

Memoria Práctica 3

Algoritmo de detección de pitch

Roger Reguan Font

Miquel Torrecilla Mercado

Processament d'àudio i veu

26 / 3 / 2020

Introducción

EL objetivo de esta práctica es crear un algoritmo que nos permita identificar el pitch (frecuencia fundamental) de un tramo de una señal de audio con voz. La práctica se basa en el uso de la autocorrelación. Deberemos implementar un sistema que reconozca las tramas de voz y entonces calcular el pitch de dicha trama. Las que no sean de voz directamente las descartaremos.

Para poder evaluar las prestaciones de nuestro algoritmo será necesario entrenarlo usando una base de datos y así poder minimizar el error cometido. Además nos introduciremos en el uso de diferentes conceptos de C++, como las clases, vectores, string iostream, etc.

Tarea 1

Una vez creado el fichero index.html con el doxygen vamos a nuestro navegador y abrimos dicho archivo. Si buscamos en el apartado “lista de TODOs” podemos ver cada una de las partes del código que deberemos completar durante el transcurso de la práctica.

Práctica 3 de PAV - detección de pitch v2.0

Search

▼ Práctica 3 de PAV - detec
PAV - P3: detección de
Lista de TODOs
► Namespaces
► Classes
► Files

Member main (int argc, const char *argv[])

[TODO]:
Modify the program syntax and the call to `doctest()` in order to add options and arguments to the program.

[TODO]:
Preprocess the input signal in order to ease pitch estimation. For instance, central-clipping or low pass filtering may be used.

[TODO]:
Postprocess the estimation in order to suppress errors. For instance, a median filter or time-warping may be used.

Member upc::PitchAnalyzer::autocorrelation (const std::vector< float > &x, std::vector< float > &r) const

[TODO]:
Compute the autocorrelation `r[i]`

Member upc::PitchAnalyzer::compute_pitch (std::vector< float > &x) const

[TODO]:
Find the lag of the maximum value of the autocorrelation away from the origin.
Choices to set the minimum value of the lag are:

- The first negative value of the autocorrelation.

Generated by **doxygen** 1.8.13

Tarea 2

La tarea 2 consiste en configurar el inicio de sesión, editando el fichero `./profile` para añadir a la variable `$PATH` la ruta del directorio con el programa. De esto modo podremos acceder al programa aunque no nos situemos en directorio donde se encuentran los propios ficheros ejecutables.

Accedemos al fichero `.profile` desde el directorio home de ubuntu. A priori nos parece que no está, pero con el comando `ls -a` lo podemos ver, dado que éste nos muestra los ficheros ocultos y `.profile` lo está. Lo abrimo con el code y situamos las siguientes líneas de comandos:

```
29  PATH+=$HOME/PAV/bin
30
31  export PATH+=:$HOME/PAV/P3/bin
32
33  PATH=$PATH:$HOME/PAV/bin
```

Ahora, llamando desde el directorio P3, los programas van a correr aunque sus ejecutables esten dentro del directorio `src/get_pitch`. Comprovemoslo:

```
roger@DESKTOP-RLTD4F2:~/PAV/P3$ run_get_pitch
get_pitch pitch_db/train/r1002.wav pitch_db/train/r1002.f0 ----
get_pitch pitch_db/train/r1004.wav pitch_db/train/r1004.f0 ----
get_pitch pitch_db/train/r1006.wav pitch_db/train/r1006.f0 ----
get_pitch pitch_db/train/r1008.wav pitch_db/train/r1008.f0 ----
get_pitch pitch_db/train/r1010.wav pitch_db/train/r1010.f0 ----
get_pitch pitch_db/train/r1012.wav pitch_db/train/r1012.f0 ----
get_pitch pitch_db/train/r1014.wav pitch_db/train/r1014.f0 ----
get_pitch pitch_db/train/r1016.wav pitch_db/train/r1016.f0 ----
get_pitch pitch_db/train/r1018.wav pitch_db/train/r1018.f0 ----
get_pitch pitch_db/train/r1020.wav pitch_db/train/r1020.f0 ----
get_pitch pitch_db/train/r1022.wav pitch_db/train/r1022.f0 ----
get_pitch pitch_db/train/r1024.wav pitch_db/train/r1024.f0 ----
get_pitch pitch_db/train/r1026.wav pitch_db/train/r1026.f0 ----
get_pitch pitch_db/train/r1028.wav pitch_db/train/r1028.f0 ----
get_pitch pitch_db/train/r1030.wav pitch_db/train/r1030.f0 ----
get_pitch pitch_db/train/r1032.wav pitch_db/train/r1032.f0 ----
get_pitch pitch_db/train/r1034.wav pitch_db/train/r1034.f0 ----
```

