pLogFile)
:   mRef                (NULL),
    mDeg                (NULL),
    mSNRDeg             ((XFLOAT)40.0),
    mMatchQuality       (0.0),
    mExtremeMatchFound  (false),
    mMinDelay           (0),
    mMaxDelay           (0),
    mRefLen             (-1),
    mDegLen             (-1),
    mTargetLen          (-1),
    mTargetRate         (-1),
    mTargetBitRes       (-1),
    mActSpeechThr       ((XFLOAT)-45.0f),
    mNumSentences       (-1),
    mCrudeDelay         (0),
    mCurResolution      (-1),
    mCurSegmentRate     (-1),
    mSegments           (NULL),
    mMergedSegments     (NULL),
    mUnusedDegSegments  (NULL),
    mMaxSigLenBuff      (maxSigLenBuff),
    matHandle           (inMatHandle),
    mpLogFile           (pLogFile)
{
    OPTTRY
    {
        if (sigRef.Data() == NULL || sigDeg.Data() == NULL ||
            (sigRef.End()-sigRef.Start()) / sigRef.SamplingFreq() < MIN_SPEECH_DURATION
||
            (sigDeg.End()-sigDeg.Start()) / sigDeg.SamplingFreq() < MIN_SPEECH_DURATION
||
            sigRef.Start() < 0 || sigDeg.Start() < 0 ||
            sigRef.BitResolution() <= 2 || sigDeg.BitResolution() <= 2)
            OPTTHROW (string("Input signals inexistent / invalid / activity too
short."));
        if (!sigRef.Preprocessed() || !sigDeg.Preprocessed())
            OPTTHROW (string("Input signals must undergo preprocessing prior to time
alignment."));
        if (sigRef.SamplingFreq() != sigDeg.SamplingFreq())
            OPTTHROW (string("Input signals must have same Fs before time
alignment."));

        mTargetLen    = sigRef.NrOfSamples();
        mTargetRate   = sigRef.SamplingFreq();
        mTargetBitRes = sigRef.BitResolution();

        //Preprocess ref and deg signal

        ScaleFilterAndResample(sigRef, mRefLen, mRef);

        mRef->SlidingWinMeanRemoval(TA_WINLEN_SIGNAL);

        mNumSentences  = sigRef.NumPause() + 1;

        ScaleFilterAndResample(sigDeg, mDegLen, mDeg, degResampFac);
```

```
        mDeg->SlidingWinMeanRemoval(TA_WINLEN_SIGNAL);

        //Compute common threshold for active speech
        CalcActSpeechThr(sigDeg);

        //Estimate overall delay
        CalcCrudeDelay();

        //Refine overall delay estimation
        RefineCrudeDelay();

        //Compute segment-wise delay.
        FineDelay();

        //Scale crude delay to mTargetRate, seg lists already scaled in FineDelay().
        mCrudeDelay *= mTargetRate/TA_SAMPLING_RATE;

        //Compute min and max delay in the entire signal
        CalcDelaySpread();
    }
    OPTCATCH( (string errorMsg))
    {
        delete mRef;
        delete mDeg;
        delete mSegments;
        delete mMergedSegments;
        delete mUnusedDegSegments;
        OPTTHROW(( string("ERROR in SQTimeAlignment::SQTimeAlignment: " + errorMsg +
"\n")));
    }
    OPTCATCH( (...))
    {
        delete mRef;
        delete mDeg;
        delete mSegments;
        delete mMergedSegments;
        delete mUnusedDegSegments;
        OPTTHROW(( string("Unspecified error in SQTimeAlignment::SQTimeAlignment.")));
    }
}

SQTimeAlignment::~SQTimeAlignment()
{
    delete mRef;
    delete mDeg;
    delete mSegments;
    delete mMergedSegments;
    delete mUnusedDegSegments;

    mRef = mDeg = NULL;
    mSegments = mMergedSegments = mUnusedDegSegments = NULL;
}

XFLOAT SQTimeAlignment::SNRDeg() const
{
    return mSNRDeg;
}

XFLOAT SQTimeAlignment::MatchQuality() const
{
    return mMatchQuality;
}

bool SQTimeAlignment::ExtremeMatchFound() const
{
    return mExtremeMatchFound;
}

int SQTimeAlignment::MinDelay() const
{
    return mMinDelay;
}

int SQTimeAlignment::MaxDelay() const
{
```

```
    return mMaxDelay;
}

long SQTimeAlignment::RefLen() const
{
    return mRefLen;
}

long SQTimeAlignment::DegLen() const
{
    return mDegLen;
}

long SQTimeAlignment::TargetLen() const
{
    return mTargetLen;
}

int SQTimeAlignment::TargetBitRes() const
{
    return mTargetBitRes;
}

XFLOAT SQTimeAlignment::ActiveSpeechThr() const
{
    return mActSpeechThr;
}

long SQTimeAlignment::CurResolution() const
{
    return mCurResolution;
}

int SQTimeAlignment::CrudeDelay() const
{
    return mCrudeDelay;
}

int SQTimeAlignment::SamplingRate() const
{
    return mTargetRate;
}

short SQTimeAlignment::NumSentences() const
{
    return mNumSentences;
}

int SQTimeAlignment::CurSegmentRate() const
{
    return mCurSegmentRate;
}

TA_SegList const* SQTimeAlignment::Segments() const
{
    return mSegments;
}

TA_SegList const* SQTimeAlignment::MergedSegments() const
{
    return mMergedSegments;
}

TA_SegList const* SQTimeAlignment::UnusedDegSegments() const
{
    return mUnusedDegSegments;
}

SQSignal const* SQTimeAlignment::refSignal() const
{
    return mRef;
}

SQSignal const* SQTimeAlignment::degSignal() const
{
    return mDeg;
```

```
}

/*****************************************************************************/

void SQTimeAlignment::ScaleFilterAndResample(SQSignal const &Input,
                                             long &lResLen,
                                             SQSignal* &Output,
                                             XFLOAT resampFac)
{
    int      iOrigFs        = Input.SamplingFreq();
    int      iGCD           = gcd(iOrigFs, TA_SAMPLING_RATE);
    XFLOAT   *fDelayLine    = NULL;
    XFLOAT   *fPaddedInput  = NULL;
    XFLOAT   *fResampledData= NULL;
    XFLOAT   *dFiltCoeffs   = NULL;
    int      iTapsLen       = 0;

    OPTTRY
    {
        if (iGCD <= 0 || Input.Data() == NULL || Input.NrOfSamples() <= 1 ||
            iOrigFs < TA_SAMPLING_RATE || iOrigFs < 1 || resampFac <= 0.0f)
            OPTTHROW (string("Invalid input arguments / signal data."));

        int     iUpFac      = TA_SAMPLING_RATE / iGCD;
        int     iDownFac    = iOrigFs / iGCD;

        if (iUpFac != 1) //Downsampling only, upsampling not supported
        {
            stringstream errorStream;
            errorStream << "Signal rate must be an integer multiple of " <<
TA_SAMPLING_RATE << ".";
            OPTTHROW( errorStream.str());
        }
        if (iOrigFs != 32000) //Cannot use precomputed filter taps, compute new ones
        {

            XFLOAT dLowFrq  =  700.0 / iOrigFs;
            XFLOAT dHighFrq = 3000.0 / iOrigFs;

            iTapsLen        = (((iDownFac*64) < (512)) ? (iDownFac*64) : (512));

            if(mpLogFile)
            {
                fprintf(mpLogFile, "Bandpass Taps: nrTaps %d\n", iTapsLen);
            }
            switch(iOrigFs)
            {
                case 8000:
                    dFiltCoeffs = sqBPTaps8k;
                    break;

                case 16000:
                    dFiltCoeffs = sqBPTaps16k;
                    break;

                case 48000:
                    dFiltCoeffs = sqBPTaps48k;
                    break;

                default:

                    if(mpLogFile)
                    {
                        fprintf(mpLogFile, "Dynamic Generation of Taps\n");
                    }
                    dFiltCoeffs     = (XFLOAT*)matMalloc(iTapsLen * sizeof(XFLOAT));
                    if (matGenBandPassCoefficients(dLowFrq, dHighFrq, dFiltCoeffs,
iTapsLen, MAT_WinBlackman) != 0)
                        OPTTHROW( string("Bandpass generation in mathlib failed."));
                break;
            }

        }
        else //Use precomputed filter taps
        {
            iTapsLen    = TA_INITS_iBPCoeffLen_32k;
```

```
            dFiltCoeffs = TA_INITS_dBPCoeffs_32k;
        }

        long lPaddedLen = Input.NrOfSamples() + 2*iTapsLen;
        int  iNumIters  = lPaddedLen / iDownFac;
        XFLOAT fMaxAmplitude = (((MAX_AMP_32BIT) < (pow((XFLOAT)2.0,
Input.BitResolution() - 1))) ? (MAX_AMP_32BIT) : (pow((XFLOAT)2.0,
Input.BitResolution() - 1)));
        XFLOAT fScaleFac = pow((XFLOAT)2.0, STD_BIT_RESOLUTION - 1) / fMaxAmplitude;

        fPaddedInput    = (XFLOAT*)matMalloc(lPaddedLen * sizeof(XFLOAT));
        fResampledData  = (XFLOAT*)matMalloc(iNumIters * iUpFac * sizeof(XFLOAT));

        matbZero(fPaddedInput, lPaddedLen);
        vsmul(Input.Data(), fScaleFac, fPaddedInput+iTapsLen, Input.NrOfSamples());

        matRunFIRMRFilter(fPaddedInput, fResampledData, iNumIters, dFiltCoeffs,
iTapsLen, iUpFac, iDownFac);

        if (Output != NULL)
            delete Output;
        lResLen     = Input.NrOfSamples() / iDownFac;
        int iOffSet = (((iNumIters-lResLen) < (iTapsLen/iDownFac + iTapsLen/2/iDownFac
+ 1)) ? (iNumIters-lResLen) : (iTapsLen/iDownFac + iTapsLen/2/iDownFac + 1));
        Output      = new SQSignal(fResampledData + iOffSet, lResLen,
                                   TA_SAMPLING_RATE, STD_BIT_RESOLUTION);

        if (dFiltCoeffs &&
            dFiltCoeffs != TA_INITS_dBPCoeffs_32k &&
            dFiltCoeffs != sqBPTaps8k &&
            dFiltCoeffs != sqBPTaps16k &&
            dFiltCoeffs != sqBPTaps48k)
            matFree(dFiltCoeffs);
        dFiltCoeffs = NULL;

        if(fPaddedInput)
            matFree(fPaddedInput);
        if(fResampledData)
            matFree(fResampledData);
        fPaddedInput = fResampledData = NULL;
    }
    OPTCATCH((...))
    {
        matFree(fPaddedInput);
        matFree(fResampledData);
        fPaddedInput = fResampledData = NULL;
        if(dFiltCoeffs && dFiltCoeffs != TA_INITS_dBPCoeffs_32k && dFiltCoeffs !=
sqBPTaps8k && dFiltCoeffs != sqBPTaps16k && dFiltCoeffs != sqBPTaps48k)
        {
            matFree(dFiltCoeffs);
            dFiltCoeffs = NULL;
        }

        OPTTHROW (string("ERROR in SQTimeAlignment::ScaleFilterAndResample using
Mathlib\n"));
    }
}

void SQTimeAlignment::CalcActSpeechThr(SQSignal const &originalDegSig)
{
    if (mRef == NULL || mDeg == NULL || mRef->Data() == NULL || mDeg->Data() == NULL)
        OPTTHROW (string("SQTimeAlignment::CalcActSpeechThr failed.\n"));

    XFLOAT const FRAMELEN = (XFLOAT)0.032;
    mRef->CalcEnvelope(FRAMELEN, TA_OVERLAPFAC, MIN_LEVEL_DB, mMaxSigLenBuff);
    mDeg->CalcEnvelope(FRAMELEN, TA_OVERLAPFAC, MIN_LEVEL_DB, mMaxSigLenBuff);

    //Estimate the noise level
    mRef->CalcASLandNoiseLevel(FRAMELEN, mMaxSigLenBuff);
    mDeg->CalcASLandNoiseLevel(FRAMELEN, mMaxSigLenBuff);

    //The bandpass in the constructor may leave too little speech in case of strong BG
noise.
    if (mDeg->CurrentASL() - mDeg->CurrentNoiseLevel() < 3.0f)
    {
        delete mDeg;
```

```
        mDeg = new SQSignal(originalDegSig);
        mDeg->Resample(TA_SAMPLING_RATE);
        mDegLen = mDeg->NrOfSamples();
        mDeg->SlidingWinMeanRemoval(TA_WINLEN_SIGNAL);

        mDeg->CalcEnvelope(FRAMELEN, TA_OVERLAPFAC, MIN_LEVEL_DB, mMaxSigLenBuff);
        mDeg->CalcASLandNoiseLevel(FRAMELEN, mMaxSigLenBuff);
    }

    mSNRDeg = mDeg->CurrentASL() - mDeg->CurrentNoiseLevel();

    mRef->LevelAlign(REF_ASL_LEVEL);
    mDeg->LevelAlign(REF_ASL_LEVEL);

    //Set the threshold for active speech, taking the max between ref and deg.
    mActSpeechThr = ((((mRef->CurrentASL() + 3 * mRef->CurrentNoiseLevel()) / 4.0f) >
((mDeg->CurrentASL() + 3 * mDeg->CurrentNoiseLevel()) / 4.0f)) ?
((mRef->CurrentASL() + 3 * mRef->CurrentNoiseLevel()) / 4.0f) :
((mDeg->CurrentASL() + 3 * mDeg->CurrentNoiseLevel()) / 4.0f));
    mActSpeechThr = (((mRef->CurrentASL()-3.0f) < (mActSpeechThr)) ?
(mRef->CurrentASL()-3.0f) : (mActSpeechThr));
    return;
}

void SQTimeAlignment::CalcCrudeDelay()
{
    XFLOAT const FCORRTHRESH      = (XFLOAT)0.42;
    XFLOAT      fMaxCorr;
    int         iDelay;
    XFLOAT      fTA_framelen  = (XFLOAT)0.18;
    XFLOAT const    FMINFRAMESTEP = (XFLOAT)0.008;
    int         iNrOfShifts   =
        round((2*(((mRef->NrOfSamples()) > (mDeg->NrOfSamples())) ?
(mRef->NrOfSamples()) : (mDeg->NrOfSamples())) - mRef->NrOfSamples()/2)
              /(fTA_framelen*TA_SAMPLING_RATE*TA_OVERLAPFAC)) | 0x1;
    int const   ISTEPSIZE     = 3;
    XFLOAT const WINLEN_ENV   = (XFLOAT)0.45;

    XFLOAT *fRefEnvNorm = NULL, *fDegEnvNorm = NULL, *fFBStmp1 = NULL, *fFBStmp2 =
NULL, *fSWPNtmp = NULL;

    OPTTRY
    {
        if (mRef == NULL || mDeg == NULL || mActSpeechThr >= 0.0f ||
            mRef->Data() == NULL || mDeg->Data() == NULL)
            OPTTHROW( string("Invalid or NULL signals or parameters."));

        int maxNumberOfFrames = (int)ceil((((mRef->NrOfSamples()) >
(mDeg->NrOfSamples())) ? (mRef->NrOfSamples()) : (mDeg->NrOfSamples())) /
            (FMINFRAMESTEP*TA_SAMPLING_RATE));
        if ((fFBStmp1 = (XFLOAT*)matMalloc(iNrOfShifts*sizeof(XFLOAT))) == NULL ||
            (fFBStmp2 = (XFLOAT*)matMalloc(iNrOfShifts*sizeof(XFLOAT))) == NULL ||
            (fSWPNtmp = (XFLOAT*)matMalloc(maxNumberOfFrames*sizeof(XFLOAT))) == NULL)
            OPTTHROW((string("ippsMalloc failed.")));

        mCrudeDelay = 0;

        //Find start and end of signal activity in Ref.
        mRef->FindActBoundaries(mActSpeechThr, 5, FMINFRAMESTEP, 0, false);

        for (; fTA_framelen * TA_OVERLAPFAC > FMINFRAMESTEP; fTA_framelen /= ISTEPSIZE)
        {
            XFLOAT curEnvStepSize = fTA_framelen * TA_OVERLAPFAC * TA_SAMPLING_RATE;
            int   refOffset      = (int)(mRef->Start() / curEnvStepSize);

            //Threshold the signal envelopes using the active speech threshold
            mRef->CalcEnvelope(fTA_framelen, TA_OVERLAPFAC, mActSpeechThr,
mMaxSigLenBuff);
            mDeg->CalcEnvelope(fTA_framelen, TA_OVERLAPFAC, mActSpeechThr,
mMaxSigLenBuff);
            vsadd(mRef->Env(), -mActSpeechThr, mRef->Env(), mRef->NrOfFrames());
            vsadd(mDeg->Env(), -mActSpeechThr, mDeg->Env(), mDeg->NrOfFrames());

            //Apply sliding window power normalization to a copy of the envelopes
            fRefEnvNorm = (XFLOAT*)matMalloc(mRef->NrOfFrames()*sizeof(XFLOAT));
```

```
            fDegEnvNorm = (XFLOAT*)matMalloc(mDeg->NrOfFrames()*sizeof(XFLOAT));

            if (fRefEnvNorm == NULL || fDegEnvNorm == NULL)

                OPTTHROW( string("matMalloc failed."));

