Pitch-Shifting y otros efectos utilizando SIMD

Macarena Piaggio, Matias Grynberg Portnoy

20 de marzo de 2020

Abstract

Pitch-shifting es un efecto de sonido mediante el cual se altera el tono percibido de un audio sin alterar su velocidad. Existen diversos algoritmos en literatura para lograr este efecto[1]. Los distintos mecanismos que componen estos algoritmos presentan una oportunidad interesante para la utilización de SIMD. [2]. La implementación del Pitch-Shifter a realizar está basada en los algoritmos presentes en [3] [4] [5]. Para esto deben implementarse de manera vectorizada: el estiramiento de la señal de audio, *overlap-add*[1], la *Fast Fourier Transform* (FFT) y su inversa, y el *resample* del audio[2].

Una vez que esta maquinaria está en su lugar pueden implementarse otros efectos de sonido con facilidad: la FFT resulta eficiente para implementar un efecto de *reverb/delay* usando convoluciones[3]. Tambien puede implementarse un efecto de *vocoder*[4]. [10]

1. Introducción

En 1997 la industria musical fue revolucionada por la aparición del Auto-Tune de Antares Audio Technologies. Lo que comenzó como un procesador de audio que buscaba corregir errores de tono ligeros en las pistas vocales de los artistas inesperadamente se popularizó como unidad de efectos, con la canción *Believe* de Cher como pionera.

En esta intersección entre la computación y la música nos interesó pararnos para este trabajo. Aplicamos nuestros conocimientos de SIMD (*Single Instruction, Multiple Data*) para aprovechar al máximo los recursos del computador al momento de aplicar *Pitch-shifting*, una de las mecánicas clave que componen el funcionamiento del efecto de Auto-Tune. En este informe nos concentraremos en explorar las ventajas que puede traer el uso de instrucciones SIMD al momento de implementar esa y otras técnicas de procesamiento de audio.

Desarrollamos implementaciones en lenguaje ensamblador Intel de 64 bits de los efectos de audio *Stretch*, *Reverb*, *Vocoder* y *Resample*; este último no lo consideramos un efecto por su cuenta, pero en conjunto con *Stretch* construyen la implementación del *Pitch-Shifter*. *Stretch* permite aumentar o reducir la velocidad del audio manteniendo su tono original, *Resample* simula un cambio de samplerate del audio por una constante (que en el caso del Pitch-Shifter estará relacionada con la constante a usar en la parte de Stretch del efecto) agregando y quitando samples según necesite, *Reverb* imita la reverberación que se generaría al escuchar al audio en un espacio cerrado amplio (por ejemplo una iglesia) basado en un audio `impulse_response` y *Vocoder* simula que el sintetizador (`carrier`) está hablando como lo hace la persona del audio introducido (`modulator`). *Pitch-Shifter* permite alterar el tono del audio sin alterar su velocidad, utilizando Resample y Stretch en conjunto.

Para cada efecto realizado proveemos una implementación en C y otra en ASM que hace uso de SIMD, para poder comparar y observar las ventajas que proporcionan las optimizaciones introducidas en la segunda implementación.

[1] Basados en [4], [6] y [7]
[2] Basado en [3]
[3] Basado en [8]
[4] Basado en [9]

2. FFT

La mayor parte de los efectos a desarrollar utilizan extensivamente *Fast Fourier Transforms* (FFT), por lo que resulta crucial tener una implementación eficiente de ésta. Para esto, creamos la estructura `complejo` que contiene dos `float`, `real` e `imaginaria`. Como las operaciones de suma y multiplicación de complejos son buenos candidatos para aprovechar SIMD, tratamos de optimizar el algoritmo en base a esto. El algoritmo de FFT que implementamos es radix-2 (Algoritmo 1). (la versión implementada tiene el detalle de ahorrar crear arreglos nuevos agregando otra variable). La complejidad del algoritmo es $O(NlgN)$

Algorithm 1 fft(x) N potencia de 2

1: $N = x.size$
2: **if** $N == 1$ **then**
3: $\quad Fx[0] = x[0]$
4: **else**
5: $\quad x_{par} = (x[0], x[2]..,x[N-1])$
6: $\quad x_{impar} = (x[1], x[3]..,x[N-2])$
7: $\quad Fx_{par} = fft(x_{par})$
8: $\quad Fx_{impar} = fft(x_{impar})$
9: $\quad$ **for** $j = 0$ **to** $N/2$ **do**
10: $\quad\quad rotacion = e^{-2\pi i \cdot j/N}$
11: $\quad\quad Fx[j] = Fx_{par}[j] + Fx_{impar}[j] \cdot rotacion$
12: $\quad\quad Fx[j + N/2] = Fx_{par}[j] - Fx_{impar}[j] \cdot rotacion$
13: $\quad$ **end for**
14: **end if**

La estrategia consiste en realizar 2 iteraciones del `for` en simultáneo. `rotación` requiere calcular seno y coseno para cada iteración, lo cual es costoso: para contrarrestar esto barajamos mantener $e^{-2\pi i/N}$ en algún registro y obtener la segunda rotación a partir de una multiplicación compleja con la primer rotación. Esto no es muy eficiente; la multiplicación compleja requiere varias operaciones y además genera error numérico considerable.

Otro problema aparece en los llamados a `cos` y `sin`: ahorrar llamados innecesarios es crucial. Nuestra solución es mantener un arreglo `rotaciones` de tamaño fijo N/2 que contenga todas las rotaciones que puedan necesitarse, lo cual podemos hacer ya que la única parte variable de su ecuación es el valor j que vale entre 0 y N/2-1. Esto tiene como consecuencia limitar la FFT a tamaños menores al doble de rotaciones precalculadas. Tomamos 1024 rotaciones precalculadas (8Kb), limitando la FFT hasta 2048. En la versión de C, como no estaba esta limitación, calculamos `cos` y `sin` cada vez, permitiendo FFT de tamaño arbitrario. En la figura 1 variamos en tamaño del arreglo a transformar de 4 a 2048, observamos la performance tanto para asm como para C con distintos flags de compilación.