También vemos que se ejecuta correctamente la orden `get_pitch`, dado que no nos aparece ningún error, ejecutando ésta desde `PAV/P3`:

```
roger@DESKTOP-RLTD4F2:~/PAV/P3$ get_pitch hola.wav hola.f0
```

Ejercicio 1

Primero de todo hay que calcular la autocorrelación de la señal de entrada. Por lo tanto hemos implementado el código siguiente:

```
void PitchAnalyzer::autocorrelation(const vector<float> &x,
vector<float> &r) const {
    for (unsigned int l=0; l<r.size(); ++l) {
        /// \TODO Compute the autocorrelation r[l]
        r[l] = 0;
        for(unsigned int j=l; j < x.size(); j++)
            r[l] += (x[j]*x[j-l]);
        r[l] /= x.size();
    }
    if (r[0] == 0.0F) //to avoid log() and divide zero
        r[0] = 1e-10;
}
```

Después hemos implementado el código para la ventana de Hamming:

```
void PitchAnalyzer::set_window(Window win_type) {
    if (frameLen == 0)
        return;
    window.resize(frameLen);
    switch (win_type) {
    case HAMMING:
        /// \TODO Implement the Hamming window
        for (unsigned int i=0; i<frameLen;i++){
            window[i] = 0.54836 -0.46164*cos((2*3.14159*i)/(frameLen-1));
        }
        break;
    case RECT:
    default:
        window.assign(frameLen, 1);
    }
}
```

Para determinar el pitch de la señal de voz hacemos uso del código de la autocorrelación previamente mencionado. Luego encontramos el máximo dentro de la franja que predeterminamos en `set_f0_ranfe()`;

```

float PitchAnalyzer::compute_pitch(vector<float> & x) const {
    if (x.size() != frameLen)
        return -1.0F;
    //Window input frame
    for (unsigned int i=0; i<x.size(); ++i)
        x[i] *= window[i];
    vector<float> r(npitch_max);
    //Compute correlation
    autocorrelation(x, r);
    vector<float>::const_iterator iR = r.begin(), iRMax = iR+npitch_min,
        iRref;
    for (iRref=iR+npitch_min; iRref<=iR+npitch_max; ++iRref){
        if (*iRref > *iRMax){
            iRMax = iRref;
        }
    }
    unsigned int lag = iRMax - r.begin();

```

Por último usamos una regla de decisión para poder identificar si un fragmento de la señal es sordo o sonoro. Esta parte del código tiene varios parámetros que nos ayudarán a optimizar el código más adelante y obtener mejores resultados.

```

bool PitchAnalyzer::unvoiced(float pot, float rlnorm, float rmaxnorm) const
{
    /// \TODO Implement a rule to decide whether the sound is voiced or
    not.
    /// * You can use the standard features (pot, rlnorm, rmaxnorm),
    /// or compute and use other ones.
    if(rlnorm < 0.9 || rmaxnorm < 0.2 || pot < -38){
        return true;
    }else return false;
}

```

Ejercicios de ampliación

Ahora nos dispondremos, a través de diversas técnicas que a continuación explicaremos, mejorar nuestro algoritmo de detección de pitch. Éstas técnicas las dividimos en dos grupos, en función de si se realizan antes o después del procesado de la señal que detecta el pitch en cuestión.