            XFLOAT meanLevdB;
            rmvesq(mRef->Env(), &meanLevdB, mRef->NrOfFrames());
            meanLevdB = 20.0f * log10(meanLevdB+1e-16f) / mActSpeechThr;

            SlidingWinPowNorm(mRef->Env(), fRefEnvNorm, mRef->NrOfFrames(),
MIN_LEVEL_DB,
                round(WINLEN_ENV / (fTA_framelen*TA_OVERLAPFAC)), -mActSpeechThr,
meanLevdB, false, 0, fSWPNtmp);
            SlidingWinPowNorm(mDeg->Env(), fDegEnvNorm, mDeg->NrOfFrames(),
MIN_LEVEL_DB,
                round(WINLEN_ENV / (fTA_framelen*TA_OVERLAPFAC)), -mActSpeechThr,
meanLevdB, false, 0, fSWPNtmp);

            //Compute cross-correlation
            FindBestShift(fRefEnvNorm+refOffset, mRef->NrOfFrames() - refOffset,
                          fDegEnvNorm, mDeg->NrOfFrames(),
                          iNrOfShifts, refOffset + round(mCrudeDelay / curEnvStepSize),
                          fMaxCorr, iDelay, false, false, fFBStmp1, fFBStmp2);

            if (fRefEnvNorm != NULL) matFree(fRefEnvNorm);
            if (fDegEnvNorm != NULL) matFree(fDegEnvNorm);

            fRefEnvNorm = fDegEnvNorm = NULL;

            if (fMaxCorr >= FCORRTHRESH)
                mCrudeDelay += round(iDelay * curEnvStepSize);
            else
                break;

            iNrOfShifts = ISTEPSIZE+2;
        }

        if (fFBStmp1 != NULL) matFree(fFBStmp1);
        if (fFBStmp2 != NULL) matFree(fFBStmp2);
        if (fSWPNtmp != NULL) matFree(fSWPNtmp);

        fFBStmp1 = fFBStmp2 = fSWPNtmp = NULL;
    }
    OPTCATCH((string errorMsg))
    {
        if (fRefEnvNorm != NULL) matFree(fRefEnvNorm);
        if (fDegEnvNorm != NULL) matFree(fDegEnvNorm);
        if (fFBStmp1    != NULL) matFree(fFBStmp1);
        if (fFBStmp2    != NULL) matFree(fFBStmp2);
        if (fSWPNtmp    != NULL) matFree(fSWPNtmp);

        fRefEnvNorm = fDegEnvNorm = fFBStmp1 = fFBStmp2 = fSWPNtmp = NULL;
        mCrudeDelay = 0;
        OPTTHROW( string("ERROR in SQTimeAlignment::CalcCrudeDelay: " + errorMsg +
"\n"));
    }

    //Store last resolution for RefineCrudeDelay()
    fTA_framelen *= ISTEPSIZE;
    mCurResolution = 2 * round(fTA_framelen * TA_OVERLAPFAC * TA_SAMPLING_RATE);
}

void SQTimeAlignment::RefineCrudeDelay()
{
    XFLOAT *fRefNorm = NULL, *fDegNorm = NULL,
           *fFBStmp1 = NULL, *fFBStmp2 = NULL,
           *fSWPNtmp = NULL;

    OPTTRY
    {
        if (mCurResolution < 3)
            return;
        if (mRef == NULL || mDeg == NULL || mRef->Data() == NULL || mDeg->Data() ==
NULL ||
            mRef->NrOfSamples() < 2 || mRef->NrOfSamples() <= mRef->Start() ||
```

```
            mDeg->NrOfSamples() < 2)
            OPTTHROW( string("Invalid input parameters."));

        if ((mCurResolution | 1) != mCurResolution)
            mCurResolution++;

        //Apply overall delay computed so far
        int  iDegOffset = (((0) > (mRef->Start() + mCrudeDelay)) ? (0) : (mRef->Start()
+ mCrudeDelay));
        int  iRefOffset = mRef->Start() + mCrudeDelay < 0 ?
            -mCrudeDelay : mRef->Start();
        long lRefNorm = mRef->NrOfSamples() - iRefOffset;
        long lDegNorm = mDeg->NrOfSamples() - iDegOffset;
        lRefNorm = lDegNorm = (((lRefNorm) < (lDegNorm)) ? (lRefNorm) : (lDegNorm));
        if (lDegNorm <= 10 || lRefNorm <= 10)
            OPTTHROW( string("Length of aligned Ref and/or Deg too short."));

        //Normalize a copy of both signals
        if ((fRefNorm = (XFLOAT*)matMalloc(lRefNorm*sizeof(XFLOAT))) == NULL ||

            (fDegNorm = (XFLOAT*)matMalloc(lDegNorm*sizeof(XFLOAT))) == NULL ||

            (fFBStmp1 = (XFLOAT*)matMalloc(mCurResolution*sizeof(XFLOAT))) == NULL ||
            (fFBStmp2 = (XFLOAT*)matMalloc(mCurResolution*sizeof(XFLOAT))) == NULL ||
            (fSWPNtmp = (XFLOAT*)matMalloc((((lRefNorm) > (lDegNorm)) ? (lRefNorm) :
(lDegNorm)) * sizeof(XFLOAT))) == NULL)
            OPTTHROW((string("ippsMalloc failed.")));

        SlidingWinPowNorm(mRef->Data()+iRefOffset, fRefNorm, lRefNorm, mActSpeechThr,
            TA_WINLEN_SIGNAL, pow(2.0f, mRef->BitResolution()-1), REF_ASL_LEVEL, true,
0, fSWPNtmp);
        SlidingWinPowNorm(mDeg->Data()+iDegOffset, fDegNorm, lDegNorm, mActSpeechThr,
            TA_WINLEN_SIGNAL, pow(2.0f, mDeg->BitResolution()-1), REF_ASL_LEVEL, true,
0, fSWPNtmp);

        //Find the shifting with the maximum correlation
        XFLOAT fMaxCorr = (XFLOAT)0.0;
        int iBestShift = 0;
        FindBestShift(fRefNorm, lRefNorm,
                      fDegNorm, lDegNorm,
                      mCurResolution, 0,
                      fMaxCorr, iBestShift,
                      true, false, fFBStmp1, fFBStmp2);

        if (fMaxCorr > (XFLOAT)0.7)
        {
            mCrudeDelay += iBestShift;
            mCurResolution = 1;
        }
    }
    OPTCATCH((string errorMsg))
    {
        if (fRefNorm != NULL) matFree(fRefNorm);
        if (fDegNorm != NULL) matFree(fDegNorm);
        if (fFBStmp1 != NULL) matFree(fFBStmp1);
        if (fFBStmp2 != NULL) matFree(fFBStmp2);
        if (fSWPNtmp != NULL) matFree(fSWPNtmp);

        fRefNorm = fDegNorm = fFBStmp1 = fFBStmp2 = fSWPNtmp = NULL;
        OPTTHROW( string("Error in SQTimeAlignment::RefineCrudeDelay: " + errorMsg +
"\n"));
    }
    if (fRefNorm != NULL) matFree(fRefNorm);
    if (fDegNorm != NULL) matFree(fDegNorm);
    if (fFBStmp1 != NULL) matFree(fFBStmp1);
    if (fFBStmp2 != NULL) matFree(fFBStmp2);
    if (fSWPNtmp != NULL) matFree(fSWPNtmp);

    fRefNorm = fDegNorm = fFBStmp1 = fFBStmp2 = fSWPNtmp = NULL;
    return;
}

void SQTimeAlignment::FineDelay()
{
    XFLOAT *fRefNorm = NULL, *fDegNorm = NULL, *fRefEnv = NULL, *fFBStmp1 = NULL,
*fFBStmp2 = NULL, *fSWPNtmp = NULL;
```

```
    OPTTRY
    {
        if (mRef == NULL || mDeg == NULL || mRef->Data() == NULL || mDeg->Data() ==
NULL ||
            mRef->NrOfSamples() < 2 || mDeg->NrOfSamples() < 2 || mActSpeechThr > 0)
            OPTTHROW( string("Invalid input parameters."));

        //Local constants definitions
        int   const FD_MAX_SEGLEN      =
            (((TA_MIN_SEGLEN+1) > (round(1.5 * TA_SAMPLING_RATE))) ? (TA_MIN_SEGLEN+1)
: (round(1.5 * TA_SAMPLING_RATE)));
        int   const FD_MIN_SPANLEN     = 425;
        XFLOAT const FD_MIN_SPANDEC     = (XFLOAT)0.15;
        XFLOAT const FD_FRAMELEN        = (XFLOAT)0.016;
        XFLOAT const FD_OVERLAPFAC      = (XFLOAT)0.75;
        int   const FD_FRAMESTEP       = round(TA_SAMPLING_RATE * FD_FRAMELEN *
FD_OVERLAPFAC);
        int   const FD_MAXSHIFT        =
        XFLOAT const FD_CORRTHRESH_SPAN = (XFLOAT)0.2;
        XFLOAT const FD_CORRTHRESH_LOUD = (XFLOAT)0.32;
        XFLOAT const FD_CORRTHRESH_QUIET= (XFLOAT)0.5;
        XFLOAT const FD_MIN_LEVEL       =
            (XFLOAT)(((mActSpeechThr) > (-55.0)) ? (mActSpeechThr) : (-55.0));

        if(mpLogFile)
            fprintf(mpLogFile, "\nFineDelay()\n");

        //Normalize a copy of both signals.
        XFLOAT fRefTargetMeanLevel = mRef->SpeechActivity() > (XFLOAT)0.2 ?
            REF_ASL_LEVEL + powTodB(mRef->SpeechActivity()) : REF_ASL_LEVEL;
        XFLOAT fDegTargetMeanLevel = mDeg->SpeechActivity() > (XFLOAT)0.2 ?
            REF_ASL_LEVEL + powTodB(mDeg->SpeechActivity()) : REF_ASL_LEVEL;
        int lRefNorm = mRef->NrOfSamples();
        int lDegNorm = mDeg->NrOfSamples();
        if ((fRefNorm = (XFLOAT*)matMalloc(lRefNorm*sizeof(XFLOAT)))    == NULL ||
            (fDegNorm = (XFLOAT*)matMalloc(lDegNorm*sizeof(XFLOAT)))    == NULL ||
            (fFBStmp1 = (XFLOAT*)matMalloc(FD_MAXSHIFT*sizeof(XFLOAT))) == NULL ||
            (fFBStmp2 = (XFLOAT*)matMalloc(FD_MAXSHIFT*sizeof(XFLOAT))) == NULL ||
            (fSWPNtmp = (XFLOAT*)matMalloc((((lRefNorm) > (lDegNorm)) ? (lRefNorm) :
(lDegNorm)) * sizeof(XFLOAT))) == NULL)
            OPTTHROW((string("matMalloc failed.")));

        SlidingWinPowNorm(mRef->Data(), fRefNorm, lRefNorm, mActSpeechThr,
TA_WINLEN_SIGNAL,
            pow((XFLOAT)2.0, mRef->BitResolution()-1), fRefTargetMeanLevel, false,
round(70e-3*TA_SAMPLING_RATE), fSWPNtmp);
        SlidingWinPowNorm(mDeg->Data(), fDegNorm, lDegNorm, mActSpeechThr,
TA_WINLEN_SIGNAL,
            pow((XFLOAT)2.0, mDeg->BitResolution()-1), fDegTargetMeanLevel, false,
round(70e-3*TA_SAMPLING_RATE), fSWPNtmp);

        mRef->CalcEnvelope(FD_FRAMELEN, FD_OVERLAPFAC, MIN_LEVEL_DB, mMaxSigLenBuff);
        int lRefFrames = mRef->NrOfFrames();
        if (lRefFrames < 3)
            OPTTHROW( string("Reference envelope too short."));

        //Create temp. envelope to find utterances in ref signal
        //Utterances will be removed from this env. as they are matched.
        fRefEnv = (XFLOAT*)matMalloc(lRefFrames * sizeof(XFLOAT));
        vmov(mRef->Env(), fRefEnv, lRefFrames);
        XFLOAT fUttThresh =

        //Loop through all utterances in the reference, starting with the loudest one.
        //Try to match each utt. in the ref. with one in the deg. signal.
        delete mSegments;
        mSegments = new TA_SegList;
        ReserveVectorMemory(FD_MIN_SPANLEN);
        mCurSegmentRate = mRef->SamplingFreq();
        bool  doMatchLoudestUtt = true,
              doIgnorePauses    = true,
              silenceFoundRef   = false,
              silenceFoundDeg   = false;
        int   uttStart, uttEnd;
        XFLOAT fMaxCorr  = (XFLOAT)0.0,
              fSpanCorr = (XFLOAT)0.0,
```

```
            fCorrThr  = FD_CORRTHRESH_LOUD;
        int   iBestShift, iGuessedShift,
              iDegStart, iMaxDegLen,
              uttLen, spanLen, spanStart,
              gapSearchPos = 0;

        XFLOAT guessFac;

        int segCounter = 0;
        while (true)
        {
            //Pick the next reference segment to match, using the ref envelope
            if (GetNextRefSegmentToMatch(uttStart, uttEnd, fUttThresh, FD_MIN_LEVEL,
                                         doMatchLoudestUtt, doIgnorePauses,
gapSearchPos,
                                         fRefEnv, lRefFrames, FD_FRAMESTEP, lRefNorm)
                != 0)
                break; //No segment left to match, exit loop.

            if (fUttThresh <= FD_MIN_LEVEL)
                fCorrThr = FD_CORRTHRESH_QUIET;

            //Check both the selected ref segment and the available deg signal
            CheckCurrentRefSeg(fRefNorm, lRefNorm, uttStart, uttEnd,
                               silenceFoundRef, FD_MAX_SEGLEN);

            if (silenceFoundRef && doIgnorePauses && !doMatchLoudestUtt)
            {
                gapSearchPos = uttEnd+1;
                continue;
            }

            CheckCurrentDegSeg(uttStart, fDegNorm, lDegNorm, !silenceFoundRef,
                               silenceFoundDeg, iDegStart, iMaxDegLen);

            if (iMaxDegLen <= 0)
            {
                uttLen = uttEnd - uttStart + 1;
                GuessBestShift(uttStart, iDegStart, iMaxDegLen, fDegNorm, lDegNorm,
                               uttLen, iGuessedShift);
                InsertSegment(uttStart, iGuessedShift, uttLen,
                              TA_SEG_MISSING, lDegNorm,
                              doMatchLoudestUtt, fRefEnv, lRefFrames,
                              FD_FRAMESTEP, MIN_LEVEL_DB);
                continue;
            }

            //Constrain segment length based on available deg signal
            uttLen    = (((uttEnd - uttStart + 1) < (iMaxDegLen)) ? (uttEnd - uttStart
+ 1) : (iMaxDegLen));
            spanLen   = uttLen;
            spanStart = uttStart;
            fMaxCorr  = fSpanCorr = 0.0f;

            //Perform matching by cross-correlation
            if (!silenceFoundRef &&
                !silenceFoundDeg &&
                uttLen >= TA_MIN_SEGLEN)
            {
                GuessBestShift(uttStart, iDegStart, iMaxDegLen, fDegNorm, lDegNorm,

                               uttLen, iGuessedShift, &guessFac);

                //Perform cross-correlation, and determine delay of current segment
                FindBestShift(fRefNorm+uttStart,  uttLen,
                              fDegNorm+iDegStart, iMaxDegLen,
                              (((uttLen+iMaxDegLen) < (FD_MAXSHIFT)) ?
(uttLen+iMaxDegLen) : (FD_MAXSHIFT)),
                              uttStart - iDegStart + iGuessedShift,
                              fMaxCorr, iBestShift, true,
                              mSegments->size() >= 1,
                              fFBStmp1, fFBStmp2,

                              guessFac);

                iBestShift += iGuessedShift;
```

```
                if (abs(iBestShift-iGuessedShift) <= 1 * TA_SAMPLING_RATE / 8000 &&
                    mSegments->size() >= 2)
                    iBestShift = iGuessedShift;

                if (fMaxCorr >= 0.2f ||
                    (fMaxCorr >= 0.1f && uttLen >= 1024) ||
                    (mSegments->size() >= 2 &&
                     fMaxCorr >  0.0f && uttLen >= 2667))

                    fSpanCorr = SpanCorr(fRefNorm, fDegNorm, lDegNorm, iBestShift,
                                         FD_CORRTHRESH_SPAN, fMaxCorr, spanStart,
                                         spanLen, fFBStmp1, fFBStmp2, FD_MIN_SPANLEN);