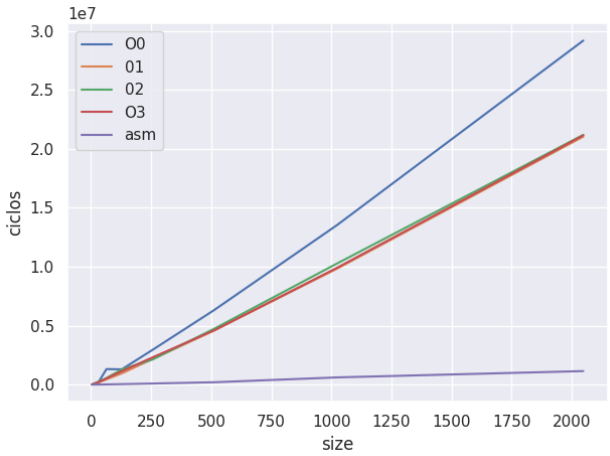


Figura 1: Comparación de ciclos insumidos para la FFT en C y asm

3. Efectos

3.1. Resample

Un audio está compuesto de un arreglo de valores que representan cada unidad de tiempo y de un samplerate que indica cuantas de estas unidades se hallan en un segundo de tiempo real. El samplerate estándar es de 44100 Hz. Si uno aumenta el samplerate, se recorrerán los elementos del audio mas rápidamente, resultando en una menor duración y un sonido mas agudo (como se altera la proporción de tiempo que representa cada unidad del arreglo, los períodos disminuyen de manera acorde). De manera análoga al reducir el samplerate la duración del audio aumenta y se produce un sonido mas grave. Para realizar el Pitch Shifter se debe implementar primero resample, este fenómeno que se da al alterar el samplerate. Como el samplerate esta estandarizado en 44100 Hz, no es buena idea cambiar esta constante. Lo que haremos sera mantener el samplerate fijo y escalar el audio para que tenga la cantidad de elementos correctos utilizando interpolación lineal, efectivamente simulando el efecto que se generaría si cambiaramos el samplerate.

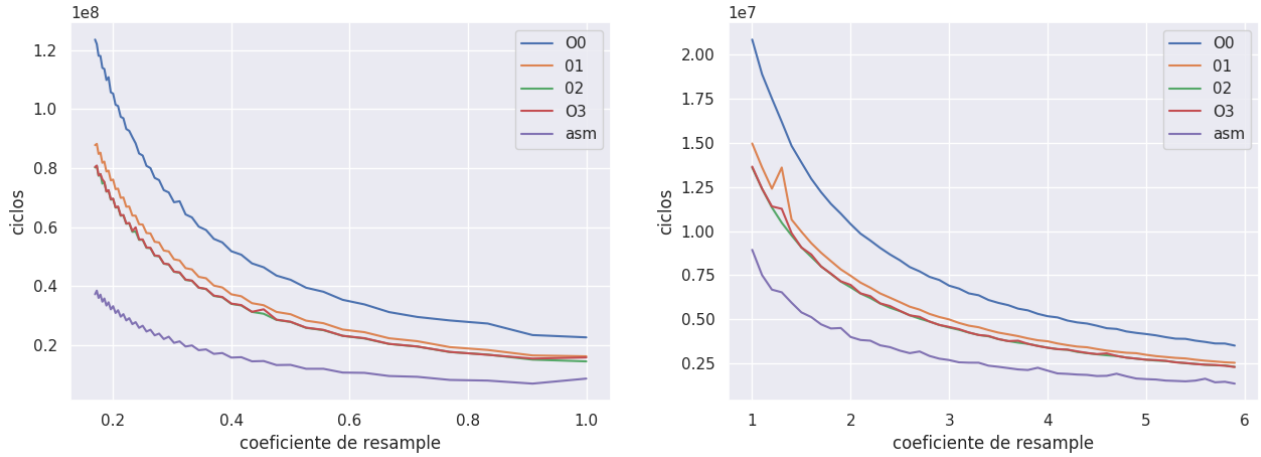
La función `resample` toma `float f`, el factor por el que se multiplicaría el `samplerate`. Un `f = 2`, por ejemplo, resultaría en un arreglo de la mitad del tamaño original. Para $f < 1$ el nuevo arreglo será de mayor tamaño que el original.

Pensar como implementar interpolación vectorizada resulto un desafió. La solución que proponemos realiza una optimización simple: en cada iteración se cargan en registros los valores del arreglo para la interpolación y se realizan dos interpolaciones en simultaneo.

```
movlhps xmm1, xmm4  ; xmm1 = | Arr[b1] | Arr[a1] | Arr[b0] | Arr[a0] |
movlhps xmm2, xmm3  ; xmm2 = | 0  | x1 | 0  | x0 |  ; con x = i*f

movlhps xmm5, xmm6
movdqa xmm3, xmm5 ; xmm3 = | 0 | a1 | 0  | a0 |
subps xmm2, xmm3 ; xmm2 = | 0 | x1-a1 | 0  | x0-a0 |
pshufd xmm2, xmm2, 1000b ; xmm2 = | .. | .. | x1-a1 | x0-a0 |

pshufd xmm3, xmm1, 10110001b
hsubps xmm3, xmm3 ; xmm3 = |..|..| Arr[b1] - Arr[a1] | Arr[b0] - Arr[a0] |
mulps xmm3, xmm2 ; xmm3  = |..|..| Arr[b1] - Arr[a1] * (x1-a1) | Arr[b0] - Arr[a0]  * (x0-a0)|

pshufd xmm1, xmm1, 1000b
addps xmm3, xmm1
; ; xmm3 |..|..|Arr[b1] - Arr[a1] * (x1-a1) + Arr[a1]| Arr[b0] - Arr[a0]  * (x0-a0) + Arr[a0]|
```

Hay otra optimización que no logramos resolver. Por mas que en cada iteración se cargan los elementos necesarios para la interpolación desde la memoria, esto no es siempre necesario. En algunas iteraciones sucederá que parte de los elementos con los cuales se realiza la interpolación lineal actual coinciden con los elementos de las anteriores iteraciones. Este problema será mas pronunciado a medida que `f` se acerca a 0, ya que se genera un arreglo de mayor tamaño, haciéndose varias interpolaciones con los mismos elementos. Como no había una manera obvia de solucionar esto, decidimos ignorarlo.

Comparamos los ciclos insumidos variando `f` en la versión en C y en ASM. Como ya mencionamos, hay dos situaciones: compresión para $f > 1$ y descompresión para $f < 1$. Analizamos cada uno por separado dejando el tamaño del arreglo para el resample fijo en $250 \cdot 1024$. Para compresión variamos `f` de 1 a 6 aumentando de a 0.1, es decir hasta una compresión de 6 veces. Para descompresión aumentamos $\frac{1}{f}$ de 1 a 6, aumentando de a 0.1, lo que equivale a una descompresión de 6 veces. Los resultado reflejan que la cantidad de ciclos aumenta con el numero de interpolaciones y por lo tanto aumenta asintóticamente a medida que `f` se acerca a 0. La versión en ASM parece crecer notablemente mas lento.