Preprocesado

En primer lugar, realizamos un preprocesado de la señal con la técnica center clipping. Ésta consiste en poner a valor '0' aquellas muestras cuyo valor sea inferior a un mínimo que nosotros establecemos. Con ésto conseguimos que aquellas muestras de voz pertenecientes a sonidos sordos queden eliminadas quedándonos sólo con los sonidos sonoros, que son los que determinan el pitch de la señal. Veamos cómo implementa dicha técnica nuestro algoritmo:

```
float max_x = 0.0;
unsigned int i;

for(i = 0; i<x.size();i++){
    if(x[i] > max_x)
        max_x =x[i];
}
float center_x = 0.015*max_x;

for(i=0; i<x.size(); i++){
    x[i] /= max_x;
    if (x[i] > center_x)
        x[i] -= center_x;
    else if ( x[i] < -center_x)
        x[i] += center_x;
    else x[i] = 0;
}
```

Postprocesado

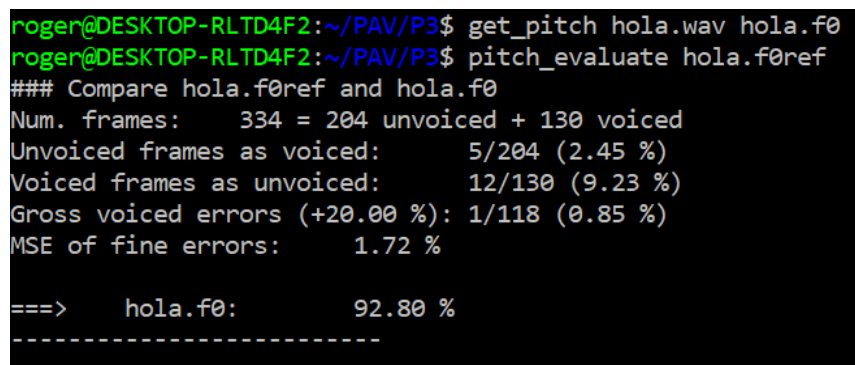
Una vez detectado el pitch de cada una de las muestras de voz sonora, aplicaremos un filtro de mediana a nuestro resultado para eliminar errores puntuales en alguna muestra y así minimizar el error total de nuestro sistema. Veámos cómo se realiza y se aplica este filtro:

```
std::vector<float> aux(f0);
unsigned int j =0;
float maximo, minimo;

for (j=2;j<aux.size() -1;++j){
    minimo = min(min(aux[j-1], aux[j]),aux[j+1]);
    maximo = max(max(aux[j-1], aux[j]),aux[j+1]);
    f0[j] = aux[j-1] + aux[j] + aux[j+1] -minimo -maximo;
}
```

Este filtro como vemos, se muestra en las muestras vecinas de la actual, es decir, en sus muestras antecesora y predecesora.

Una vez implementados nuestros algoritmos de pre y post procesado veámos cómo mejoran los resultados finales:



```
roger@DESKTOP-RLTD4F2:~/PAV/P3$ get_pitch hola.wav hola.f0
roger@DESKTOP-RLTD4F2:~/PAV/P3$ pitch_evaluate hola.f0ref
### Compare hola.f0ref and hola.f0
Num. frames:    334 = 204 unvoiced + 130 voiced
Unvoiced frames as voiced:    5/204 (2.45 %)
Voiced frames as unvoiced:    12/130 (9.23 %)
Gross voiced errors (+20.00 %): 1/118 (0.85 %)
MSE of fine errors:    1.72 %

==>    hola.f0:    92.80 %
-----
```

Conclusiones

El error cuadrático medio es de 1.72% , lo cual es un buen resultado. Nuestro algoritmo sólo ha detectado con un error mayor al 20% el pitch de una de las 118 muestras consideradas como sonidos sonoros, algo muy positivo. El total de muestras sonoras era de 130, por lo que nuestro algoritmo ha dado por sordas 12 de ellas. Del mismo modo podemos observar el error en el caso contrario, y es aún más bajo. Por todo esto, podemos concluir que nuestro algoritmo es bastante fiable.