            }

            //Write found delay to segments list
            if (silenceFoundRef ||
                silenceFoundDeg ||
                uttLen  < TA_MIN_SEGLEN ||
                spanLen < FD_MIN_SPANLEN ||
                fSpanCorr < fCorrThr ||
                (doMatchLoudestUtt && (XFLOAT)spanLen /
                 (((uttLen) < (TA_SAMPLING_RATE*0.375f)) ? (uttLen) :
(TA_SAMPLING_RATE*0.375f))
                  < FD_MIN_SPANDEC))
            {
                if (!doMatchLoudestUtt)
                {
                    TA_SEG_TYPE segType = silenceFoundRef ? TA_SEG_PAUSE :
TA_SEG_GUESSED;
                    GuessBestShift(uttStart, iDegStart, iMaxDegLen, fDegNorm, lDegNorm,
                                   uttLen, iGuessedShift);
                    InsertSegment(uttStart, iGuessedShift, uttLen,
                                  segType, lDegNorm,
                                  doMatchLoudestUtt, fRefEnv, lRefFrames,
                                  FD_FRAMESTEP, MIN_LEVEL_DB,
(((fMaxCorr/FD_CORRTHRESH_QUIET) < (1.0f)) ?
(fMaxCorr/FD_CORRTHRESH_QUIET) : (1.0f)));
                }
                else
                    RemoveUttFromEnv(fRefEnv, lRefFrames, FD_FRAMESTEP, MIN_LEVEL_DB,
                                     uttStart, uttEnd-uttStart+1);
            }
            else
            {
                WriteMatchedSegment(spanStart, spanLen, iBestShift,
                                    iDegStart, iMaxDegLen, lDegNorm,
                                    doMatchLoudestUtt, fRefEnv, lRefFrames,
                                    FD_FRAMESTEP, MIN_LEVEL_DB);
            }

            segCounter++;
        }

        //Perform post-processings

        MergeConsecutiveSegments();

        mMatchQuality = CalcMatchQuality(mSegments);

        if (mMatchQuality < (XFLOAT)0.75)
        {

        FixExtremeMatches(fRefNorm, fDegNorm, lRefNorm, lDegNorm);

        FixIncorrectMatches();

        }

        FineDelayPostProc(fDegNorm, lDegNorm, lRefNorm);

        matFree(fRefEnv);
        fRefEnv = NULL;

        if (fRefNorm != NULL) matFree(fRefNorm);
```

```
        if (fDegNorm != NULL) matFree(fDegNorm);
        if (fFBStmp1 != NULL) matFree(fFBStmp1);
        if (fFBStmp2 != NULL) matFree(fFBStmp2);
        if (fSWPNtmp != NULL) matFree(fSWPNtmp);

        fRefNorm = fDegNorm = fFBStmp1 = fFBStmp2 = fSWPNtmp = NULL;
    }
    OPTCATCH((string errorMsg))
    {
        matFree(fRefEnv);
        fRefEnv = NULL;

        if (fRefNorm != NULL) matFree(fRefNorm);
        if (fDegNorm != NULL) matFree(fDegNorm);
        if (fFBStmp1 != NULL) matFree(fFBStmp1);
        if (fFBStmp2 != NULL) matFree(fFBStmp2);
        if (fSWPNtmp != NULL) matFree(fSWPNtmp);

        fRefNorm = fDegNorm = fFBStmp1 = fFBStmp2 = fSWPNtmp = NULL;
        OPTTHROW( string("ERROR in FineDelay: " + errorMsg + "\n"));
    }
    OPTCATCH((...))
    {
        matFree(fRefEnv);
        fRefEnv = NULL;

        if (fRefNorm != NULL) matFree(fRefNorm);
        if (fDegNorm != NULL) matFree(fDegNorm);
        if (fFBStmp1 != NULL) matFree(fFBStmp1);
        if (fFBStmp2 != NULL) matFree(fFBStmp2);
        if (fSWPNtmp != NULL) matFree(fSWPNtmp);

        fRefNorm = fDegNorm = fFBStmp1 = fFBStmp2 = fSWPNtmp = NULL;
        OPTTHROW( string("ERROR in FineDelay: Unknown error.\n"));
    }
}

XFLOAT SQTimeAlignment::CalcMatchQuality(TA_SegList const *segList)
{
    OPTTRY
    {
        if (segList == NULL || segList->size() == 0)
            OPTTHROW( string("FineDelay did not execute successfully / invalid segments
list."));

        int const TA_SCALED_SHIFT_TOLERANCE =
            round(TA_SHIFT_TOLERANCE * mCurSegmentRate / (XFLOAT)TA_SAMPLING_RATE);

        int totMatchedLen   = 0,
            totMatchableLen = 0;

        //Go through entire segments list.
        //TA_SEG_MACHED segments count towards the match quality.
        //TA_SEG_GUESSED are also counted towards the match quality if they
        //are adjacent to TA_SEG_MATCHED egments and have a similar delay.
        for (unsigned int i = 0; i < segList->size(); i++)
        {
            if ((*segList)[i].segType == TA_SEG_PAUSE)
                continue;
            if ((*segList)[i].segType == TA_SEG_MISSING)
            {
                totMatchableLen += (*segList)[i].segLen;
                continue;
            }

            TA_SEG_TYPE curSegType = (*segList)[i].segType;
            int curShift       = (*segList)[i].degPos - (*segList)[i].refPos,
                curSegLen      = (*segList)[i].segLen,
                matchedSegsLen = curSegType == TA_SEG_MATCHED ? curSegLen : 0,
                guessedSegsLen = curSegType != TA_SEG_MATCHED ? curSegLen : 0,
                lastConsecSeg;
            XFLOAT matchedVSguessedRatio = (XFLOAT)0.0;

            for (lastConsecSeg = i+1;
                 lastConsecSeg < (int)segList->size() &&
                 (*segList)[lastConsecSeg].segType != TA_SEG_MISSING &&
```

```
                (*segList)[lastConsecSeg].segType != TA_SEG_PAUSE   &&
                abs( (*segList)[lastConsecSeg].degPos -
(*segList)[lastConsecSeg].refPos - curShift ) <=
TA_SCALED_SHIFT_TOLERANCE;
                lastConsecSeg++)
            {
                curSegLen  = (*segList)[lastConsecSeg].segLen;
                curSegType = (*segList)[lastConsecSeg].segType;

                if (curSegType == TA_SEG_MATCHED)
                    matchedSegsLen += curSegLen;
                else
                    guessedSegsLen += curSegLen;
            }

            i = lastConsecSeg - 1;
            totMatchableLen += matchedSegsLen + guessedSegsLen;

            if (matchedSegsLen > 0)
            {
                matchedVSguessedRatio = matchedSegsLen / (XFLOAT)(matchedSegsLen +
guessedSegsLen);
                totMatchedLen += (int)(matchedSegsLen + matchedVSguessedRatio *
guessedSegsLen);
            }
        }

        if (totMatchableLen <= 0.0f)
            OPTTHROW( string("Total non-pause signal length in segment list = 0 ?!"));

        return (totMatchedLen / (XFLOAT)totMatchableLen);
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in CalcMatchQuality: " + errorMsg + "\n"));
    }
}

void SQTimeAlignment::CalcDelaySpread()
{
    OPTTRY
    {
        if (mSegments == NULL || mSegments->size() == 0)
            OPTTHROW( string("Empty segments list."));

        int maxDelay, minDelay, i = 0;

        while (i < (int)mSegments->size() && (*mSegments)[i].segType != TA_SEG_MATCHED)
            i++;
        if (i == (int)mSegments->size())
            return;

        maxDelay = minDelay =
            (*mSegments)[i].degPos - (*mSegments)[i].refPos;

        for (; i < (int)mSegments->size(); i++)
        {
            if ((*mSegments)[i].segType != TA_SEG_MATCHED)
                continue;

            maxDelay = ((((*mSegments)[i].degPos - (*mSegments)[i].refPos) >
(maxDelay)) ? ((*mSegments)[i].degPos - (*mSegments)[i].refPos) :
(maxDelay));
            minDelay = ((((*mSegments)[i].degPos - (*mSegments)[i].refPos) <
(minDelay)) ? ((*mSegments)[i].degPos - (*mSegments)[i].refPos) :
(minDelay));
        }

        mMinDelay = minDelay;
        mMaxDelay = maxDelay;
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in CalcDelaySpread: " + errorMsg + "\n"));
    }
}
```

```
void SQTimeAlignment::SlidingWinPowNorm(XFLOAT const *fIn, XFLOAT *fOut,
                                        long len,
                                        XFLOAT const &sigThreshdB,
                                        int WINLEN,
                                        XFLOAT const &signalMaxAmp,
                                        XFLOAT const &meanOutLevel,
                                        bool doAvoidShortBursts,
                                        int const hystLenInSamples,
                                        XFLOAT *tempBuff)
{
    OPTTRY
    {

        const double ROUNDFACTOR = 1e8;

        if ((WINLEN | 0x1) != WINLEN)
            WINLEN--;

        if (fIn == NULL || fOut == NULL || len < WINLEN+3 ||
            fIn == fOut || WINLEN < 2)
            OPTTHROW( string("Invalid input parameters. Note: In-place operation is not
supported."));

        double dWinEnergy  = 0.0;
        double dEnergThr   =
               signalMaxAmp*signalMaxAmp * dBtoPow(sigThreshdB) * WINLEN;
        int    hystCounter = 0;
        XFLOAT fTemp;
        rmvesq(fIn, &fTemp, WINLEN/2);

        dWinEnergy = floor(ROUNDFACTOR*fTemp*fTemp * (int)(WINLEN/2))/ROUNDFACTOR;

        double *fInSquared = tempBuff != NULL ? tempBuff : (double*)matMalloc(len *
sizeof(double));

        matbSqr2(fIn, fInSquared, len);

        //Process first WINLEN/2 data points
        for (int i = 0; i < WINLEN/2+1; i++)
        {
            if (dWinEnergy > dEnergThr || (hystCounter > 0 && dWinEnergy > 0.0))
            {
                fOut[i]     = (XFLOAT)(fIn[i] / sqrt(dWinEnergy / WINLEN));
                hystCounter = dWinEnergy > dEnergThr ?
                    (((hystLenInSamples) < (hystCounter+1)) ? (hystLenInSamples) :
(hystCounter+1)) :
                    (((0) > (hystCounter-1)) ? (0) : (hystCounter-1));
            }
            else
            {
                fOut[i]     = (XFLOAT)0.0;
                hystCounter = 0;
            }

            dWinEnergy += floor(ROUNDFACTOR *
fIn[i+WINLEN/2]*fIn[i+WINLEN/2])/ROUNDFACTOR;

        }

        //Main loop
        for (int i = WINLEN/2+1; i < len - WINLEN/2; i++)
        {
            if (dWinEnergy > dEnergThr || (hystCounter > 0 && dWinEnergy > 0.0))
            {
                fOut[i]     = (XFLOAT)(fIn[i] / sqrt(dWinEnergy / WINLEN));
                hystCounter = dWinEnergy > dEnergThr ?
                    (((hystLenInSamples) < (hystCounter+1)) ? (hystLenInSamples) :
(hystCounter+1)) :
                    (((0) > (hystCounter-1)) ? (0) : (hystCounter-1));
            }
            else
            {
                fOut[i]     = (XFLOAT)0.0;
                hystCounter = 0;
            }
```

```
            dWinEnergy = floor(matSum(&fInSquared[i - WINLEN/2], WINLEN) *
ROUNDFACTOR)/ROUNDFACTOR;

        }

        //Process last WINLEN/2 data points
        for (int i = len - WINLEN/2; i < len; i++)
        {
            if (dWinEnergy > dEnergThr || (hystCounter > 0 && dWinEnergy > 0.0))
            {
                fOut[i]     = (XFLOAT)(fIn[i] / sqrt(dWinEnergy / WINLEN));
                hystCounter = dWinEnergy > dEnergThr ?
                    (((hystLenInSamples) < (hystCounter+1)) ? (hystLenInSamples) :
(hystCounter+1)) :
                    (((0) > (hystCounter-1)) ? (0) : (hystCounter-1));
            }
            else
            {
                fOut[i]     = (XFLOAT)0.0;
                hystCounter = 0;
            }

            dWinEnergy -= floor(ROUNDFACTOR *
fIn[i-WINLEN/2-1]*fIn[i-WINLEN/2-1])/ROUNDFACTOR;

        }

        if (doAvoidShortBursts)
        {
            int actStart = 0;
            for (int i = 0; i < len; i++)
            {
                while ((i < len) && ( fOut[i] == (XFLOAT)0.0 )) i++;
                actStart = i;
                while ( (i < len ) && (fOut[i] != (XFLOAT)0.0) ) i++;
                if (i - actStart < 2*WINLEN)
                    for (int j = actStart; j < i; j++)
                        fOut[j] = (XFLOAT)0.0;
            }
        }

        //Rescale processed data to desired level
        XFLOAT fScalefac;
        rmvesq(fOut, &fScalefac, len);
        if (fScalefac > (XFLOAT)0.0)
        {
            fScalefac = pow((XFLOAT)10.0, (XFLOAT)meanOutLevel/(XFLOAT)20.0) /
(fScalefac / signalMaxAmp);
            vsmul(fOut, fScalefac, fOut, len);
        }

        if(fInSquared && fInSquared != tempBuff)
            matFree(fInSquared);
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in SQTimeAlignment::SlidingWinPowNorm:" + errorMsg +
"\n"));
    }
}

void SQTimeAlignment::FindBestShift(XFLOAT const* fRef, int const lRef,
                                    XFLOAT const* fDeg, int const lDeg,
                                    int iNumLags, int const iDegOffset,
                                    XFLOAT &fMaxCorr, int &iMaxPos,
                                    bool  const doFindAbsMax,
                                    bool  const doWeightDegOffset,
                                    XFLOAT *fCorrs, XFLOAT *fWeighted,
                                    XFLOAT const guessReliability,
                                    short const normalizationType,
                                    bool  const checkPeakiness)
{
    OPTTRY
    {
        if (fRef == NULL || fDeg == NULL || fCorrs == NULL || fWeighted == NULL ||
```

```
            lRef <= 0 || lDeg <= 0 || iNumLags  < 2)
            OPTTHROW( string("Invalid input arguments."));
        if ((iNumLags | 0x1) != iNumLags)
            iNumLags++;

        XFLOAT const FBS_MIN_PEAKINESS_FAC = (XFLOAT)20.0;

        int   const FBS_MAXIMA_NUM   = 30;
        XFLOAT const FBS_MAXIMA_RATIO =  (XFLOAT)0.9;
        int   const FBS_NON_PENALIZED_ZONE
            = 256 * TA_SAMPLING_RATE / 8000;
        int   const FBS_RAMP_LEN
            = 2000* TA_SAMPLING_RATE / 8000;

        //Initialize cosine weights
        static bool   isCosineBowInitialized = false;
        static XFLOAT cosineBowWeighting[2*FBS_RAMP_LEN+1];
        if (!isCosineBowInitialized)
        {
            for (int i = -FBS_RAMP_LEN; i <= FBS_RAMP_LEN; i++)
                cosineBowWeighting[i+FBS_RAMP_LEN] =
                    0.5f  *  ( cos(i/(XFLOAT)FBS_RAMP_LEN * (XFLOAT)PI) + 1 );
            isCosineBowInitialized = true;
        }

        int numNonZeroSampRef = 0, numNonZeroSampDeg = 0;
        for (int i = 0; i < lRef; i++)
            if (fRef[i] != 0.0f)
                numNonZeroSampRef++;
        for (int i = 0; i < lDeg; i++)
            if (fDeg[i] != 0.0f)
                numNonZeroSampDeg++;

        //Compute the normalizing factor:
        XFLOAT fDivisor;
        int numNonZeroSampBoth;
        XFLOAT fAutocorrRef; svesq(fRef, &fAutocorrRef, lRef);
        XFLOAT fAutocorrDeg; svesq(fDeg, &fAutocorrDeg, lDeg);

        switch (normalizationType)
        {
        case NORM_NONE:
            fDivisor = (XFLOAT)1.0;
            break;
        case NORM_SIMPLE_GEOMETRIC_MEAN:
            fDivisor = sqrt (fAutocorrRef*fAutocorrDeg);
            break;
        case NORM_WEIGHTED_GEOMETRIC_MEAN:
            numNonZeroSampBoth = (((numNonZeroSampRef) < (numNonZeroSampDeg)) ?
(numNonZeroSampRef) : (numNonZeroSampDeg));
            fDivisor = sqrt( (fAutocorrRef/numNonZeroSampRef) *
                             (fAutocorrDeg/numNonZeroSampDeg) )
                             * numNonZeroSampBoth;
            break;
        default:
            OPTTHROW( string ("No valid normalization type for correlation."));
            break;
        }

        //Compute the cross-correlation at iNumLags different shifts
        int lowestLag = -iNumLags/2 + iDegOffset;

        OPTTRY
        {
            matCrossCorr(matHandle, fRef, lRef, fDeg, lDeg, fCorrs, iNumLags,
lowestLag);
        }
        OPTCATCH((...))
        {
            char* dbgs = "matCrossCorr Crash";
            OPTTHROW( string (dbgs));
        }

        //Normalize the correlation vector
        vsdiv(fCorrs, fDivisor, fWeighted, iNumLags);
        if (doFindAbsMax)
```

```
            matbAbs2(fWeighted, fCorrs, iNumLags);
        else
            vmov(fWeighted, fCorrs, iNumLags);