(a) Descompresión

(b) Compresión

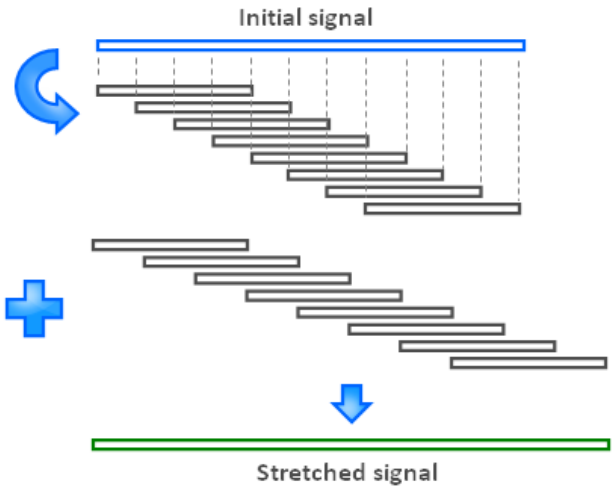
Figura 2: Comparación de ciclos insumidos para resample en C y ASM

3.2. Stretch

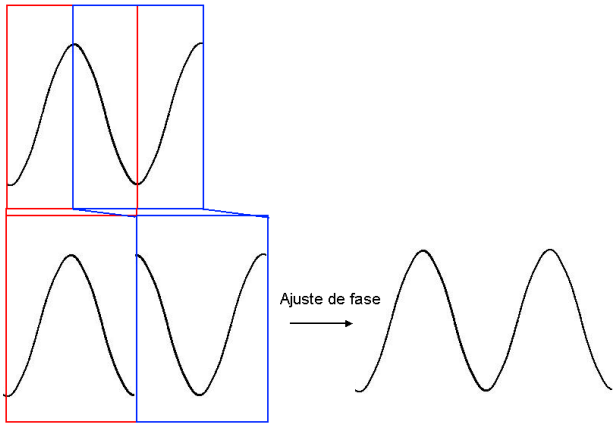
La segunda transformación a implementar para el Pitch Shifter es stretch. Si tenemos un audio con un samplerate dado, por ejemplo 44kHz, y lo reproducimos doblando este samplerate (es decir, doblando la cantidad de samples reproducidas por unidad de tiempo), estaremos efectivamente contrayendo el audio en términos de tiempo. Sin embargo, como ya se vio, esto tiene el efecto de alterar el tono del audio inicial, en este caso aumentándolo. Para poder implementar el Pitch Shifter deseado es necesario implementar una transformación stretch tal que el audio final sea estirado o contraído de la manera deseada, pero mantenga el tono del audio original.

A grandes rasgos, el algoritmo a seguir es el siguiente:

- Se divide al audio en "ventanas" de tamaño `window_size` con cierto solapamiento entre ellas.
- Se reposiciona estos bloques de manera que, conceptualmente, si los bloques se encontraban a distancia d, ahora estarán a distancia $d/\texttt{f}$, dando el estiramiento o contracción indicada. (figura 3a)
- Antes de sintetizar el audio, se corrige la fase de cada una de las frecuencias. Mas técnicamente, se aplica FFT en cada bloque y se estima el cambio de fase entre los bloques contiguos para cada frecuencia. Con esto se alteran las distintas fases del bloque para que se correspondan con la nueva posición de este y con el cambio de fase para esa determinada frecuencia. (figura 3b) (el algoritmo en realidad agrega una corrección al cambio de fase dada por el cambio de fase con el bloque anterior)
- Luego se realizan las IFFT y se van sumando los resultados (*overlap-add*).

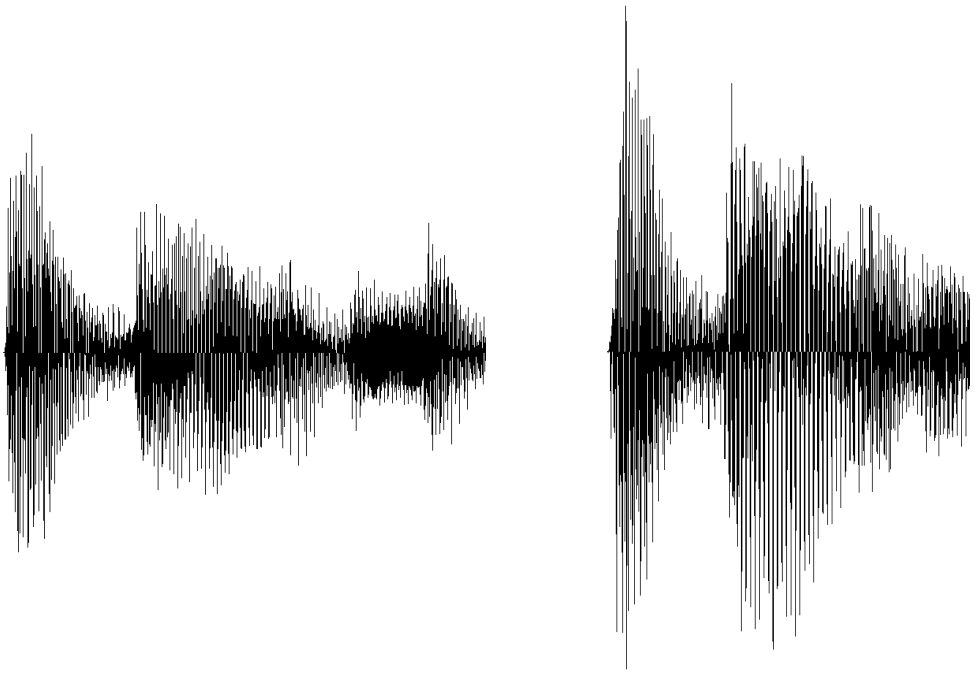


(a) Reposicionamiento de los bloques



(b) Ajuste de fase para una frecuencia

Figura 3: Ilustraciones sobre el comportamiento de Stretch



(a) Waveform de un audio antes de aplicar stretch

(b) Waveform del mismo audio después de aplicar stretch

La cantidad de frames que separan el comienzo de una ventana y el comienzo de la siguiente está dada por `hop`. Un hop más pequeño significara más solapamiento y por lo tanto más ventanas.

Uno de los detalles técnicos de la implementación es que al tomar los bloques de audio, no se los deja simplemente como un corte del audio original sino que se multiplica los elementos por una función, la *window function*. Existen diversas opciones para esta: la elegida, tomada de bibliografía[4], es la función de *hanning*

$$w[n] = \frac{sin(\pi \cdot n)}{N-1}^2 \text{ con N la longitud de la ventana}$$

En cada iteración, al tomar un bloque, se multiplica el primer elemento por $w[0]$, el segundo por $w[1]$, etc. De manera análoga, durante la resíntesis se aplica nuevamente la función de hanning. Utilizar ventanas rectangulares (no utilizar una *window function*) introduce discontinuidades en el sonido, especialmente en conjunto con el método overlap-add, por lo que hacemos uso de la función de hanning para darle una forma a las ventanas que mitigue estos efectos, de acuerdo a los resultados de los trabajos de investigación consultados. Como la fórmula de Hanning sólo depende del tamaño de la ventana se pueden precalcular todos los valores que vamos a necesitar (en ASM, esto se realiza en el ciclo `init_hanning`).

Las optimizaciones realizadas por nuestro algoritmo de stretch son las siguientes:

- Se procesa, de la misma manera que en C, de a una ventana de audio. Para procesar una ventana de audio es necesario tener también su anterior, por lo que se trabaja con dos arreglos de tamaño `window_size` en cada iteración del ciclo, `a2` la i-ésima ventana y `a1` la i-1-ésima. En ambas implementaciones aprovechamos que en la iteración anterior ya trajimos a memoria una de las dos ventanas de audio a usar, por lo que al final de cada ciclo intercambiamos las direcciones de los punteros a los arreglos `a2` y `a1`, y en el siguiente ciclo mantenemos el arreglo `a1` (que es el `a2` del ciclo anterior) y sobreescribimos el espacio del arreglo `a2` con el audio de la nueva ventana a usar.
- En el ciclo en que se multiplican las ventanas de audio rectangulares por los valores de hanning para cambiar la forma de las ventanas (`ciclo_a`), se procesan cuatro floats por vez en lugar de uno como en

la versión de C. Se levantan ocho valores del arreglo `audio` a dos registros XMM, cuatro de la ventana i-ésima y cuatro de la ventana i-1-ésima, que se encuentran en la misma posición relativa dentro de su ventana. Se multiplica a dichos registros por los cuatro del arreglo `hanning` correspondientes, que son los mismos para ambos registros ya que el valor de la función de hanning que se usa depende de la posición relativa en la ventana de los valores traídos a memoria.

- En el ciclo en que se corrigen las fases de las frecuencias (`ciclo_3`) se trabaja con dos elementos de `s2` (y sus correspondientes de `s1`) a la vez. Recordemos que `s2` se corresponde a la ventana i-ésima de audio y `s1` se corresponde a la ventana i-1-ésima, luego de aplicarles FFT. C trabaja de a un elemento de estos arreglos.

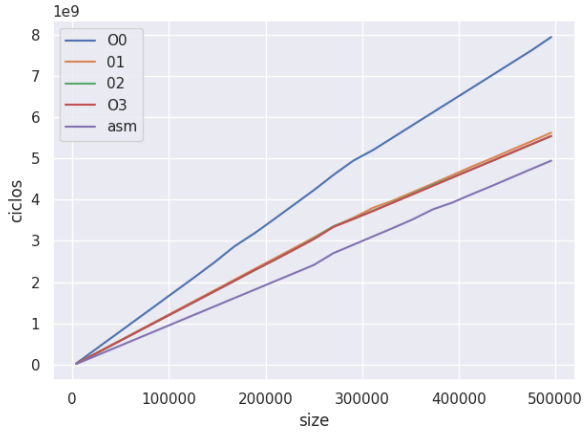
- La bajada a memoria de los datos calculados, presentes en `s1`, se hace de a cuatro elementos a la vez, en lugar de a uno.

Se puede observar el efecto de estos cambios en la cantidad de ciclos insumidos por el programa en las figuras 5a, 5b y 5c.

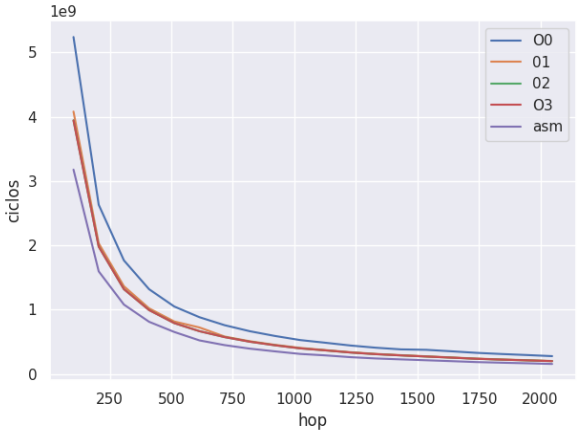
Al tener *stretch* cuatro variables de entrada y ser un efecto computacionalmente costoso, decidimos experimentar con distintos valores de las variables de a una, dejando fijas a las demás. Por esto, los resultados obtenidos son parciales, es decir, las ventajas o desventajas observadas de la implementación no representan todas las posibles combinaciones de los parámetros. Aun así, es natural, dada la forma del algoritmo, pensar que los resultados parciales reflejan el comportamiento general de la implementación. Los resultados muestran que la performance de ASM es notablemente superior a la obtenida por el compilador de C, especialmente para audios más largos. Todos los gráficos incluidos presentan el promedio de los resultados obtenidos de realizar diez corridas independientes de cada experimento, utilizando como audio de entrada un arreglo de valores arbitrarios `float audio[n]`.

La figura 5a muestra que los ciclos insumidos escalan linealmente con el largo del audio. La dependencia lineal es intuitiva ya que el costo es proporcional a la cantidad de ventanas, entonces duplicar el audio duplicará la cantidad de ventanas. El gráfico de resultados nos permite concluir que para entradas grandes la implementación en ASM es fuertemente preferible. La figura 5b evidencia lo mismo, ya que la cantidad de ventanas es inversamente proporcional a `hop`: duplicar el `hop` reduce la cantidad de ventanas a la mitad. Para `hop` pequeños, donde el número de ventanas es considerable, se muestra una reducción de alrededor del 25 % con respecto a los ciclos insumidos por las versiones en C más optimizadas.

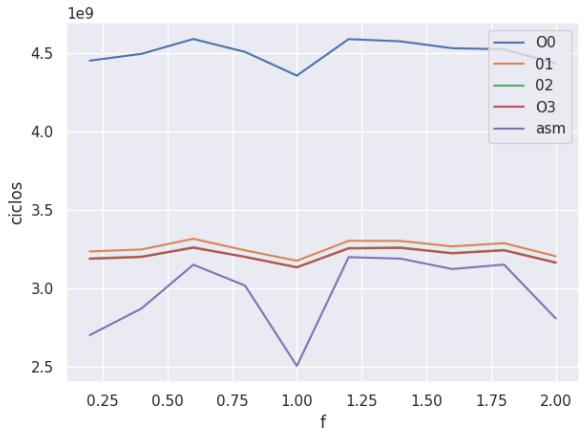
La figura 5c presenta un resultado inesperado. El algoritmo es prácticamente independiente de `f`; solo es usado para algunas pocas multiplicaciones y divisiones. Uno entonces estaría inclinado a pensar que el valor especifico de `f` no tendría efecto en esta métrica. Los resultados no terminan de confirmar esta hipótesis: para `f` = 1 todas las implementaciones mostraron mayor eficiencia (este decaimiento es particularmente notorio en la implementación en ASM). Además, al tener todas la misma forma general, no se puede descartar el resultado como ruido. No incluimos ninguna guarda para evitar el caso f=1, que implica no hacerle ninguna modificación al audio, por lo que los resultados de ASM resultan particularmente sorprendentes. Estas observaciones nos llevan a creer que el costo de la instrucción de división depende del divisor, es decir que la operación de división resulta mas rápida para algunos `f`, pero no tenemos evidencia de esto ya que Intel no hace pública mucha información respecto a la implementación de sus instrucciones. Un experimento que podría realizarse para apoyar esta teoría sería realizar varias divisiones con distintos valores y tomar un promedio de los ciclos de reloj. Habiendo mencionado esto, dejamos este resultado abierto a mayor investigación sin encontrar una justificación adecuada a lo obtenido, ya que esta problemática está fuera del alcance de este trabajo.