        //Weight correlations vector by distance from center shift
        if (doWeightDegOffset)
        {
            XFLOAT FBS_MIN_WEIGHT = (XFLOAT)0.50;

            if (guessReliability < 0)
                OPTTHROW( string("This should never happen!"));
            else
                FBS_MIN_WEIGHT = limit((XFLOAT)0.8 - guessReliability * (XFLOAT)0.3,
(XFLOAT)0.5, (XFLOAT)0.8);

            matbSet(FBS_MIN_WEIGHT, fWeighted, iNumLags);

            int nonPenZoneStart = (((0) > (iNumLags/2 - FBS_NON_PENALIZED_ZONE/2)) ?
(0) : (iNumLags/2 - FBS_NON_PENALIZED_ZONE/2)),
                nonPenZoneEnd   = (((iNumLags) < (iNumLags/2 +
FBS_NON_PENALIZED_ZONE/2)) ? (iNumLags) : (iNumLags/2 +
FBS_NON_PENALIZED_ZONE/2));
            matbSet(1.0f, fWeighted+nonPenZoneStart, nonPenZoneEnd-nonPenZoneStart+1);

            int rampStart = (((0) > (nonPenZoneStart - (((iNumLags/2) <
((int)FBS_RAMP_LEN)) ? (iNumLags/2) : ((int)FBS_RAMP_LEN)))) ? (0) :
(nonPenZoneStart - (((iNumLags/2) < ((int)FBS_RAMP_LEN)) ? (iNumLags/2) :
((int)FBS_RAMP_LEN)))),
                rampEnd   = (((iNumLags) < (nonPenZoneEnd + (((iNumLags/2) <
((int)FBS_RAMP_LEN)) ? (iNumLags/2) : ((int)FBS_RAMP_LEN)))) ?
(iNumLags) : (nonPenZoneEnd + (((iNumLags/2) < ((int)FBS_RAMP_LEN)) ?
(iNumLags/2) : ((int)FBS_RAMP_LEN))));

            if (mMaxSigLenBuff == NULL || iNumLags > (((lRef) > (lDeg)) ? (lRef) :
(lDeg)))
            {
                matAddProduct1(cosineBowWeighting + FBS_RAMP_LEN -
(nonPenZoneStart-rampStart), 1-FBS_MIN_WEIGHT,
                        fWeighted + rampStart,       nonPenZoneStart-1 - rampStart +
1);
                matAddProduct1(cosineBowWeighting + FBS_RAMP_LEN + 1,
         1-FBS_MIN_WEIGHT,
                        fWeighted + nonPenZoneEnd+1, rampEnd-1 - (nonPenZoneEnd+1) +
1);
                vmul    (fWeighted, fCorrs, fWeighted, iNumLags);
            }
            else
            {
                vsmul(cosineBowWeighting + FBS_RAMP_LEN - (nonPenZoneStart-rampStart),
1-FBS_MIN_WEIGHT,
                      mMaxSigLenBuff + rampStart,       nonPenZoneStart-1 - rampStart +
1);
                vsmul(cosineBowWeighting + FBS_RAMP_LEN + 1,
1-FBS_MIN_WEIGHT,
                      mMaxSigLenBuff + nonPenZoneEnd+1, rampEnd-1 - (nonPenZoneEnd+1) +
1);
                vadd (fWeighted + rampStart,       mMaxSigLenBuff + rampStart,
                      fWeighted + rampStart,       nonPenZoneStart-1 - rampStart + 1);
                vadd (fWeighted + nonPenZoneEnd+1, mMaxSigLenBuff + nonPenZoneEnd+1,
                      fWeighted + nonPenZoneEnd+1, rampEnd-1 - (nonPenZoneEnd+1) + 1);
                vmul(fWeighted,      fCorrs, mMaxSigLenBuff, iNumLags);
                vmov(mMaxSigLenBuff,         fWeighted,      iNumLags);
            }
        }

        //Find max cross-corr in the (possibly weighted) vector
        int iMaxIndex = -1, tempIndex = -1, finalIndex, searchIdx;
        XFLOAT tempMaxCorr = 0.0f, finalCorr;
        if (!doWeightDegOffset)
            fMaxCorr = matMaxExt(fCorrs, iNumLags, &iMaxIndex);
        else
        {
            fMaxCorr = matMaxExt(fWeighted, iNumLags, &iMaxIndex);
            fWeighted[iMaxIndex] = 0.0f;
            finalIndex = iMaxIndex;
            finalCorr  = fMaxCorr;
```

```
            for (int i = 0; fMaxCorr > 0.0f && i < FBS_MAXIMA_NUM-1; i++)
            {
                searchIdx = (((iNumLags-1-finalIndex) < (finalIndex)) ?
(iNumLags-1-finalIndex) : (finalIndex));
                tempMaxCorr = matMaxExt(fWeighted+searchIdx, 2*(iNumLags/2-searchIdx),
&tempIndex);
                tempIndex += searchIdx;

                if (tempMaxCorr < FBS_MAXIMA_RATIO * fMaxCorr || searchIdx ==
iNumLags/2)
                    break;

                fWeighted[tempIndex] = 0.0f;
                finalIndex = tempIndex;
                finalCorr  = tempMaxCorr;
            }
            iMaxIndex = finalIndex;
            fMaxCorr  = finalCorr;
        }

        //Clean up and leave directly if there is no cross-correlation whatsoever.
        if (fMaxCorr <= 0.0f)
        {
            iMaxPos = 0;
            return;
        }

        //Determine the peakiness of the *original* (non-weighted) cross-corr vector.
        if (checkPeakiness)
        {
            int nonZeroStart = 0, nonZeroEnd = iNumLags-1;
            while (fCorrs[nonZeroStart] < (XFLOAT)0.01 && nonZeroStart < iNumLags-1)
                nonZeroStart++;
            while (fCorrs[nonZeroEnd]   < (XFLOAT)0.01 && nonZeroEnd > 0)
                nonZeroEnd--;
            if (nonZeroEnd-nonZeroStart+1 >= 21)
            {
                XFLOAT fCorrAvg;
                fCorrAvg = matMean(fCorrs+nonZeroStart, nonZeroEnd-nonZeroStart+1);
                if (fCorrAvg > (XFLOAT)0.0 && fMaxCorr / fCorrAvg >
FBS_MIN_PEAKINESS_FAC)
                    fMaxCorr = TA_INFINITE_CORR;
            }
        }

        iMaxPos = iMaxIndex - iNumLags/2;
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in SQTimeAlignment::FindBestShift: " + errorMsg +
"\n"));
    }
    OPTCATCH((...))
    {
        OPTTHROW( string("ERROR in SQTimeAlignment::FindBestShift: Unknown error.\n"));
    }
}

void SQTimeAlignment::ReserveVectorMemory(int minSegLen)
{
    if (mSegments == NULL || mRef == NULL || mDeg == NULL || minSegLen <= 0)
        OPTTHROW( string("ERROR in ReserveVectorMemory: Invalid input data.\n"));

    int const RVM_MAX_NUMSEGS = 100;
    int const RVM_MIN_NUMSEGS =  10;

    //Try to guess the number of segments ultimately used
    int approxNumSegs = ((((int)ceil(mRef->NrOfSamples() * mRef->SpeechActivity() /
minSegLen)) > ((int)ceil(mDeg->NrOfSamples() * mDeg->SpeechActivity() /
minSegLen))) ? ((int)ceil(mRef->NrOfSamples() * mRef->SpeechActivity() /
minSegLen)) : ((int)ceil(mDeg->NrOfSamples() * mDeg->SpeechActivity() /
minSegLen)));

    mSegments->reserve(limit(approxNumSegs, RVM_MIN_NUMSEGS, RVM_MAX_NUMSEGS));
}
```

```
int SQTimeAlignment::GetNextRefSegmentToMatch (int &uttStart, int &uttEnd,
                                               XFLOAT &uttThresh,
                                               XFLOAT const &FD_MIN_LEVEL,
                                               bool &doMatchLoudestUtt,
                                               bool &doIgnorePauses,
                                               int &gapSearchPos,
                                               XFLOAT const *env, int const &lEnv,
                                               int const &frameStep, int const
&lSignal)
{
    OPTTRY
    {
        if (env == NULL || lEnv <= 0 || frameStep < 1 || lSignal < frameStep)
            OPTTHROW( string("Invalid input arguments."));

        while (true)
        {
            if (doMatchLoudestUtt) //Find loudest continuous speech above set threshold
in Ref
            {
                bool uttFound;
                FindLoudestUtt(env, lEnv, lSignal, frameStep,
                               uttThresh, uttFound,
                               uttStart, uttEnd);
                if (!uttFound && uttThresh > FD_MIN_LEVEL)
                {
                    uttThresh  = FD_MIN_LEVEL;
                    continue;
                }
                else if (!uttFound)
                    doMatchLoudestUtt = false;
                else
                    return 0;
            }

            //Once no speech to match is left in the ref envelope,
            //find gaps in the segments list and match those.
            if (!doMatchLoudestUtt)
            {
                if(mSegments->size() == 0)
                {

                    InsertSegment((((0) > (-mCrudeDelay)) ? (0) : (-mCrudeDelay)),
                                  mCrudeDelay,
                                  (((mRef->NrOfSamples() - (((0) > (-mCrudeDelay)) ?
(0) : (-mCrudeDelay))) < (mDeg->NrOfSamples() - (((0)
> (mCrudeDelay)) ? (0) : (mCrudeDelay)))) ?
(mRef->NrOfSamples() - (((0) > (-mCrudeDelay)) ? (0)
: (-mCrudeDelay))) : (mDeg->NrOfSamples() - (((0) >
(mCrudeDelay)) ? (0) : (mCrudeDelay)))),
                                  TA_SEG_GUESSED,
                                  mDeg->NrOfSamples(), false,
                                  (XFLOAT*)env, lEnv, frameStep, MIN_LEVEL_DB);
                }
                int gapStart, gapEnd;
                if (mSegments->findNextGapInRefSegList(gapStart, gapEnd,
                                                       gapSearchPos, lSignal)
                    < 0)
                {
                    if (!doIgnorePauses)
                        return -1;
                    else
                    {
                        doIgnorePauses = false;
                        gapSearchPos   = 0;
                        continue;
                    }
                }
                else
                {
                    if (gapStart > gapEnd)
                        OPTTHROW( string("findNextGapInRefSegList() returned invalid
gap,\n\
                            probably due to a corruped segments list."));
```

```
                    uttStart = gapStart;
                    uttEnd   = gapEnd;
                    return 0;
                }
            }
        }
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in GetNextRefSegmentToMatch: " + errorMsg + "\n"));
    }
}

void SQTimeAlignment::FindLoudestUtt (XFLOAT const *fEnv, int const &envLen,
                                      int const &sigLen, int const &frameStep,
                                      XFLOAT const &uttThresh,
                                      bool &foundUtt,
                                      int &uttStartSamples, int &uttEndSamples)
{
    OPTTRY
    {
        if (fEnv == NULL || envLen <= 3 || frameStep < 1 || sigLen < frameStep)
            OPTTHROW( string("Invalid input parameters."));

        XFLOAT uttSum     = (XFLOAT)0.0,
               maxUtt     = (MIN_LEVEL_DB - 10.0f) * envLen;
        int    uttLen     = 0,
               uttStartFrames = 0, uttEndFrames = 0;
        int const FLU_MIN_UTTLEN = TA_MIN_SEGLEN / frameStep;

        foundUtt = false;
        for (int i = 0; i < envLen; i++)
        {
            if (fEnv[i] >= uttThresh)
            {
                uttSum  += fEnv[i] - MIN_LEVEL_DB;
                uttLen++;
                if (i == envLen-1 &&
                    ((uttSum > maxUtt &&
                      (uttEndFrames - uttStartFrames + 1 < FLU_MIN_UTTLEN ||
                       uttLen - 1 >= FLU_MIN_UTTLEN))
                     ||
                     (uttEndFrames - uttStartFrames + 1 < FLU_MIN_UTTLEN &&
                      uttLen > uttEndFrames - uttStartFrames + 1)))
                {
                    maxUtt = uttSum;
                    uttStartFrames = i-uttLen;
                    uttEndFrames   = i;
                    foundUtt = true;
                }
            }
            else if (uttLen > 0)
            {
                if ((uttSum > maxUtt &&
                     (uttEndFrames - uttStartFrames + 1 < FLU_MIN_UTTLEN ||
                      uttLen - 1 >= FLU_MIN_UTTLEN))
                    ||
                    (uttEndFrames - uttStartFrames + 1 < FLU_MIN_UTTLEN &&
                     uttLen > uttEndFrames - uttStartFrames + 1))
                {
                    maxUtt = uttSum;
                    uttStartFrames = i-uttLen;
                    uttEndFrames   = i-1;
                    foundUtt = true;
                }
                uttSum = (XFLOAT)0.0;
                uttLen = 0;
            }
        }

        if (foundUtt)
        {
            uttStartSamples = (int)(uttStartFrames * frameStep);
            uttStartSamples = ((((((0) > (uttStartSamples)) ? (0) : (uttStartSamples)))
< (sigLen)) ? ((((0) > (uttStartSamples)) ? (0) : (uttStartSamples))) :
(sigLen));
```

```
            uttEndSamples   = (int)((uttEndFrames+1) * frameStep);
            uttEndSamples   = ((((((0) > (uttEndSamples)) ? (0) : (uttEndSamples))) <
(sigLen)) ? ((((0) > (uttEndSamples)) ? (0) : (uttEndSamples))) :
(sigLen));
        }
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in FindLoudestUtt: " + errorMsg + "\n"));
    }
}

void SQTimeAlignment::GuessBestShift (int const iRefPos,
                                      int const degStart,
                                      int const degLen,
                                      XFLOAT const *degSig,
                                      int const totDegLen,
                                      int const uttLen,
                                      int &iGuessedShift,
                                      XFLOAT *guessReliability,
                                      int const endOf1stHalf)
{
    OPTTRY
    {
        if (mSegments == NULL || iRefPos < 0 || degStart < 0 || degSig == NULL ||
            (guessReliability != NULL && degLen <= 0))
            OPTTHROW( string("Invalid input arguments."));

        if (guessReliability != NULL)
            *guessReliability = -1.0f;

        TA_SegList::iterator nextSeg = mSegments->findInsLoc(iRefPos);
        TA_SegList::iterator prevSeg = nextSeg == mSegments->begin() ?
            mSegments->end() : nextSeg - 1;

        if (nextSeg != mSegments->end())
        {
            while (nextSeg != mSegments->end() &&
                   (nextSeg->segType == TA_SEG_MISSING || nextSeg->segType ==
TA_SEG_PAUSE))
                nextSeg++;

            if (endOf1stHalf > 0 && nextSeg != mSegments->end() &&
                iRefPos < endOf1stHalf && nextSeg->refPos > endOf1stHalf)
                nextSeg = mSegments->end();
        }
        if (prevSeg != mSegments->end())
        {
            while (prevSeg != mSegments->begin() &&
                   (prevSeg->segType == TA_SEG_MISSING || prevSeg->segType ==
TA_SEG_PAUSE))
                prevSeg--;
            if (prevSeg->segType == TA_SEG_MISSING || prevSeg->segType == TA_SEG_PAUSE)
                prevSeg = mSegments->end();

            if (endOf1stHalf > 0 && prevSeg != mSegments->end() &&
                iRefPos > endOf1stHalf && prevSeg->refPos < endOf1stHalf)
                prevSeg = mSegments->end();
        }

        if (nextSeg == mSegments->end() && prevSeg == mSegments->end())
        {
            iGuessedShift = mCrudeDelay;

            iGuessedShift = (((0 - iRefPos) > (iGuessedShift)) ? (0 - iRefPos) :
(iGuessedShift));

            if (degLen > 0)
            {
                iGuessedShift = (((degStart - iRefPos) > (iGuessedShift)) ? (degStart -
iRefPos) : (iGuessedShift));
                iGuessedShift = ((((degStart+degLen-1) - (iRefPos+uttLen-1)) <
(iGuessedShift)) ? ((degStart+degLen-1) - (iRefPos+uttLen-1)) :
(iGuessedShift));
            }
            return;
```

```
        }

        if (prevSeg == mSegments->end() ||
            (nextSeg != mSegments->end() &&
             iRefPos+uttLen == nextSeg->refPos &&
             prevSeg->refPos+prevSeg->segLen < iRefPos))
        {
            iGuessedShift = nextSeg->degPos - nextSeg->refPos;
            if (guessReliability != NULL)
                *guessReliability = RateGuessReliability(nextSeg->refPos - (iRefPos +
uttLen - 1),
                                                         uttLen, degStart, degLen,
degSig, totDegLen);
        }

        else if (nextSeg == mSegments->end() ||
                 (prevSeg != mSegments->end() &&
                  iRefPos == prevSeg->refPos+prevSeg->segLen &&
                  nextSeg->refPos > iRefPos+uttLen))
        {
            iGuessedShift = prevSeg->degPos - prevSeg->refPos;
            if (guessReliability != NULL)
                *guessReliability = RateGuessReliability(iRefPos - (prevSeg->refPos +
prevSeg->segLen - 1),
                                                         uttLen, degStart, degLen,
degSig, totDegLen);
        }

        else if (iRefPos+uttLen == nextSeg->refPos &&
                 iRefPos == prevSeg->refPos+prevSeg->segLen)
        {
            if (prevSeg->segType == nextSeg->segType)
                iGuessedShift = prevSeg->segLen >= nextSeg->segLen ?
                    prevSeg->degPos - prevSeg->refPos :
                    nextSeg->degPos - nextSeg->refPos;
            else
                iGuessedShift = prevSeg->segType == TA_SEG_MATCHED ?
                    prevSeg->degPos - prevSeg->refPos :
                    nextSeg->degPos - nextSeg->refPos;

            if (guessReliability != NULL)
                *guessReliability = 1;
        }

        else
        {
            int gapLen        = nextSeg->refPos - (prevSeg->refPos + prevSeg->segLen) +
1;
            int distToPrevSeg = iRefPos - (prevSeg->refPos + prevSeg->segLen - 1);
            int distToNextSeg = nextSeg->refPos - (iRefPos + uttLen - 1);
            int prevSegShift  = prevSeg->degPos - prevSeg->refPos;
            int nextSegShift  = nextSeg->degPos - nextSeg->refPos;
            if (gapLen <= 0 || gapLen-uttLen <= 0)
            {
                iGuessedShift = prevSegShift;
                if (guessReliability != NULL)
                    *guessReliability = 0.0f;
            }
            else
            {
                iGuessedShift = (prevSegShift * (gapLen-uttLen-distToPrevSeg) +
                                 nextSegShift * (gapLen-uttLen-distToNextSeg)) /
(gapLen-uttLen);
                if (guessReliability != NULL)
                    *guessReliability = RateGuessReliability((((distToPrevSeg) <
(distToNextSeg)) ? (distToPrevSeg) : (distToNextSeg)),
                                                             abs(prevSegShift -
nextSegShift));
            }
        }

        int uncorrectedGuess = iGuessedShift;

        iGuessedShift = (((0 - iRefPos) > (iGuessedShift)) ? (0 - iRefPos) :
(iGuessedShift));
```