(a) Variando el tamaño del audio. `f` = 0.5, `window_size` = 2048, `hop` = 128



(b) Variando el hop. tamaño del audio = 250*1024, `f` = 0.5, `window_size` = 2048



(c) Variando f. tamaño del audio = 250*1024, `window_size` = 2048, `hop` = 128

Figura 5: Comparación de ciclos insumidos para stretch en C y asm

3.3. Pitch Shifter

El procedimiento para el pitch shifter consiste simplemente en combinar el uso de *stretch* y *resample*. Para hacer al audio más agudo lo que se hace es dilatar al audio sin alterar su tono utilizando *stretch*, y luego se contrae con resample, devolviéndolo a su tamaño original pero con el tono efectivamente alterado. Entonces, si se multiplica a las frecuencias por `f` (se llama a `repitch` con `f` algún valor distinto positivo), la función procede a llamar a `stretch(audio,1/f)` y con el audio resultante realizar un resample de la forma `resample(streched_audio,f)`,

Como este algoritmo es simplemente dos llamados a funciones anteriores, su performance se desprende de los resultados previos. Los ciclos insumidos por *pitch shifter* van a ser la suma de los insumidos por *stretch* y *resample*. Es relevante mencionar que *stetch* es un procedimiento mucho mas costoso que *resample* y por lo tanto *stretch* es el paso determinante del algoritmo. En otras palabras, el *pitch shifter* tiene un consumo de ciclos de reloj muy similar al de *stretch*.

3.4. Reverberación

Dado un audio, supongamos que se quiere simular el efecto de realizar ese sonido en, por ejemplo, una iglesia. Producir un sonido en una iglesia genera una respuesta particular, la reverberación de la iglesia: el objetivo es lograr un efecto similar artificialmente.

Una forma relativamente simple de hacer esto es agregar copias con cierto delay al audio original para que actúen como una serie de ecos. Es decir, cada tantas unidades de tiempo se replica el audio completo. Pueden multiplicarse estas copias por constantes para asemejarlas a replicas que van perdiendo fuerza. El problema con este método es que da resultados bastante irreales, en los ambientes físicos generalmente no se escuchan replicas en intervalos precisos si no que se da un decaimiento más continuo. Lo que puede hacerse es agregar copias en todas las unidades de tiempo; esto quiere decir que en cada parte del audio se escuchan replicas

de todo lo sucedido anteriormente (en la practica, los ecos se extinguen luego de cierto tiempo, asi que serán replicas de todo lo sucedido en un intervalo). En concreto, tenemos un arreglo `ir` (por lo general se llama a este *impulse response*) donde cada entrada representa la fuerza para el delay con ese índice. Matemáticamente, lo que haremos se conoce como la *convolución lineal* de *audio* y *ir*

$$out[i] = \sum_{j=0}^{min(i,ir.size)} audio[i-j] \cdot ir[j]$$

Una primera implementación podría ser la siguiente.

Algorithm 2 ConvLineal-Directo($audio, ir$)

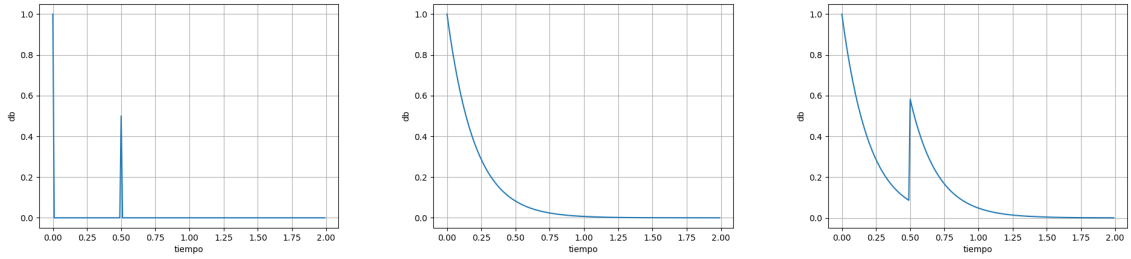
```
1: out[0..audio.size + ir.size − 1] = 0
2: for i = 0 to audio.size do
3:     for j = 0 to ir.size do
4:         out[i + j] = out[i + j] + audio[i] · ir[j]
5:     end for
6: end for
```

Este algoritmo puede implementarse usando SIMD de manera simple. La función `convolucion_lineal_directa` realiza 4 iteraciones de cada ciclo por medio de `pshufd` y `mulps`

Sobre lo desarrollado hasta ahora surgen 2 problemas.

- Una serie de ecos elegida arbitrariamente puede no reflejar el comportamiento del verdadero ambiente a imitar; es necesario algo mas de información.
- Este algoritmo, si se tiene un audio de N muestras, introduce M ecos de la manera directa (algoritmo 2), lo cual requiere $N * M$ sumas y multiplicaciones. Esto puede resultar demasiado lento para M grande, incluso optimizando la performance en ASM.

La solución al primer problema es la utilización de un *impulse response* real. Tomamos un audio del ambiente a imitar donde lo que se escucha es la reverberación producida por un pulso inicial, idealmente de mínima duración. Con esto, se realiza la convolución lineal. La idea intuitiva es que cada entrada del audio actúa como un pulso, generando la misma respuesta escalada por la magnitud de este pulso particular. La reverberación es simplemente la suma de cada una de estas respuestas (figura 6).



(a) Audio de entrada (b) Impulse response (c) Convolución lineal

Figura 6: Intuición detrás de la convolución lineal

La solución al segundo problema es mas técnica, introducimos un concepto mas. Supongamos que tenemos dos secuencias a y b de largo N. Se define la convolución circular: a diferencia de la convolución lineal, los ecos del final no extienden el arreglo sino que se aplican sobre el principio de este.

$$out[i] = \sum_{j=0}^{N} a[(i-j) \pmod N] \cdot b[j]$$

Similar a la convolución lineal, podemos implementarla directamente de la siguiente forma:

Algorithm 3 ConvCircular-Directo(a, b)

1: $out[0..N] = 0$
2: **for** $i = 0$ **to** N **do**
3: **for** $j = 0$ **to** N **do**
4: $out[i + j \pmod N] = out[i + j \pmod N] + a[i] \cdot b[j]$
5: **end for**
6: **end for**

Hay una propiedad matemática para mejorar la complejidad. Por el teorema de la convolución, la transformada de la convolución circular es la multiplicación de las transformadas.

$$F(convCircular(a, b))[i] = F(a)[i] \cdot F(b)[i]$$

Como realizar FFT es $O(N \cdot lgN)$, el nuevo algoritmo consiste en transformar ambos arreglos, realizar la multiplicación y antitransformar (Algoritmo 4).