```
        if (degLen > 0)
        {
            iGuessedShift = (((degStart - iRefPos) > (iGuessedShift)) ? (degStart -
iRefPos) : (iGuessedShift));
            iGuessedShift = ((((degStart+degLen-1) - (iRefPos+uttLen-1)) <
(iGuessedShift)) ? ((degStart+degLen-1) - (iRefPos+uttLen-1)) :
(iGuessedShift));
        }
        else
        {
            iGuessedShift = degStart - (iRefPos+uttLen/2);
            iGuessedShift = (((0 - iRefPos) > (iGuessedShift)) ? (0 - iRefPos) :
(iGuessedShift));
            iGuessedShift = (((totDegLen - (iRefPos+uttLen)) < (iGuessedShift)) ?
(totDegLen - (iRefPos+uttLen)) : (iGuessedShift));
        }

        if (iGuessedShift != uncorrectedGuess && guessReliability != NULL)
            *guessReliability = 0.0f;

        return;
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in GuessBestShift: " + errorMsg + "\n"));
    }
    OPTCATCH((...))
    {
        OPTTHROW( string("ERROR in GuessBestShift: Unknown error.\n"));
    }
}

XFLOAT SQTimeAlignment::RateGuessReliability(int distToClosestSeg,
                                             int shiftDiffBetweenSegs)
{
    if (distToClosestSeg < 0)
        return (XFLOAT)-1.0;

    int const RGR_MAX_NONPENALIZED_DIST      =
        6000 * TA_SAMPLING_RATE / 8000;
    int const RGR_MAX_DIST                   =
        round(1.5f * TA_SAMPLING_RATE);
    int const RGR_MAX_NONPENALIZED_SHIFTDIFF =
        400 * TA_SAMPLING_RATE / 8000;
    int const RGR_MAX_SHIFTDIFF              =
        6000 * TA_SAMPLING_RATE / 8000;

    XFLOAT reliability;

    reliability = limit((XFLOAT)1.0 - (distToClosestSeg - RGR_MAX_NONPENALIZED_DIST) /
                        (XFLOAT)(RGR_MAX_DIST - RGR_MAX_NONPENALIZED_DIST),
                       (XFLOAT)0.0, (XFLOAT)1.0);

    reliability *= limit((XFLOAT)1.0 - (shiftDiffBetweenSegs -
RGR_MAX_NONPENALIZED_SHIFTDIFF) /
                         (XFLOAT)(RGR_MAX_SHIFTDIFF - RGR_MAX_NONPENALIZED_SHIFTDIFF),
                        (XFLOAT)0.0, (XFLOAT)1.0);

    return reliability;
}

XFLOAT SQTimeAlignment::RateGuessReliability(int distToClosestSeg, int uttLen,
                                             int degStart, int availDegLen,
                                             XFLOAT const *degSig, int totDegLen)
{
    if (distToClosestSeg < 0 || uttLen < 1 || degStart < 0 || availDegLen < 1 ||
        degSig == NULL || totDegLen < 1 || totDegLen < degStart+availDegLen)
        OPTTHROW( string("ERROR in RateGuessReliability: Invalid input arguments.\n"));

    XFLOAT const RGR_MAX_NONPENALIZED_CHOICEFAC =
        (XFLOAT)3.0;
    XFLOAT const RGR_MAX_CHOICEFAC              =
        (XFLOAT)8.0;
    int   const RGR_MIN_UTTLEN                  =
        round(0.25f * TA_SAMPLING_RATE);
```

```
    XFLOAT reliability = RateGuessReliability(distToClosestSeg, 0);

    int numNonSilentSamples = availDegLen, cnt;
    for (int i = degStart; i < degStart+availDegLen; i++)
    {
        for (cnt = 0; i < degStart+availDegLen && degSig[i] == 0.0f; i++, cnt++);
        if (cnt > 40)
            numNonSilentSamples -= cnt;
    }
    numNonSilentSamples = (((numNonSilentSamples) > (0)) ? (numNonSilentSamples) :
(0));

    XFLOAT choiceFac = numNonSilentSamples / (XFLOAT)(((uttLen) > (RGR_MIN_UTTLEN)) ?
(uttLen) : (RGR_MIN_UTTLEN));

    reliability *= limit((XFLOAT)1.0 - (choiceFac - RGR_MAX_NONPENALIZED_CHOICEFAC) /
                         (RGR_MAX_CHOICEFAC - RGR_MAX_NONPENALIZED_CHOICEFAC),
                        (XFLOAT)0.0, (XFLOAT)1.0);

    return reliability;
}

void SQTimeAlignment::CheckCurrentDegSeg (int const &uttStart, XFLOAT const *degSignal,
                                          int const &lDegSignal,
                                          bool const &doCheck,
                                          bool &flag,
                                          int &iDegStart, int &iMaxDegLen)
{
    OPTTRY
    {
        if (uttStart < 0 || lDegSignal < 1 || mSegments == NULL)
            OPTTHROW( string("Invalid input parameters."));

        if (mSegments->size() == 0)
        {
            iDegStart  = 0;
            iMaxDegLen = lDegSignal;
            return;
        }

        TA_SegList::iterator nextSeg = mSegments->findInsLoc(uttStart);
        TA_SegList::iterator prevSeg = nextSeg == mSegments->begin() ?
            mSegments->end() : nextSeg - 1;

        //Find the segments surrounding uttStart to find a gap of unused degraded
signal.
        if (nextSeg != mSegments->end())
        {
            for (;
                 nextSeg != mSegments->end() &&
                 (nextSeg->segType == TA_SEG_MISSING || nextSeg->segType ==
TA_SEG_PAUSE);
                 nextSeg++);
        }
        if (prevSeg != mSegments->end())
        {
            for (;
                 prevSeg != mSegments->begin() &&
                 (prevSeg->segType == TA_SEG_MISSING || prevSeg->segType ==
TA_SEG_PAUSE);
                 prevSeg--);
            if (prevSeg->segType == TA_SEG_MISSING || prevSeg->segType == TA_SEG_PAUSE)
                prevSeg = mSegments->end();
        }

        if (prevSeg == mSegments->end())
            iDegStart = 0;
        else
            iDegStart = prevSeg->degPos + prevSeg->segLen;

        if (nextSeg == mSegments->end())
            iMaxDegLen = lDegSignal - iDegStart;
        else
            iMaxDegLen = nextSeg->degPos - iDegStart;

        flag = false;
```

```
        if (doCheck)
        {
            int leadingSilenceLen, trailingSilenceLen, i;

            for (i = iDegStart, leadingSilenceLen = 0;
                 i < iDegStart+iMaxDegLen && degSignal[i] == 0.0;
                 i++, leadingSilenceLen++);

            for (i = iMaxDegLen+iDegStart-1, trailingSilenceLen = 0;
                 i >= iDegStart+leadingSilenceLen && degSignal[i] == 0.0;
                 i--, trailingSilenceLen++);

            if (leadingSilenceLen + trailingSilenceLen >= iMaxDegLen)
                flag = true;
        }

        iMaxDegLen = (((iMaxDegLen) < (lDegSignal - iDegStart)) ? (iMaxDegLen) :
(lDegSignal - iDegStart));
        return;
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in CheckCurrentDegSeg: " + errorMsg + "\n"));
    }
}

XFLOAT SQTimeAlignment::SpanCorr(XFLOAT const *ref, XFLOAT const *deg,
                                 int const lDeg, int const iBestShift,
                                 XFLOAT const THR,
                                 XFLOAT const fMaxCorr,
                                 int& uttStartSamples, int& spanLen,
                                 XFLOAT *buff, XFLOAT *tmpBuff, int const
FD_MIN_SPANLEN)
{
    OPTTRY
    {
        if (ref == NULL || deg == NULL || lDeg <= 0 || spanLen <= 0 || buff == NULL ||
            tmpBuff == NULL || uttStartSamples < 0 || THR < 0)
            OPTTHROW( string("Invalid input arguments."));

        //Shorten spanLen if iBestShift makes it go past the deg. signal boundaries
        if (uttStartSamples + iBestShift < 0)
        {
            spanLen        += uttStartSamples + iBestShift;
            uttStartSamples = 0 - iBestShift;
        }
        spanLen = (((spanLen) < (lDeg - (uttStartSamples + iBestShift))) ? (spanLen) :
(lDeg - (uttStartSamples + iBestShift)));

        if (spanLen <= 0)
            return 0.0;

        XFLOAT fRefAutocorr, fDegAutocorr, fDivisor;
        svesq(ref+uttStartSamples, &fRefAutocorr, spanLen); fRefAutocorr /= spanLen;
        svesq(deg+uttStartSamples+iBestShift, &fDegAutocorr, spanLen); fDegAutocorr /=
spanLen;
        fDivisor = sqrt(fRefAutocorr * fDegAutocorr);

        vmul(ref+uttStartSamples, deg+uttStartSamples+iBestShift, buff, spanLen);

        XFLOAT fTotCorr = 0.0, fSpanCorr = 0.0;
        sve(buff, &fTotCorr, spanLen);
        if (fTotCorr < 0.0)
            vneg(buff, buff, spanLen);

        vsdiv(buff,    fDivisor, tmpBuff, spanLen);
        vsadd(tmpBuff, -THR,     buff,    spanLen);

        XFLOAT fMaxSum = 0.0, fSum = 0.0, fMaxSum2 = 0.0, fThr;
        int spanStart = 0, spanEnd, curSpanEnd, spanLen2;
        int curSpanStart = 0, curSpanLen = spanLen;
        int const FBS_MAX_SPANSTART = spanLen - FD_MIN_SPANLEN;

        //Compute span cross-correlation
        while (true)
        {
```

```
            for (;
                 spanStart < FBS_MAX_SPANSTART && buff[spanStart] <= 0.0;
                 spanStart++);
            if (spanStart > FBS_MAX_SPANSTART) break;

            curSpanEnd = spanEnd  = spanStart;
            fSum       = fMaxSum2 = fThr = 0.0;
            spanLen2   = 1;
            do
            {
                fSum += buff[spanEnd] - fThr;

                if (fSum > fMaxSum2 && spanLen2 >= FD_MIN_SPANLEN)
                    fMaxSum2 = fSum, curSpanEnd = spanEnd;

                spanEnd++;
                spanLen2 = spanEnd-spanStart+1;

                fThr = spanLen2 <= FD_MIN_SPANLEN ? 0.0 : (((0.2) <
(((XFLOAT)(spanLen2-FD_MIN_SPANLEN)/FD_MIN_SPANLEN) * 0.7 *
fMaxSum2/(curSpanEnd-spanStart+1))) ? (0.2) :
(((XFLOAT)(spanLen2-FD_MIN_SPANLEN)/FD_MIN_SPANLEN) * 0.7 *
fMaxSum2/(curSpanEnd-spanStart+1)));
            }
            while(spanEnd < spanLen && fSum > 0);

            if (fMaxSum2 > fMaxSum)
            {
                fMaxSum       = fMaxSum2;
                curSpanStart  = spanStart;
                curSpanLen    = curSpanEnd+1-spanStart;
            }
            spanStart = spanEnd;
        }

        //Also compute span correlation running backwards
        spanStart = 0;
        spanEnd   = spanLen-1;
        bool backwardSpanFound = false;
        int curSpanStart2 = 0, curSpanEnd2 = spanLen-1, curSpanLen2 = spanLen;
        while (true)
        {
            for (;
                 spanEnd > FD_MIN_SPANLEN-1 && buff[spanEnd] <= 0.0;
                 spanEnd--);
            if (spanEnd < FD_MIN_SPANLEN-1) break;

            curSpanStart2 = spanStart = spanEnd;
            fSum          = fMaxSum2  = fThr = 0.0;
            spanLen2      = 1;
            do
            {
                fSum += buff[spanStart] - fThr;

                if (fSum > fMaxSum2 && spanLen2 >= FD_MIN_SPANLEN)
                    fMaxSum2 = fSum, curSpanStart2 = spanStart;

                spanStart--;
                spanLen2 = spanEnd-spanStart+1;

                fThr = spanLen2 <= FD_MIN_SPANLEN ? 0.0 : (((0.2) <
(((XFLOAT)(spanLen2-FD_MIN_SPANLEN)/FD_MIN_SPANLEN) * 0.3 *
fMaxSum2/(spanEnd-curSpanStart2+1))) ? (0.2) :
(((XFLOAT)(spanLen2-FD_MIN_SPANLEN)/FD_MIN_SPANLEN) * 0.3 *
fMaxSum2/(spanEnd-curSpanStart2+1)));
            }
            while(spanStart >= 0 && fSum > 0);

            if (fMaxSum2 > fMaxSum)
            {
                fMaxSum     = fMaxSum2;
                curSpanEnd2 = spanEnd;
                curSpanLen2 = curSpanEnd2+1-curSpanStart2;
                backwardSpanFound = true;
            }
            spanEnd = spanStart;
```

```
        }

        XFLOAT fSpanCorr2;
        mve(buff+curSpanStart,  &fSpanCorr,  curSpanLen);
        mve(buff+curSpanStart2, &fSpanCorr2, curSpanLen2);
        if (backwardSpanFound && fSpanCorr2 > fSpanCorr)
        {
            curSpanStart = curSpanStart2;
            curSpanEnd   = curSpanEnd2;
            curSpanLen   = curSpanLen2;
            fSpanCorr    = fSpanCorr2;
        }

        fSpanCorr       += THR;
        uttStartSamples += curSpanStart;
        spanLen          = curSpanLen;

        return fSpanCorr;
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in SpanCorr: " + errorMsg + "\n"));
    }
}

void SQTimeAlignment::InsertSegment(int const &uttStart, int const &uttShift,
                                    int const &uttLen, int const &segType,
                                    int const &lDegSigLen,
                                    bool const &doMatchLoudestUtt, XFLOAT *env,
                                    int const &lEnv, int const &frameStep,
                                    XFLOAT const &envSilenceVal,

                                    XFLOAT guessFac)

{
    OPTTRY
    {
        if (uttStart < 0 || uttLen <= 0 ||
            ((uttShift + uttStart < 0 || uttStart + uttLen + uttShift > lDegSigLen) &&
             (segType != TA_SEG_MISSING)))
            OPTTHROW( string ("Invalid input arguments."));

        if (doMatchLoudestUtt)
            RemoveUttFromEnv(env, lEnv, frameStep, envSilenceVal, uttStart, uttLen);

        if (guessFac < 0.0)
            switch(segType)
            {
                case TA_SEG_MATCHED: guessFac = 1.0; break;
                case TA_SEG_GUESSED:
                case TA_SEG_PAUSE:   guessFac = 0.5; break;
                case TA_SEG_MISSING: guessFac = 0.0; break;
            }

        int uttMaxEnd = 0;
        switch(segType)
        {
        case TA_SEG_MATCHED:
        case TA_SEG_GUESSED:

                                mSegments->insert(mSegments->findInsLoc(uttStart),
                                                  TA_segStruct(uttStart,
                                                               uttStart + uttShift,
                                                               uttLen,
                                                               (TA_SEG_TYPE)segType,

                                                               false, guessFac));

                            break;
        case TA_SEG_MISSING:
        case TA_SEG_PAUSE:   mSegments->insert(mSegments->findInsLoc(uttStart),
                                               TA_segStruct(uttStart,
                                                            limit(uttStart + uttShift,
0, lDegSigLen-uttLen),
                                                            uttLen,
                                                            (TA_SEG_TYPE)segType,
```

```
                                                                false, guessFac));

            break;
        default:            OPTTHROW( string("Unrecognized TA_SEG_TYPE."));
        }
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in InsertSegment: " + errorMsg + "\n"));
    }
}

void SQTimeAlignment::RemoveUttFromEnv(XFLOAT *env, int const &lEnv,
                                       int const &frameStep, XFLOAT const
&envSilenceVal,
                                       int const &start, int const &len)
{
    OPTTRY
    {
        if (env == NULL || lEnv < 1 || frameStep < 1 || start < 0 || len < 0)
            OPTTHROW( string("Invalid input arguments."));

        //Convert from samples to frames
        int startFrames = (((start / frameStep) > (0)) ? (start / frameStep) : (0));
        int endFrames   = ((((start+len-1) / frameStep) < (lEnv-1)) ? ((start+len-1) /
frameStep) : (lEnv-1));

        for (int i = startFrames; i <= endFrames; i++)
            env[i] = envSilenceVal;

    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in RemoveUttFromEnv: " + errorMsg + "\n"));
    }
}

void SQTimeAlignment::WriteMatchedSegment(int &spanStart, int &spanLen, int const
&iBestShift,
                                          int const &iDegStart, int const &iMaxDegLen,
int const &lDegNorm,
                                          bool const &doRemoveCurUtt, XFLOAT *env, int
const &lEnv,
                                          int const &frameStep, XFLOAT const
&envSilenceVal)
{
    OPTTRY
    {
        if (spanStart < 0 || spanLen < 1 || iDegStart < 0 || iMaxDegLen < 1)
            OPTTHROW( string("Invalid input arguments."));