Algorithm 4 ConvCircular(a, b)

1: $Fa = fft(a)$
2: $Fa = fft(b)$
3: **for** $i = 0$ **to** N **do**
4: $Fout[i] = Fa[i] \cdot Fb[i]$
5: **end for**
6: $out = ifft(Fout)$

Para realizar la convolución lineal puede usarse la convolución circular si se extiende el arreglo con ceros de manera que el final no afecte el principio del arreglo. La implementación `fft_asm` esta limitada a un rango muy pequeño de tamaños por lo que no puede realizarse una convolución circular que utilice ambos arreglos completamente. El algoritmo propuesto consiste en realizar la convolución lineal por bloques utilizando una serie de convoluciones circulares con la mayor FFT posible; sumándolos de manera coherente puede obtenerse el resultado buscado. Implementamos `convolución_circular_asm` utilizando SIMD. En cada ciclo se realizan 4 multiplicaciones complejas en simultáneo.

Esta es la descripción de como realizamos la convolución lineal, que es lo que se necesita para reverberación. Lo asombroso de este efecto es la riqueza de sonidos que se pueden obtener. Distintos ambientes tienen su *impulse response* que afecta de manera diferente al audio de entrada. También pueden usarse otras modificaciones mas extremas. Por ejemplo, al utilizar una *impulse response* invertida en el tiempo, la convolución genera un sonido con efectos de ecos previos al audio, imitando que el audio también estuviera siendo reproducido en reversa. La canción *Feel Flows* usa este efecto.

Incluimos un archivo con algunas *impulse response*. En la práctica, el algoritmo es lento, aún con las optimizaciones. Toma un tiempo aceptable para audios de alrededor de 1 minuto e *impulse response* de unos pocos segundos.

Para analizar esta lentitud, variamos el tamaño de la entrada de 1024 a $50 * 1024$ tomando el promedio de 5 iteraciones (figura 7). Estos tamaños están bastante alejados de los tamaños usuales para lo que seria un audio normal (para referencia, tomando un samplerate de 44100, un audio de 30 segundos tiene 1292 bloques de 1024 entradas cada uno). El procedimiento resultaba excesivamente lento por lo que nos redujimos a estas muestras mucho más pequeñas. Aun así, es razonable esperar que el comportamiento a mayor tamaño se asemeje a este.

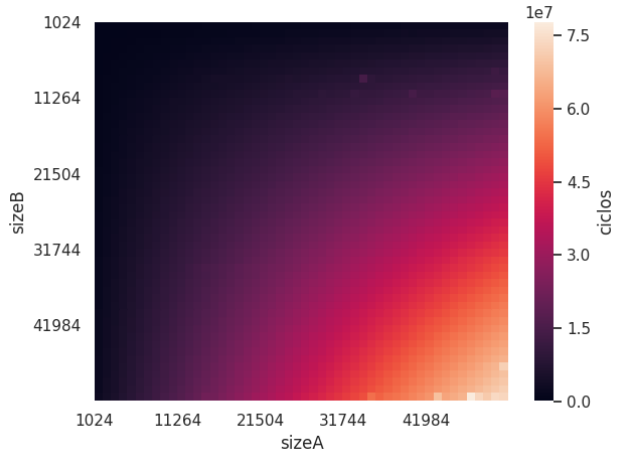


Figura 7: Comparación de los ciclos insumidos al variar el tamaño de las entradas para convolución por bloques.

Este resultado tiene varios aspectos interesantes. En primer lugar, es simétrico con respecto a ambas entradas. A primera vista esto es extraño, ya que el algoritmo trata de manera diferente a ambas entradas. Por la forma de la figura, la complejidad parece de $sizeA \cdot sizeB$; observando los detalles del algoritmo puede concluirse que realizar la convolución por bloques efectivamente tiene esta complejidad. Más aún, intercambiar ambas entradas resulta en un procedimiento casi idéntico (uno puede usar mas memoria que el otro ya que requiere mantener la FFT de todos sus bloques al mismo tiempo).

Esto explica que al aumentar levemente el tamaño de la *impulse response* el tiempo requerido se vea drásticamente afectado. Como se supone que $audio.size >> ir.size$, añadir un bloque a la ir tiene un fuerte impacto, ya que debe aplicarse este bloque a todos los de $audio$, lo cual tiene un costo $O(audio.size)$.

El método propuesto termina teniendo la misma complejidad que realizar el algoritmo directo. Surge la pregunta ¿Es este algoritmo realmente más eficiente? Para verificar esto, comparamos los ciclos insumidos por ambos algoritmos con distintos tamaños de entrada. Dejando $sizeB$ fijo en $10 \cdot 1024$, en la figura 8 se muestra los resultados variando $sizeA$ de 1024 a $50 * 1024$.

En primer lugar, se observa claramente la linealidad de ambos métodos. La constante de la convolución por bloques resulta mucho menor que la directa, por lo que concluimos que la convolución por bloques resulta mas eficiente. Un desafio interesante sería investigar por qué se da esta relación entre las constantes.

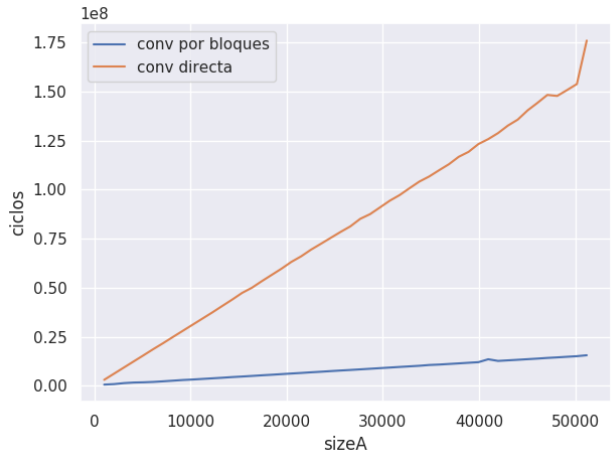
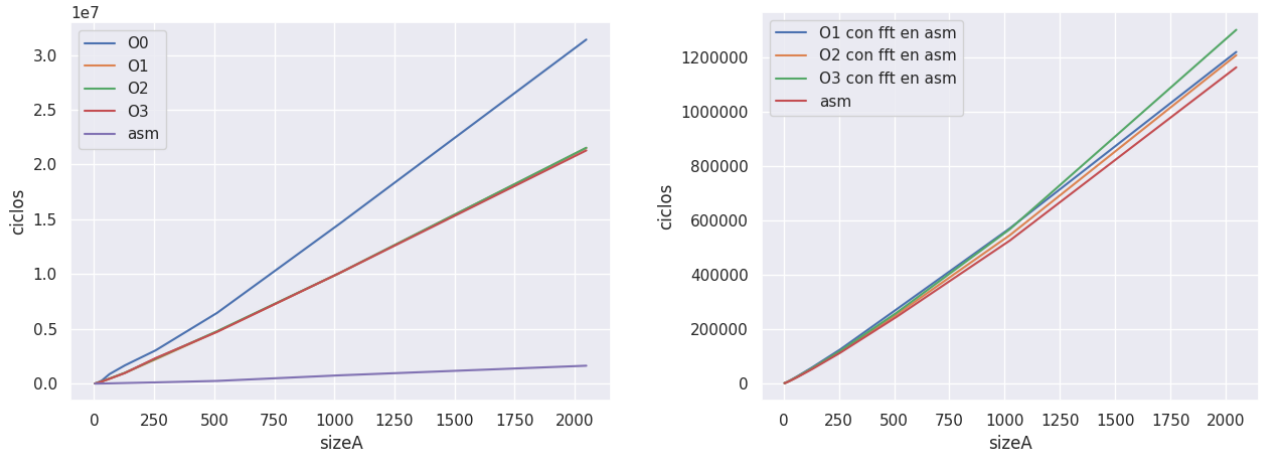