        //Check iBestShift: Are we intruding into unavailable deg. parts?
        if (spanStart+iBestShift < iDegStart)
        {
            int overLapLen = (((iDegStart - (spanStart+iBestShift)) < (spanLen)) ?
(iDegStart - (spanStart+iBestShift)) : (spanLen));
            InsertSegment(spanStart, iBestShift, overLapLen,
                          TA_SEG_MISSING, lDegNorm,
                          doRemoveCurUtt, env, lEnv,
                          frameStep, envSilenceVal);
            spanLen  -= overLapLen;
            spanStart = iDegStart - iBestShift;
        }

        if (spanStart+spanLen+iBestShift > iDegStart+iMaxDegLen)
        {
            int overLapLen =
                (((spanStart+spanLen+iBestShift - (iDegStart+iMaxDegLen)) < (spanLen))
? (spanStart+spanLen+iBestShift - (iDegStart+iMaxDegLen)) : (spanLen));
            InsertSegment(iDegStart+iMaxDegLen-iBestShift, iBestShift, overLapLen,
                          TA_SEG_MISSING, lDegNorm,
                          doRemoveCurUtt, env, lEnv,
                          frameStep, envSilenceVal);
            spanLen  -= overLapLen;
        }
```

```
        //Write remaining matched segment using iBestShift
        if (spanLen > 0)
            InsertSegment(spanStart, iBestShift, spanLen,
                          TA_SEG_MATCHED, lDegNorm,
                          doRemoveCurUtt, env, lEnv,
                          frameStep, envSilenceVal);
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in WriteMatchedSegment: " + errorMsg + "\n"));
    }
}

void SQTimeAlignment::CheckCurrentRefSeg(XFLOAT const *ref, int const lRef,
                                         int &uttStart, int &uttEnd,
                                         bool &flag, int const FD_MAX_SEGLEN)
{
    OPTTRY
    {
        if (ref == NULL || lRef <= 0 || uttStart < 0 || uttEnd < 0)
            OPTTHROW( string("Invalid input arguments."));

        uttEnd = (((uttEnd) < (mSegments->findMaxSegEndRef(uttEnd, lRef-1))) ? (uttEnd)
: (mSegments->findMaxSegEndRef(uttEnd, lRef-1)));

        //Avoid running into existing segments
        TA_SegList::iterator nextSeg = mSegments->findInsLoc(uttStart);
        TA_SegList::iterator prevSeg = nextSeg == mSegments->begin() ?
            mSegments->end() : nextSeg - 1;
        if (prevSeg != mSegments->end())
            uttStart = __max(prevSeg->refPos + prevSeg->segLen - 1, uttStart);
        if (uttStart > uttEnd)
            return;

        flag = false;
        int cnt = 0, i;
        int const THR = 256;
        for (i = uttStart; i < uttEnd+1; i++)
        {
            if (ref[i] == 0.0) cnt++;
            else
            {
                if (cnt >= THR) break;
                else            cnt = 0;
            }
        }
        if (cnt >= THR)
        {
            if (i - cnt == uttStart)
            {
                flag   = true;
                uttEnd = i - 1;
            }
            else
            {
                flag   = false;
                uttEnd = i - cnt - 1;
            }
        }

        if (!flag)
            uttEnd = (((uttEnd) < (uttStart + FD_MAX_SEGLEN - 1)) ? (uttEnd) :
(uttStart + FD_MAX_SEGLEN - 1));
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in CheckCurrentRefSeg: " + errorMsg + "\n"));
    }
    OPTCATCH((...))
    {
        OPTTHROW( string("ERROR in CheckCurrentRefSeg: Unknown error.\n"));
    }
}

void SQTimeAlignment::FixExtremeMatches(XFLOAT const *fRefNorm, XFLOAT const *fDegNorm,
```

```
                                        int const &lRefNorm, int const &lDegNorm)
{
    TA_SegList *SegListCopy = NULL;

    OPTTRY
    {
        if (lRefNorm < 1 || lDegNorm < 1 || fRefNorm == NULL || fDegNorm == NULL ||
            mSegments == NULL || mSegments->size() == 0)
            OPTTHROW( string("ERROR in FixExtremeMatches: Invalid input params or
segments list.\n"));

        int const FEM_MIN_NR_MATCHED_SEGS = 10;

        //Verify that there are enough matched segments for a good estimation
        int numMatchedSegs = 0;
        for (int i = 0; i < (int)mSegments->size(); i++)
            if ((*mSegments)[i].segType == TA_SEG_MATCHED)
                numMatchedSegs++;
        if (numMatchedSegs < FEM_MIN_NR_MATCHED_SEGS)
            return;

        //Estimate the range of 'realistic' values for the shifts betw. ref and deg.
        int lowerOuterFence_1stHalf, lowerOuterFence_2ndHalf,
            upperOuterFence_1stHalf, upperOuterFence_2ndHalf,
            endOf1stHalf;
        EstimateShiftsRange(numMatchedSegs, lRefNorm, endOf1stHalf,
                            lowerOuterFence_1stHalf, lowerOuterFence_2ndHalf,
                            upperOuterFence_1stHalf, upperOuterFence_2ndHalf);

        //Search and remove extreme outliers (i.e. beyond fences)
        int prevGoodSeg = -1, nextGoodSeg = -1, firstBadSeg = -1, lastBadSeg = -1;
        int numPrevBadActSegs = 0;
        bool badSegFound = false;
        for (int i = 0; i <= (int)mSegments->size(); i++)
        {
            if (i < (int)mSegments->size() &&
                ((*mSegments)[i].segType == TA_SEG_MATCHED ||
                 (*mSegments)[i].segType == TA_SEG_GUESSED ||

                 (badSegFound && (((*mSegments)[firstBadSeg].refPos < endOf1stHalf) ==
((*mSegments)[i].refPos < endOf1stHalf))))

                &&
                (((*mSegments)[i].refPos < endOf1stHalf &&
                  ((*mSegments)[i].degPos - (*mSegments)[i].refPos <
lowerOuterFence_1stHalf ||
                   (*mSegments)[i].degPos - (*mSegments)[i].refPos >
upperOuterFence_1stHalf))
                 ||
                 ((*mSegments)[i].refPos >= endOf1stHalf &&
                  ((*mSegments)[i].degPos - (*mSegments)[i].refPos <
lowerOuterFence_2ndHalf ||
                   (*mSegments)[i].degPos - (*mSegments)[i].refPos >
upperOuterFence_2ndHalf))
                 ||
                 (badSegFound && (*mSegments)[i].segType == TA_SEG_MISSING))
                &&
                (((*mSegments)[i].segType != TA_SEG_MATCHED && (*mSegments)[i].segType
!= TA_SEG_GUESSED)
                 ||

                 CorrCheck(&(*mSegments)[i], fRefNorm, lRefNorm, fDegNorm, lDegNorm,
numPrevBadActSegs)))

            {

                if (badSegFound)
                    lastBadSeg  = i;
                else
                {
                    firstBadSeg = i;
                    lastBadSeg  = i;
                    badSegFound = true;
                }
            }
            else
```

```
        {
            if (badSegFound)
            {
                if (SegListCopy == NULL)
                    SegListCopy = new TA_SegList(*mSegments);

                nextGoodSeg = i;

                int firstSpeechIdx = firstBadSeg, lastSpeechIdx = lastBadSeg;
                for (; firstSpeechIdx < (int)mSegments->size() &&
                     mSegments->at(firstSpeechIdx).segType == TA_SEG_PAUSE;
                     firstSpeechIdx++);
                for (; lastSpeechIdx >= firstSpeechIdx &&
                     mSegments->at(lastSpeechIdx ).segType == TA_SEG_PAUSE;
                     lastSpeechIdx--);
                int consecBadSpeechLen = mSegments->at(lastSpeechIdx).refPos +
                                         mSegments->at(lastSpeechIdx).segLen -
                                         mSegments->at(firstSpeechIdx).refPos;
                if (consecBadSpeechLen >= round(1.0f * TA_SAMPLING_RATE))
                {
                    TA_SegList badSegs;
                    badSegs.insert(badSegs.begin(),
                        mSegments->getIterator(firstBadSeg),
                        mSegments->getIterator(lastBadSeg+1));
                    XFLOAT longGroupMatchQual = (XFLOAT)-1.0;
                    OPTTRY
                    {
                        longGroupMatchQual = CalcMatchQuality(&badSegs);
                        if (longGroupMatchQual > 0.75f)
                        {
                            prevGoodSeg       = i;
                            badSegFound       = false;
                            firstBadSeg       = lastBadSeg = -1;
                            numPrevBadActSegs = 0;
                            continue;
                        }
                    }
                    OPTCATCH((...)){}
                }

                mExtremeMatchFound = true;

                int j = firstBadSeg-1;
                for (; j >= 0 &&
                     ((*mSegments)[j].segType == TA_SEG_MISSING ||
                      (*mSegments)[j].segType == TA_SEG_PAUSE);
                     j--);
                if (j < firstBadSeg-1 && j >= 0 &&
                    (*mSegments)[j].segType != TA_SEG_MISSING &&
                    (*mSegments)[j].segType != TA_SEG_PAUSE)
                {
                    prevGoodSeg = j;

                    if (((*mSegments)[j+1].refPos < endOf1stHalf) ==
((*mSegments)[firstBadSeg].refPos < endOf1stHalf) &&
                        (*mSegments)[j+1].refPos != endOf1stHalf)

                        firstBadSeg = j+1;
                }

                for (j = lastBadSeg+1; j < (int)mSegments->size() &&
                     ((*mSegments)[j].segType == TA_SEG_MISSING ||
                      (*mSegments)[j].segType == TA_SEG_PAUSE);
                     j++);
                if (j > lastBadSeg+1 && j < (int)mSegments->size() &&
                    (*mSegments)[j].segType != TA_SEG_MISSING &&
                    (*mSegments)[j].segType != TA_SEG_PAUSE)
                {
                    nextGoodSeg = j;

                    if (((*mSegments)[j-1].refPos < endOf1stHalf) ==
((*mSegments)[firstBadSeg].refPos < endOf1stHalf) &&
                        (*mSegments)[j-1].refPos != endOf1stHalf)

                        lastBadSeg  = j-1;
                }
```

```
                    bool doFixPausesAtStart = false, doFixPausesAtEnd = false;
                    CheckForSurroundingPauses(firstBadSeg, lastBadSeg,
                                              doFixPausesAtStart, doFixPausesAtEnd);

                    if (lastBadSeg < firstBadSeg)
                    {
                        prevGoodSeg       = i;
                        badSegFound       = false;
                        firstBadSeg       = lastBadSeg = -1;
                        numPrevBadActSegs = 0;
                        continue;
                    }

                    int iMaxDegLen, iDegStart, uttStart, uttLen;
                    GetSignalLengths(prevGoodSeg, firstBadSeg, lastBadSeg, nextGoodSeg,
                                     uttStart, uttLen, iDegStart, iMaxDegLen,
lDegNorm);

                    if (iDegStart < 0 || iMaxDegLen < 1 || uttStart < 0 || uttLen < 1)
                    {
                        prevGoodSeg       = i;
                        badSegFound       = false;
                        firstBadSeg       = lastBadSeg = -1;
                        numPrevBadActSegs = 0;
                        continue;
                    }

                    TA_SegList deletedSegments;
                    deletedSegments.insert(deletedSegments.begin(),
                                           mSegments->getIterator(firstBadSeg),
                                           mSegments->getIterator(lastBadSeg+1));

                    //Remove the bad segments.
                    mSegments->erase(mSegments->getIterator(firstBadSeg),
                                     mSegments->getIterator(lastBadSeg+1));

                    int guessedShift, GBS_iMaxDegLen = iMaxDegLen;
                    if (iDegStart + iMaxDegLen == lDegNorm)
                        GBS_iMaxDegLen = (((iMaxDegLen) > (uttLen)) ? (iMaxDegLen) :
(uttLen));
                    GuessBestShift(uttStart, iDegStart, GBS_iMaxDegLen, fDegNorm,
lDegNorm,
                                   uttLen, guessedShift, NULL, endOf1stHalf);

                    bool newShiftIsSameAsOld = true;
                    for (int k = 0; k < (int)deletedSegments.size(); k++)
                        if (deletedSegments[k].segType != TA_SEG_MISSING &&
                            deletedSegments[k].segType != TA_SEG_PAUSE &&
                            deletedSegments[k].degPos - deletedSegments[k].refPos !=
guessedShift)
                            newShiftIsSameAsOld &= false;

                    if (newShiftIsSameAsOld ||
                        uttStart+uttLen + guessedShift <=  0 ||
                        uttStart        + guessedShift >= lDegNorm)
                    {
                        mSegments->insert(mSegments->findInsLoc(deletedSegments[0].refP
os),
                                          deletedSegments.begin(),
deletedSegments.end());
                        prevGoodSeg       = i;
                        badSegFound       = false;
                        firstBadSeg       = lastBadSeg = -1;
                        numPrevBadActSegs = 0;
                        continue;
                    }

                    //Insert fixed segment.
                    WriteMatchedSegment(uttStart, uttLen, guessedShift,
                                        iDegStart, iMaxDegLen, lDegNorm, false,
                                        NULL, 0, 0, 0.0f);

                    if (uttLen > 0)
                    {
                        mSegments->findInsLoc(uttStart)->segType = TA_SEG_GUESSED;
```

```
                        mSegments->findInsLoc(uttStart)->dontMergeWithOthers = true;
                        mSegments->findInsLoc(uttStart)->reliability = 0.0f;
                    }

                    FixPauses(doFixPausesAtStart, doFixPausesAtEnd, lDegNorm);

                    //Move indices after modification of segments list.
                    prevGoodSeg       = mSegments->findInsLocIdx(uttStart);
                    i                 = prevGoodSeg;
                    badSegFound       = false;
                    firstBadSeg       = lastBadSeg = -1;
                    numPrevBadActSegs = 0;
                }
                else
                {
                    prevGoodSeg       = i;
                    numPrevBadActSegs = 0;
                }
            }
        }
        delete SegListCopy;
        SegListCopy = NULL;
        return;
    }
    OPTCATCH((string errorMsg))
    {
        mSegments->swap(*SegListCopy);
        delete SegListCopy;
        SegListCopy = NULL;

        return;

    }
    OPTCATCH((...))
    {
        mSegments->swap(*SegListCopy);
        delete SegListCopy;
        SegListCopy = NULL;

        return;

    }
}

void SQTimeAlignment::EstimateShiftsRange(int const &numMatchedSegs, int lRefNorm, int
&endOf1stHalf,
                                          int &lowerOuterFence_1stHalf, int
&lowerOuterFence_2ndHalf,
                                          int &upperOuterFence_1stHalf, int
&upperOuterFence_2ndHalf)
{
    long *shifts_1stHalf[2] = {NULL, NULL}, *shifts[2] = {NULL, NULL},
         *shifts_2ndHalf[2] = {NULL, NULL};
    int const ESR_MIN_FENCE_WIDTH
        = (((round(0.032f * TA_SAMPLING_RATE)) < (round(FRAME_LEN * TA_SAMPLING_RATE)))
? (round(0.032f * TA_SAMPLING_RATE)) : (round(FRAME_LEN * TA_SAMPLING_RATE)));
    int const ESR_MIN_MID_PAUSE_LEN
        = round(0.33f * TA_SAMPLING_RATE);

    OPTTRY
    {
        if (numMatchedSegs < 1 || lRefNorm < 1)
            OPTTHROW( string("Invalid input arguments."));

        shifts[0] = (long*)matMalloc(numMatchedSegs * sizeof(long));
        shifts[1] = (long*)matMalloc(numMatchedSegs * sizeof(long));
        int j = 0, i, counter, q[3], qNum, iqr,
            totMatchedLen = 0, totMatchedLen_1stHalf = 0, totMatchedLen_2ndHalf = 0,
            lastMatchedSegOf1stHalf = -1,
            longestMidPauseLen = ESR_MIN_MID_PAUSE_LEN;
        endOf1stHalf = lRefNorm;

        //Copy the shifts of matched segments to a shifts array,
        //and try to detect the middle pause between two sentences, if present.
        for (i = 0; i < (int)mSegments->size(); i++)
        {
```