Figura 8: Comparación de performance con convolución directa.

Por otra parte, también implementamos `convolución_circular` en C para poder explorar qué ventajas se obtuvieron en la versión en ASM. Un detalle es que realizar la FFT probablemente consuma la mayor parte de los ciclos. Al ser que esto es una función externa a la rutina, lo propio a la rutina no sera lo determinante en la performance. Además, se tienen a su vez 2 versiones de fft, en ASM y en C, y no es claro cual comparación seria la mas adecuada. Por ello decidimos realizar el experimento modificando la convolución circular en C para que use ambas versiones (figura 9).



(a) Utilizando FFT de C

(b) Utilizando FFT de ASM

Figura 9: Comparación de ciclos insumidos para la convolución circular en C y ASM

Usando la version de FFT C, los resultados son prácticamente iguales a la comparación entre versiones de FFT en la figura 1. Al ser la diferencia entre las FFT es tan extrema, no logra verse que posibles mejoras se obtuvo en SIMD. Como era de esperarse, el intercambiar la FFT de C por la de ASM elimino la ventaja de la convolución circular en ASM. Aun así, esta última consume levemente menos ciclos por lo que la optimización es ligeramente superior a la lograda por el compilador (esto, de nuevo, se explica teniendo en cuenta que la parte optimizada no consume la mayor parte de los ciclos). Aumentar el flag de compilación de O0 afectó el desempeño de C, pero tuvo menor impacto para otros flags.

3.5. Vocoder

El vocoder es un efecto que utiliza dos entradas, *carrier* y *modulator*. Usualmente el *modulator* es una voz y el *carrier* un sintetizador. El resultado asemeja al sintetizador hablando. Este efecto es bastante común en musica: la canción *Mr. Roboto* de Styx es un ejemplo clásico.

El algoritmo (simplificado) consiste en imponer las magnitudes para las distintas frecuencias del *modulator* al *carrier* (figura 10). Las verdaderas implementaciones son mas complejas que la que realizaremos, separan la FFT en bandas de distintos tamaños, y multiplican estas por el mismo valor. También sucede que en algunas circunstancias intercambian el *carrier* por ruido blanco.

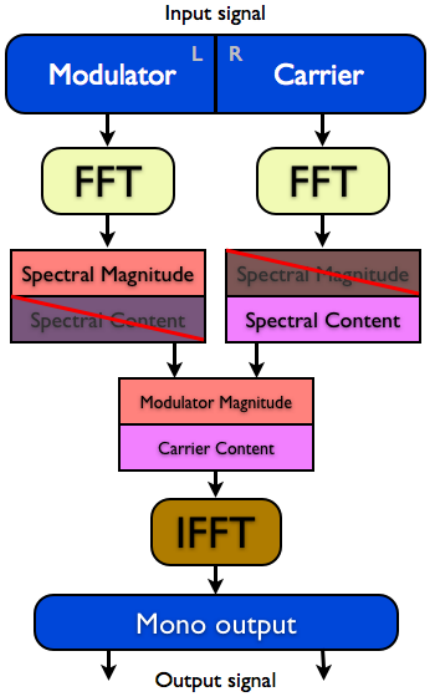


Figura 10: Proceso del vocoder

Utilizamos el *vocoder* más como una exploración de las mejoras en SIMD que como una implementación realistica que obtenga el efecto de audio esperado. **El efecto de esta versión esta lejos de asemejarse a lo deseado**. Aun así, resulta útil como primer enfoque para una verdadera versión de un vocoder. Nuestra versión simplemente multiplica cada elemento de la FFT del bloque del *carrier* por la magnitud correspondiente para el del *modulator*. Luego realizamos la IFFT y se suma al resultado después de multiplicar por `hanning`. Utilizamos un overlap de 50 %.

Algorithm 5 Vocoder($modulator, carrier, window\_size$)

$out[0..modulator.size] = 0$
$i = 0$
while $i < modulator.size$ **do**
  $Fm = fft(modulator[i : i + window\_size]$
  $Fc = fft(carrier[i : i + window\_size]$
  for $j = 0$ **to** $window\_size$ **do**
    $Fvoc[j] = |Fm[j]| \cdot Fc[j]$
  end for
  $voc = ifft(Fvoc)$
  for $j = 0$ **to** $window\_size$ **do**
    $out[i + j] = out[i + j] + hanning[j] \cdot voc[j]$
  end for
  $i = i + window\_size/2$
end while

Para optimizar la performance, lo relevante es reducir el tiempo del cuerpo de cada iteración. Como los llamados a FFT son independientes de las posibles mejoras que se puedan hacer, lo que buscamos es aprovechar SIMD en los `for` internos. Con esto en cuenta, implementamos el vocoder tanto en ASM como en C.