```
            if ((*mSegments)[i].segType == TA_SEG_MATCHED)
            {
                shifts[0][j]   = (long)((*mSegments)[i].degPos -
(*mSegments)[i].refPos);
                shifts[1][j]   = (*mSegments)[i].segLen;
                totMatchedLen += (*mSegments)[i].segLen;
                j++;
            }
            if (j > 0 && j < numMatchedSegs && (*mSegments)[i].segType == TA_SEG_PAUSE
&&
                (*mSegments)[i].segLen > longestMidPauseLen)
            {
                longestMidPauseLen      = (*mSegments)[i].segLen;
                lastMatchedSegOf1stHalf = j-1;
                endOf1stHalf            = (*mSegments)[i].refPos;
                totMatchedLen_1stHalf   = totMatchedLen;
            }
        }
        if (lastMatchedSegOf1stHalf < 0)
        {
            XFLOAT matchedLenNorm = totMatchedLen / (XFLOAT)PI, pauseFac, maxPauseFac =
0.0f;
            int sumMatchedLen = 0, bestPauseIdx = -1, pauseLen;
            for (i = 0, j = 0; i < (int)mSegments->size(); i++)
            {
                if ((*mSegments)[i].segType != TA_SEG_MATCHED &&
                    (*mSegments)[i].segType != TA_SEG_PAUSE)
                    continue;
                if ((*mSegments)[i].segType == TA_SEG_MATCHED)
                {
                    sumMatchedLen += (*mSegments)[i].segLen;
                    j++;
                }
                else
                {
                    pauseLen = (*mSegments)[i].segLen;
                    pauseFac = sin(sumMatchedLen / matchedLenNorm) *
sqrt((XFLOAT)pauseLen);
                    if (pauseFac > maxPauseFac && sumMatchedLen < totMatchedLen)
                    {
                        longestMidPauseLen      = pauseLen;
                        lastMatchedSegOf1stHalf = j-1;
                        endOf1stHalf            = (*mSegments)[i].refPos;
                        totMatchedLen_1stHalf   = sumMatchedLen;
                        maxPauseFac             = pauseFac;
                    }
                }
            }
        }
        totMatchedLen_2ndHalf = totMatchedLen - totMatchedLen_1stHalf;

        //Determine the fences for extreme outliers by computing the quartiles
        int q2_1stHalf, q2_2ndHalf;
        if (lastMatchedSegOf1stHalf == -1)
        {
            sortTwoVectors(shifts[0], shifts[1], 1, numMatchedSegs);
            for (i = 0, counter = 0, qNum = 0; i < numMatchedSegs && qNum < 3; i++)
            {
                counter += shifts[1][i];
                if (counter >= round((qNum+1)*0.25f*totMatchedLen))
                    q[qNum++] = shifts[0][i];
            }
            if (qNum != 3)
            {
                if (qNum < 1)
                    OPTTHROW( string("Could not compute quartiles of shifts of matched
segments."));
                else
                    for (i = 3-1; i >= qNum; i--)
                        q[i] = q[qNum-1];
            }

            iqr = q[3-1] - q[1-1];
            q2_1stHalf = q2_2ndHalf = q[2-1];
            lowerOuterFence_1stHalf = lowerOuterFence_2ndHalf = q[1-1] - 3*iqr;
            upperOuterFence_1stHalf = upperOuterFence_2ndHalf = q[3-1] + 3*iqr;
```

```
        }
        else
        {
            shifts_1stHalf[0] = (long*)matMalloc((((lastMatchedSegOf1stHalf+1) > (1)) ?
(lastMatchedSegOf1stHalf+1) : (1)) * sizeof(long));
            shifts_1stHalf[1] = (long*)matMalloc((((lastMatchedSegOf1stHalf+1) > (1)) ?
(lastMatchedSegOf1stHalf+1) : (1)) * sizeof(long));
            shifts_2ndHalf[0] = (long*)matMalloc((((numMatchedSegs -
(lastMatchedSegOf1stHalf+1)) > (1)) ? (numMatchedSegs -
(lastMatchedSegOf1stHalf+1)) : (1)) * sizeof(long));
            shifts_2ndHalf[1] = (long*)matMalloc((((numMatchedSegs -
(lastMatchedSegOf1stHalf+1)) > (1)) ? (numMatchedSegs -
(lastMatchedSegOf1stHalf+1)) : (1)) * sizeof(long));
            ivmov(shifts[0], shifts_1stHalf[0], (((lastMatchedSegOf1stHalf+1) > (1)) ?
(lastMatchedSegOf1stHalf+1) : (1)));
            ivmov(shifts[1], shifts_1stHalf[1], (((lastMatchedSegOf1stHalf+1) > (1)) ?
(lastMatchedSegOf1stHalf+1) : (1)));
            ivmov(shifts[0]+lastMatchedSegOf1stHalf+1, shifts_2ndHalf[0],
(((numMatchedSegs - (lastMatchedSegOf1stHalf+1)) > (1)) ? (numMatchedSegs -
(lastMatchedSegOf1stHalf+1)) : (1)));
            ivmov(shifts[1]+lastMatchedSegOf1stHalf+1, shifts_2ndHalf[1],
(((numMatchedSegs - (lastMatchedSegOf1stHalf+1)) > (1)) ? (numMatchedSegs -
(lastMatchedSegOf1stHalf+1)) : (1)));

            sortTwoVectors(shifts_1stHalf[0], shifts_1stHalf[1], 1,
lastMatchedSegOf1stHalf+1);
            for (i = 0, counter = 0, qNum = 0; i < (((lastMatchedSegOf1stHalf+1) > (1))
? (lastMatchedSegOf1stHalf+1) : (1)) && qNum < 3; i++)
            {
                counter += shifts_1stHalf[1][i];
                if (counter >= round((qNum+1)*0.25f*totMatchedLen_1stHalf))
                    q[qNum++] = shifts_1stHalf[0][i];
            }
            if (qNum != 3)
            {
                if (qNum < 1)
                    OPTTHROW( string("Could not compute quartiles of shifts of matched
segments."));
                else
                    for (i = 3-1; i >= qNum; i--)
                        q[i] = q[qNum-1];
            }

            iqr = q[3-1] - q[1-1];
            q2_1stHalf = q[2-1];
            lowerOuterFence_1stHalf = q[1-1] - 3*iqr,
            upperOuterFence_1stHalf = q[3-1] + 3*iqr;

            sortTwoVectors(shifts_2ndHalf[0], shifts_2ndHalf[1], 1,
numMatchedSegs-(lastMatchedSegOf1stHalf+1));
            for (i = 0, counter = 0, qNum = 0; i < (((numMatchedSegs -
(lastMatchedSegOf1stHalf+1)) > (1)) ? (numMatchedSegs -
(lastMatchedSegOf1stHalf+1)) : (1)) && qNum < 3; i++)
            {
                counter += shifts_2ndHalf[1][i];
                if (counter >= round((qNum+1)*0.25f*totMatchedLen_2ndHalf))
                    q[qNum++] = shifts_2ndHalf[0][i];
            }
            if (qNum != 3)
            {
                if (qNum < 1)
                    OPTTHROW( string("Could not compute quartiles of shifts of matched
segments."));
                else
                    for (i = 3-1; i >= qNum; i--)
                        q[i] = q[qNum-1];
            }

            iqr = q[3-1] - q[1-1];
            q2_2ndHalf = q[2-1];
            lowerOuterFence_2ndHalf = q[1-1] - 3*iqr,
            upperOuterFence_2ndHalf = q[3-1] + 3*iqr;
        }

        if (upperOuterFence_1stHalf - lowerOuterFence_1stHalf < ESR_MIN_FENCE_WIDTH)
        {
```

```
            lowerOuterFence_1stHalf = q2_1stHalf - ESR_MIN_FENCE_WIDTH/2;
            upperOuterFence_1stHalf = q2_1stHalf + ESR_MIN_FENCE_WIDTH/2;
        }
        if (upperOuterFence_2ndHalf - lowerOuterFence_2ndHalf < ESR_MIN_FENCE_WIDTH)
        {
            lowerOuterFence_2ndHalf = q2_2ndHalf - ESR_MIN_FENCE_WIDTH/2;
            upperOuterFence_2ndHalf = q2_2ndHalf + ESR_MIN_FENCE_WIDTH/2;
        }

        matFree(shifts[0]);
        matFree(shifts[1]);
        matFree(shifts_1stHalf[0]);
        matFree(shifts_1stHalf[1]);
        matFree(shifts_2ndHalf[0]);
        matFree(shifts_2ndHalf[1]);
        shifts[0] = shifts[1] = shifts_1stHalf[0] = shifts_1stHalf[1] =
shifts_2ndHalf[0] =
            shifts_2ndHalf[1] = NULL;
    }
    OPTCATCH((string errorMsg))
    {
        matFree(shifts[0]);
        matFree(shifts[1]);
        matFree(shifts_1stHalf[0]);
        matFree(shifts_1stHalf[1]);
        matFree(shifts_2ndHalf[0]);
        matFree(shifts_2ndHalf[1]);
        shifts[0] = shifts[1] = shifts_1stHalf[0] = shifts_1stHalf[1] =
shifts_2ndHalf[0] =
            shifts_2ndHalf[1] = NULL;
        OPTTHROW( string("ERROR in EstimateShiftsRange: " + errorMsg + "\n"));
    }
    OPTCATCH((...))
    {
        matFree(shifts[0]);
        matFree(shifts[1]);
        matFree(shifts_1stHalf[0]);
        matFree(shifts_1stHalf[1]);
        matFree(shifts_2ndHalf[0]);
        matFree(shifts_2ndHalf[1]);
        shifts[0] = shifts[1] = shifts_1stHalf[0] = shifts_1stHalf[1] =
shifts_2ndHalf[0] =
            shifts_2ndHalf[1] = NULL;
        OPTTHROW( string("ERROR in EstimateShiftsRange: Unknown error.\n"));
    }
}

bool SQTimeAlignment::CorrCheck(TA_segStruct const *curSeg,
                                XFLOAT const *fRefNorm, int lRefNorm,
                                XFLOAT const *fDegNorm, int lDegNorm,
                                int &numPrevBadActSegs)
{
    if (curSeg == NULL || curSeg->refPos < 0 || curSeg->degPos < 0 ||
        curSeg->segLen < 1 ||
        (curSeg->segType != TA_SEG_MATCHED && curSeg->segType != TA_SEG_GUESSED) ||
        fRefNorm == NULL || fDegNorm == NULL || lRefNorm < curSeg->refPos +
curSeg->segLen ||
        lDegNorm < curSeg->degPos + curSeg->segLen)
        OPTTHROW( string("ERROR in CorrCheck: Invalid input arguments.\n"));

    XFLOAT const *refSpan = fRefNorm + curSeg->refPos;
    XFLOAT const *degSpan = fDegNorm + curSeg->degPos;
    int    spanLen = curSeg->segLen;

    XFLOAT fRefAutocorr, fDegAutocorr, fDivisor;
    svesq(refSpan, &fRefAutocorr, spanLen);
    svesq(degSpan, &fDegAutocorr, spanLen);
    fDivisor = sqrt(fRefAutocorr * fDegAutocorr);

    if (AlmostEqualUlpsFinal((float)fRefAutocorr, (float)0.0) &&
AlmostEqualUlpsFinal((float)fDegAutocorr, (float)0.0))
        return false;
    else if (AlmostEqualUlpsFinal((float)fDivisor, (float)0.0))
        return true;

    XFLOAT crossCorr;
```

```
    dotpr(refSpan, degSpan, &crossCorr, spanLen);
    crossCorr = fabs(crossCorr / fDivisor);

    XFLOAT corrThr = pow(0.6, 1.0 / (XFLOAT)limit(numPrevBadActSegs, 1, 4));

    if (crossCorr < corrThr)
    {
        XFLOAT refPow, degPow;
        rmvesq(mRef->Data() + curSeg->refPos, &refPow, spanLen);
        refPow /= mRef->MaxAmplitude();
        rmvesq(mDeg->Data() + curSeg->degPos, &degPow, spanLen);
        degPow /= mDeg->MaxAmplitude();

        if (rmsTodB(refPow) > mRef->CurrentASL() - 24.0 &&
            rmsTodB(degPow) > mDeg->CurrentASL() - 24.0)
            numPrevBadActSegs++;
    }

    return crossCorr < corrThr;
}

void SQTimeAlignment::CheckForSurroundingPauses(int& firstBadSeg, int& lastBadSeg,
                                                bool& doFixPausesAtStart,
                                                bool& doFixPausesAtEnd)
{
    if (firstBadSeg < 0 || lastBadSeg >= (int)mSegments->size() || firstBadSeg >
lastBadSeg)
        OPTTHROW( string("ERROR in CheckForSurroundingPauses: Invalid input segment
indices.\n"));

    int j = lastBadSeg+1;
    for (; j < (int)mSegments->size() &&
         ((*mSegments)[j].segType == TA_SEG_PAUSE ||
          (*mSegments)[j].segType == TA_SEG_MISSING);
         j++);

    //Only pause/missing segments following, the first following segment being a pause
or missing?
    if (j == (int)mSegments->size() && lastBadSeg+1 < (int)mSegments->size() &&
        ((*mSegments)[lastBadSeg+1].segType == TA_SEG_PAUSE ||
         (*mSegments)[lastBadSeg+1].segType == TA_SEG_MISSING))
    {
        for (j--; j >= firstBadSeg &&
             ((*mSegments)[j].segType == TA_SEG_PAUSE ||
              (*mSegments)[j].segType == TA_SEG_MISSING);
             j--);
        lastBadSeg = j;
        doFixPausesAtEnd = true;
    }

    for (j = firstBadSeg-1; j >= 0 &&
         ((*mSegments)[j].segType == TA_SEG_PAUSE ||
          (*mSegments)[j].segType == TA_SEG_MISSING);
         j--);

    //Only pause/missing segments preceding, the first preceding segment being a pause
or missing?
    if (j < 0 && firstBadSeg-1 >= 0 &&
        ((*mSegments)[firstBadSeg-1].segType == TA_SEG_PAUSE ||
         (*mSegments)[firstBadSeg-1].segType == TA_SEG_MISSING))
    {
        for (j++; j < (int)mSegments->size() &&
             ((*mSegments)[j].segType == TA_SEG_PAUSE ||
              (*mSegments)[j].segType == TA_SEG_MISSING);
             j++);
        firstBadSeg = j;
        doFixPausesAtStart = true;
    }
}

void SQTimeAlignment::GetSignalLengths(int prevGoodSeg, int firstBadSeg,
                                       int lastBadSeg, int nextGoodSeg,
                                       int& uttStart, int& uttLen, int& iDegStart,
                                       int& iMaxDegLen, int lDegNorm)
{
    if (prevGoodSeg > nextGoodSeg || firstBadSeg < 0 || firstBadSeg > lastBadSeg ||
```

```
        lastBadSeg >= (int)mSegments->size())
        OPTTHROW( string("ERROR in GetSignalLengths: Invalid input segment
indices.\n"));

    if (nextGoodSeg == (int)mSegments->size() ||
        (*mSegments)[nextGoodSeg].segType == TA_SEG_PAUSE ||
        (*mSegments)[nextGoodSeg].segType == TA_SEG_MISSING)
    {
        if (prevGoodSeg < 0)

        return;

        iDegStart  = (*mSegments)[prevGoodSeg].degPos +
(*mSegments)[prevGoodSeg].segLen;
        iMaxDegLen = lDegNorm - iDegStart;
        uttStart   = ((((*mSegments)[prevGoodSeg].refPos +
(*mSegments)[prevGoodSeg].segLen) > ((*mSegments)[firstBadSeg-1].refPos +
(*mSegments)[firstBadSeg-1].segLen)) ? ((*mSegments)[prevGoodSeg].refPos +
(*mSegments)[prevGoodSeg].segLen) : ((*mSegments)[firstBadSeg-1].refPos +
(*mSegments)[firstBadSeg-1].segLen));
        uttLen     = ((*mSegments)[lastBadSeg].refPos +
(*mSegments)[lastBadSeg].segLen) - uttStart;
    }
    else if (prevGoodSeg < nextGoodSeg && prevGoodSeg >= 0)
    {
        if ((*mSegments)[prevGoodSeg].segType != TA_SEG_MISSING)
        {
            iMaxDegLen = (*mSegments)[nextGoodSeg].degPos -
((*mSegments)[prevGoodSeg].degPos + (*mSegments)[prevGoodSeg].segLen);
            iDegStart  = (*mSegments)[prevGoodSeg].degPos +
(*mSegments)[prevGoodSeg].segLen;
            uttStart   = ((((*mSegments)[prevGoodSeg].refPos +
(*mSegments)[prevGoodSeg].segLen) > ((*mSegments)[firstBadSeg-1].refPos +
(*mSegments)[firstBadSeg-1].segLen)) ? ((*mSegments)[prevGoodSeg].refPos +
(*mSegments)[prevGoodSeg].segLen) : ((*mSegments)[firstBadSeg-1].refPos +
(*mSegments)[firstBadSeg-1].segLen));
            uttLen     = ((*mSegments)[lastBadSeg].refPos +
(*mSegments)[lastBadSeg].segLen) - uttStart;
        }
        else
        {
            iDegStart  = prevGoodSeg == 0 ? 0 : ((*mSegments)[prevGoodSeg-1].degPos +
(*mSegments)[prevGoodSeg-1].segLen);
            iMaxDegLen = (*mSegments)[nextGoodSeg].degPos - iDegStart;
            uttStart   = ((((*mSegments)[prevGoodSeg].refPos +
(*mSegments)[prevGoodSeg].segLen) > ((*mSegments)[firstBadSeg-1].refPos +
(*mSegments)[firstBadSeg-1].segLen)) ? ((*mSegments)[prevGoodSeg].refPos +
(*mSegments)[prevGoodSeg].segLen) : ((*mSegments)[firstBadSeg-1].refPos +
(*mSegments)[firstBadSeg-1].segLen));
            uttLen     = ((*mSegments)[lastBadSeg].refPos +
(*mSegments)[lastBadSeg].segLen) - uttStart;
        }
    }
    else
    {
        iMaxDegLen = (*mSegments)[nextGoodSeg].degPos;
        iDegStart  = 0;
        uttStart   = 0;
        uttLen     = ((*mSegments)[lastBadSeg].refPos +
(*mSegments)[lastBadSeg].segLen);
    }
}

void SQTimeAlignment::FixPauses(bool doFixPausesAtStart, bool doFixPausesAtEnd, int
lDegNorm)
{
    if (doFixPausesAtEnd)
    {
        int  pauseStart      = (int)mSegments->size()-1;
        int  totPauseLen     = 0;
        int  minDegPausePos  = lDegNorm;
        bool onlyMissingSegs = true;
        while (pauseStart >= 0 &&
               ((*mSegments)[pauseStart].segType == TA_SEG_PAUSE ||
                (*mSegments)[pauseStart].segType == TA_SEG_MISSING))
        {
```

```
            totPauseLen     += (*mSegments)[pauseStart].segLen;
            onlyMissingSegs &= (*mSegments)[pauseStart].segType == TA_SEG_MISSING;
            minDegPausePos   = (((minDegPausePos) < ((*mSegments)[pauseStart].degPos))
? (minDegPausePos) : ((*mSegments)[pauseStart].degPos));
            pauseStart--;
        }

        int lastUttSeg = pauseStart;
        pauseStart++;
        int pauseEndPos = (*mSegments)[pauseStart].degPos + totPauseLen - 1;

        if (!onlyMissingSegs && lastUttSeg >= 0 && totPauseLen > 0 && pauseEndPos <
lDegNorm-1 &&
            (*mSegments)[lastUttSeg].degPos + (*mSegments)[lastUttSeg].segLen >
(*mSegments)[pauseStart].degPos)
        {
            int lastUttEndPos = (*mSegments)[lastUttSeg].degPos +
(*mSegments)[lastUttSeg].segLen - 1;
            int shiftLen      =

            mSegments->erase(mSegments->getIterator(pauseStart+1),
                             mSegments->end());

            (*mSegments)[pauseStart].degPos += shiftLen;
            (*mSegments)[pauseStart].segLen  = totPauseLen;
        }
        else if (onlyMissingSegs && lastUttSeg >= 0 && totPauseLen > 0)
        {
            (*mSegments)[pauseStart].degPos = minDegPausePos;
            (*mSegments)[pauseStart].segLen = totPauseLen;

            mSegments->erase(mSegments->getIterator(pauseStart+1),
                             mSegments->end());
        }
    }

    if (doFixPausesAtStart)
    {
        int  pauseEnd        = 0;
        int  totPauseLen     = 0;
        int  minDegPausePos  = lDegNorm;
        bool onlyMissingSegs = true;
        while (pauseEnd < (int)mSegments->size() &&
               ((*mSegments)[pauseEnd].segType == TA_SEG_PAUSE ||
                (*mSegments)[pauseEnd].segType == TA_SEG_MISSING))
        {
            totPauseLen     += (*mSegments)[pauseEnd].segLen;
            onlyMissingSegs &= (*mSegments)[pauseEnd].segType == TA_SEG_MISSING;
            minDegPausePos   = (((minDegPausePos) < ((*mSegments)[pauseEnd].degPos)) ?
(minDegPausePos) : ((*mSegments)[pauseEnd].degPos));
            pauseEnd++;
        }

        int firstUttSeg = pauseEnd;
        pauseEnd--;
        int pauseStartPos = totPauseLen - ((*mSegments)[pauseEnd].degPos +
(*mSegments)[pauseEnd].segLen);

        if (!onlyMissingSegs && firstUttSeg < (int)mSegments->size() && totPauseLen > 0
&& pauseStartPos > 0 &&
            (*mSegments)[pauseEnd].degPos + (*mSegments)[pauseEnd].segLen >
(*mSegments)[firstUttSeg].degPos)
        {
            int pauseEndPos = (*mSegments)[pauseEnd].degPos +
(*mSegments)[pauseEnd].segLen - 1;
            int shiftLen    =

            (*mSegments)[pauseEnd].degPos -= shiftLen;
            (*mSegments)[pauseEnd].segLen  = totPauseLen;

            mSegments->erase(mSegments->begin(),
                             mSegments->getIterator(pauseEnd));
        }
        else if (onlyMissingSegs && firstUttSeg < (int)mSegments->size() && totPauseLen
> 0)
        {
```

```
            (*mSegments)[pauseEnd].refPos = (*mSegments)[0].refPos;
            (*mSegments)[pauseEnd].degPos = minDegPausePos;
            (*mSegments)[pauseEnd].segLen = totPauseLen;

            mSegments->erase(mSegments->begin(),
                             mSegments->getIterator(pauseEnd));
        }
    }
}

void SQTimeAlignment::FixIncorrectMatches()
{
    OPTTRY
    {
        if (mSegments == NULL || mSegments->size() == 0)
            OPTTHROW( string("Invalid input parameters."));
        if (mSegments->size() <= 4)
            return;

        int const FIM_MIN_FIXABLE_GAP_LEN =
            round(10e-3f * mCurSegmentRate);
        int const FIM_MAX_FIXABLE_GAP_LEN =
            round(50e-3f * mCurSegmentRate);

        TA_SegList::iterator seg = mSegments->end(), matchedSeg = mSegments->end();
        int listLen = (int)mSegments->size(), degGapLen = -1;

        for (int i = 0; i < listLen; i++)
        {
            if ((seg = mSegments->getIterator(i))->segType != TA_SEG_MISSING ||
                seg->segLen < FIM_MIN_FIXABLE_GAP_LEN || seg->segLen >
FIM_MAX_FIXABLE_GAP_LEN)
                continue;

            if (i+2 < listLen && (seg+1)->segType == TA_SEG_MATCHED &&
                (seg+2)->degPos > (seg+1)->degPos + (seg+1)->segLen &&
                (seg+2)->segType != TA_SEG_MISSING)
            {
                if ((i+3 >= listLen || (seg+3)->segType != TA_SEG_MISSING) &&
                    (i-2 <  0       || (seg-2)->degPos + (seg-2)->segLen ==
(seg-1)->degPos))
                {
                    matchedSeg = seg+1;
                    degGapLen  = (seg+2)->degPos - (matchedSeg->degPos +
matchedSeg->segLen);
                }
            }

            else if (i > 2 && (seg-1)->segType == TA_SEG_MATCHED &&
                     (seg-2)->degPos + (seg-2)->segLen < (seg-1)->degPos &&
                     (seg-2)->segType != TA_SEG_MISSING)
            {
                if ((i-3 <  0       || (seg-3)->segType != TA_SEG_MISSING) &&
                    (i+2 >= listLen || (seg+2)->degPos == (seg+1)->degPos +
(seg+1)->segLen))
                {
                    matchedSeg = seg-1;
                    degGapLen  = matchedSeg->degPos - ((seg-2)->degPos +
(seg-2)->segLen);
                }
            }

            if (matchedSeg != mSegments->end() &&
                degGapLen >= FIM_MIN_FIXABLE_GAP_LEN &&
                degGapLen <= FIM_MAX_FIXABLE_GAP_LEN)
            {
                matchedSeg->segType     = TA_SEG_GUESSED;
                matchedSeg->reliability = 0.0f;
            }

            matchedSeg = mSegments->end();
            degGapLen  = -1;
        }
    }
    OPTCATCH((string errorMsg))
    {
```

```
        OPTTHROW( string("ERROR in FixIncorrectMatches: " + errorMsg + "\n"));
    }
    OPTCATCH((...))
    {
        OPTTHROW( string("ERROR in FixIncorrectMatches: Unknown error.\n"));
    }
}

void SQTimeAlignment::MergeConsecutiveSegments()
{
    if (mSegments == NULL || mSegments->size() == 0)
        OPTTHROW( string("ERROR in MergeConsecutiveSegments: No segments list.\n"));

    //Merge consecutive segments of same type with identical shift
    for (int i = 0; i < (int)mSegments->size()-1; i++)
    {
        int   lastConsecSeg;
        int   totConsecSegLen = 0,
              curShift        = (*mSegments)[i].degPos - (*mSegments)[i].refPos;
        XFLOAT avgReliability  = (XFLOAT)0.0;

        for (lastConsecSeg  = i+1;
             lastConsecSeg < (int)mSegments->size() &&
             (*mSegments)[lastConsecSeg].segType == (*mSegments)[i].segType &&
             ((*mSegments)[i].segType == TA_SEG_MISSING ||
              (*mSegments)[lastConsecSeg].degPos - (*mSegments)[lastConsecSeg].refPos
== curShift);
             lastConsecSeg++)
        {
            totConsecSegLen += (*mSegments)[lastConsecSeg].segLen;
            avgReliability  += (*mSegments)[lastConsecSeg].reliability;
        }

        //Found 1 or more consecutive segs w/ same shift and same type?
        if (totConsecSegLen > 0)
        {
            mSegments->erase(mSegments->getIterator(i+1),
                             mSegments->getIterator(lastConsecSeg));

            //Calculate new shift value for consecutive TA_SEG_MISSING segments
            if ((*mSegments)[i].segType == TA_SEG_MISSING)
            {
                int degStart = (*mSegments)[i].degPos + (*mSegments)[i].segLen/2;
                (*mSegments)[i].degPos = (((0) > (degStart - ((*mSegments)[i].segLen +
totConsecSegLen)/2)) ? (0) : (degStart - ((*mSegments)[i].segLen +
totConsecSegLen)/2));
            }

            //Expand the first segment of the series
            (*mSegments)[i].segLen += totConsecSegLen;

            if ((*mSegments)[i].segType == TA_SEG_PAUSE || (*mSegments)[i].segType ==
TA_SEG_GUESSED)
                (*mSegments)[i].reliability = avgReliability /
(XFLOAT)(lastConsecSeg-i);
        }
    }
}

void SQTimeAlignment::FineDelayPostProc(XFLOAT const *fDegNorm, int const &lDegNorm,
                                        int const &lRefNorm)
{
    OPTTRY
    {
        if (lDegNorm <= 0 || lRefNorm <= 0 || fDegNorm == NULL ||
            mSegments == NULL || mSegments->size() == 0)
            OPTTHROW( string("Invalid input parameters."));

        for (TA_SegList::iterator seg = mSegments->begin(); seg != mSegments->end();
seg++)
        {
            if (seg->segType == TA_SEG_MISSING)
            {
                if (seg != mSegments->begin() && (seg-1)->segType == TA_SEG_GUESSED)
                    (seg-1)->dontMergeWithOthers = true;
                if (seg != mSegments->end()-1 && (seg+1)->segType == TA_SEG_GUESSED)
```

```
                    (seg+1)->dontMergeWithOthers = true;
            }
        }

        delete mMergedSegments;
        mMergedSegments = new TA_SegList;

        for (int i = 0; i < (int)mSegments->size(); i++)
        {
            if ((*mSegments)[i].segType == TA_SEG_MISSING ||
(*mSegments)[i].dontMergeWithOthers)
            {
                mMergedSegments->insert(mMergedSegments->getIterator(i),
(*mSegments)[i]);
                continue;
            }

            TA_SEG_TYPE curSegType   = (*mSegments)[i].segType,
                        prevSegType  = TA_SEG_PAUSE,
                        firstSegType = (*mSegments)[i].segType,
                        finalSegType = curSegType;
            int lastConsecSeg,
                lastMatchedSeg,
                lenTillLastMatchedSeg= 0,
                curSegLen            = (*mSegments)[i].segLen,
                totConsecSegLen      = curSegLen,
                curShift             = (*mSegments)[i].degPos - (*mSegments)[i].refPos,
                prevMatchedSegLen    = curSegType == TA_SEG_MATCHED? curSegLen : 0,
                prevGuessedSegLen    = curSegType != TA_SEG_MATCHED? curSegLen : 0;
            XFLOAT avgWeightedRlblt   = (*mSegments)[i].reliability * curSegLen;

            //Find last consecutive seg w/ same or similar shift
            for (lastConsecSeg = i+1, lastMatchedSeg = -1;
                 lastConsecSeg < (int)mSegments->size() &&

                 (*mSegments)[lastConsecSeg].segType != TA_SEG_MISSING &&
                 !(*mSegments)[lastConsecSeg].dontMergeWithOthers &&

                 (((*mSegments)[lastConsecSeg].segType != TA_SEG_PAUSE && firstSegType
!= TA_SEG_PAUSE) ||
                  ((*mSegments)[lastConsecSeg].segType == TA_SEG_PAUSE && firstSegType
== TA_SEG_PAUSE))
                 &&

                 abs( (*mSegments)[lastConsecSeg].degPos -
(*mSegments)[lastConsecSeg].refPos - curShift ) <= TA_SHIFT_TOLERANCE
                 &&

                 (prevSegType != TA_SEG_PAUSE || firstSegType == TA_SEG_PAUSE ||
                  firstSegType == TA_SEG_MATCHED || (*mSegments)[lastConsecSeg].segType
== TA_SEG_GUESSED) &&

                 (*mSegments)[lastConsecSeg].refPos + curShift +
(*mSegments)[lastConsecSeg].segLen <= mDegLen;

                 lastConsecSeg++)
            {
                curSegLen        = (*mSegments)[lastConsecSeg].segLen;
                curSegType       = (*mSegments)[lastConsecSeg].segType;
                totConsecSegLen += curSegLen;
                avgWeightedRlblt+= (*mSegments)[lastConsecSeg].reliability * curSegLen;

                if (curSegType == TA_SEG_MATCHED)
                {
                    if (curSegLen + prevMatchedSegLen > 5*prevGuessedSegLen)
                        finalSegType = curSegType;

                    lastMatchedSeg        = lastConsecSeg;
                    lenTillLastMatchedSeg = totConsecSegLen;
                    prevMatchedSegLen    += curSegLen;
                }
                else
                {
                    if (curSegLen + prevGuessedSegLen > prevMatchedSegLen/5)
                        finalSegType = TA_SEG_GUESSED;
```

```
                    prevGuessedSegLen += curSegLen;
                }
            }

            if (totConsecSegLen > (*mSegments)[i].segLen)
            {   //found 1 or more consecutive segs w/ same or similar shift

                if (firstSegType == TA_SEG_PAUSE)
                    finalSegType = TA_SEG_PAUSE;

                if (finalSegType == TA_SEG_GUESSED && lastMatchedSeg > i &&
                    lenTillLastMatchedSeg >= 0.75f * totConsecSegLen &&
                    prevMatchedSegLen > lenTillLastMatchedSeg - prevMatchedSegLen)
                {
                    mMergedSegments->insert(mMergedSegments->getIterator(i),
                                            TA_segStruct((*mSegments)[i].refPos,
(*mSegments)[i].degPos,
                                                         lenTillLastMatchedSeg,
TA_SEG_MATCHED,
                                                         false,
avgWeightedRlblt/totConsecSegL
en));
                    prevSegType = TA_SEG_MATCHED;
                    i = lastMatchedSeg - 1;
                }
                else
                {
                    mMergedSegments->insert(mMergedSegments->getIterator(i),
                                            TA_segStruct((*mSegments)[i].refPos,
(*mSegments)[i].degPos,
                                                         totConsecSegLen, finalSegType,
                                                         false,
avgWeightedRlblt/totConsecSegL
en));

                    prevSegType = curSegType;
                    i = lastConsecSeg - 1;
                }
            }
            else
            {
                mMergedSegments->insert(mMergedSegments->getIterator(i),
(*mSegments)[i]);
                prevSegType = curSegType;
            }
        }

        //Fill mUnusedDegSegments with, well, unused deg signal parts!
        delete mUnusedDegSegments;
        mUnusedDegSegments = new TA_SegList();
        int unusedDegSegStart = 0, unusedDegSegLen = 0, suggestedShift = 0;
        while (mSegments->findNextGapInDegSegList(unusedDegSegStart, unusedDegSegLen,
suggestedShift,
                                                  fDegNorm, lDegNorm, lRefNorm)
               == 0)
        {
            int suggestedRefPos = unusedDegSegStart - suggestedShift;
            mUnusedDegSegments->insert(mUnusedDegSegments->findInsLocDeg(unusedDegSegSt
art),
                                       TA_segStruct(suggestedRefPos, unusedDegSegStart,
                                                    unusedDegSegLen, TA_SEG_MISSING,
                                                    false, 0.0f));
            unusedDegSegStart += unusedDegSegLen;
        }

        //Scale all segment lists to mTargetRate
        for (int i = 0; i < (int)mSegments->size(); i++)
        {
            (*mSegments)[i].refPos *= mTargetRate/TA_SAMPLING_RATE;
            (*mSegments)[i].degPos *= mTargetRate/TA_SAMPLING_RATE;
            (*mSegments)[i].segLen *= mTargetRate/TA_SAMPLING_RATE;
        }
        for (int i = 0; i < (int)mMergedSegments->size(); i++)
        {
            (*mMergedSegments)[i].refPos *= mTargetRate/TA_SAMPLING_RATE;
            (*mMergedSegments)[i].degPos *= mTargetRate/TA_SAMPLING_RATE;
```

```
            (*mMergedSegments)[i].segLen *= mTargetRate/TA_SAMPLING_RATE;
        }
        for (int i = 0; i < (int)mUnusedDegSegments->size(); i++)
        {
            (*mUnusedDegSegments)[i].refPos *= mTargetRate/TA_SAMPLING_RATE;
            (*mUnusedDegSegments)[i].degPos *= mTargetRate/TA_SAMPLING_RATE;
            (*mUnusedDegSegments)[i].segLen *= mTargetRate/TA_SAMPLING_RATE;
        }
        mCurSegmentRate = mTargetRate;
    }
    OPTCATCH((string errorMsg))
    {
        OPTTHROW( string("ERROR in FineDelayPostProc: " + errorMsg + "\n"));
    }
}

}
```