Para verificar que mejoras fueron obtenidas planteamos primero un argumento teórico en base a la complejidad algorítmica. Tomando N el tamaño del *modulator* y W el tamaño de la ventana, se realizan $O(N/W)$ iteraciones. Dentro de cada iteración lo que domina la complejidad son las FFT, por esto cada iteración es $O(W \cdot lgW)$. Entonces, multiplicando por la cantidad de iteraciones, la complejidad del vocoder es $O(N \cdot lgW)$. Como en reverb las optimizaciones en ASM se verán opacadas por FFT; peor aun, este algoritmo realiza tres transformadas por iteración, por lo suponemos que el efecto sera más pronunciado. Para poder hablar de las mejoras exclusivas a la implementación separemos el trabajo que se hace en cada iteración excluyendo las FFT, es $O(W)$, digamos $K \cdot W$. Agregando esta variable, repetimos el argumento sobre la complejidad del vocoder.

$$\text{ciclos por iteración} = \text{ciclos por fft} + \text{resto de iteración} = cte_1 \cdot W \cdot lgW + K \cdot W$$

$$\text{ciclos totales} = \text{cantidad de iteraciones} \cdot \text{ciclos por iteración} = cte_2 \cdot N/W \cdot (cte_1 \cdot W \cdot lgW + K \cdot W)$$

Agrupando constantes y redefiniendo K como K/cte_1 queda

$$\text{ciclos totales} = cte \cdot N \cdot (lgW + K)$$

Esta es nuestra estimación de los ciclos insumidos. Lo que queremos observar por lo tanto es esta variable K. Una mejor optimización tendrá un menor K. Con este argumento en mente, experimentamos con los ciclos insumidos variando el tamaño del *modulator* y los flag de compilación de C. Utilizamos la fft de asm en esta comparación. Fijamos `window_size` en 1024 (figura 11).

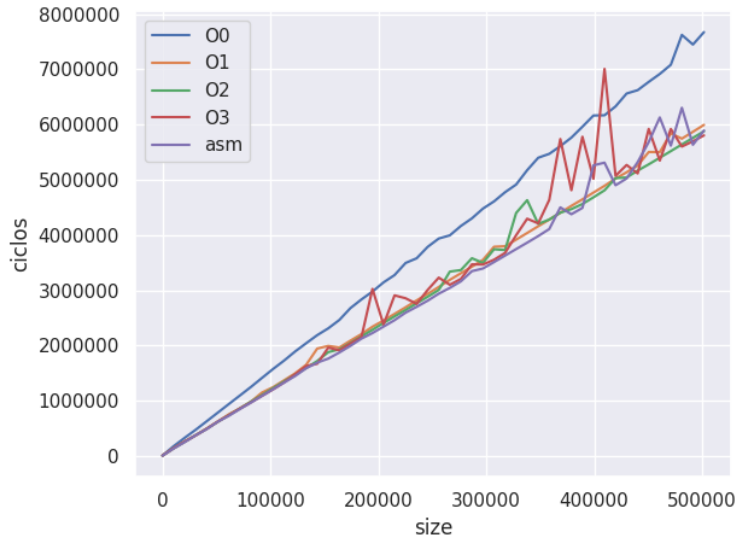


Figura 11: Comparación de los ciclos insumidos al variar el tamaño del *modulator* en vocoder. Fft de asm y `window_size` = 1024

El resultado muestra que O0 consume notablemente mas ciclos aunque utiliza la misma FFT. Esto significa que aunque FFT domine las iteraciones, el resto del procedimiento tiene cierta influencia en la performance. Mayores flags de compilación sí alcanzaron la performance de ASM. Al tener aproximadamente la misma pendiente, las K deben estar bastante mas cerca, no puede distinguirse cuál optimización es mejor entre estas. Esto significa en la práctica que las mejoras obtenidas por la implementación en ASM no son suficientemente grandes como para que el tiempo de cómputo se vea afectado y por lo tanto no se justifica el trabajo de programar en ASM en contraposición a C utilizando `gcc` con flags de optimización. Aún así, resulta interesante explorar los K. Una interpretación de que las pendientes sean similares es que como el K es pequeño para las mejores versiones, su contribución a la pendiente no es significativa por lo que las rectas son esencialmente paralelas, que es lo que se ve.

Otra forma de visualizar K es variando `window_size`. Cuando este sea pequeño la FFT será menos costosa, dejando más en evidencia los costos de otras partes del procedimiento. Dicho de otra forma, para `window_size` pequeñas, el resto cobra importancia. Dejamos el *modulator* fijo en tamaño 250∗1024 y variamos las `window_size` de 4 a 2048. La figura 12 muestra justamente esto, la versión en ASM empieza con una pequeña ventaja relativa, pero la pierde rápidamente a medida que crece el `window_size`. Por ultimo, se evidencia la complejidad logarítmica en W que ya mencionamos. Siendo más exactos, este resultado coincide muy bien con el análisis teórico previo, al mantener N fijo la contribución de K actúa como una constante sumando $cte \cdot N \cdot K$. Puede verse claramente las distintas constantes K para cada version y asm obtiene la menor, que era lo buscado.

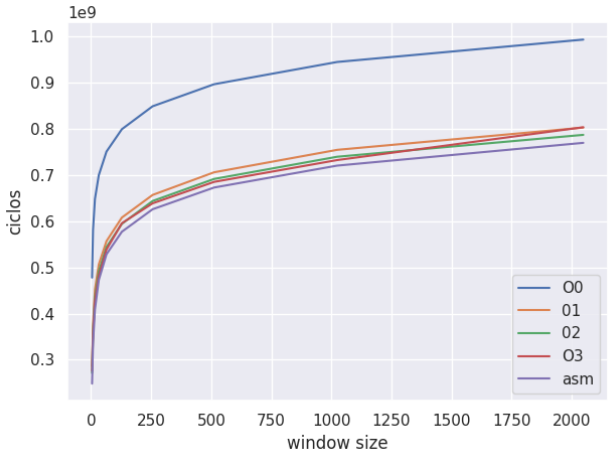


Figura 12: Comparación de los ciclos insumidos al variar el `window_size` entre 4 y 2048 en vocoder. FFT de ASM y $modulator = 250 \cdot 1024$

4. Modo de Uso

Para poder compilar el código provisto incluimos un archivo Makefile. El código que proveemos hace uso de la librería `sndfile`[5], para instalarla se puede usar el comando `make install` en el directorio raíz. Estando ya instalada la librería se puede compilar el trabajo utilizando el comando `make`. Se incluye el archivo `ejemplos.sh` que descarga la librería, compila el trabajo y crea ejemplos para cada uno de los filtros.

Una vez compilado el programa se ejecuta con el comando `./main` desde el directorio raíz. **Todos los audios** que se utilicen como parámetros deben ser de tipo mono y extensión `.wav`. Los parámetros disponibles para la utilización del programa son los siguientes:

- `stretch`: aplica el efecto *stretch* (sección 3.2) a un audio pasado por parámetro, con un factor de estiramiento `float`.
  - `./main stretch <audio><float>[window_size][hop]`
  - `./main stretch mi_audio.wav 0.75`

  Donde `window_size` y `hop` son parámetros opcionales, donde el primero debe ser potencia de 2 y menor o igual a 2048, y el segundo debe ser menor al primero. Por default, sus valores son `2048` y `128` respectivamente. Para estiramientos grandes, el sonido dara un mejor resultado usando `hop` pequeños. `float` debe ser positivo.
- `Repitch`: aplica el efecto *Pitch-Shift* (sección 3.3) a un audio por el `float` dado.
  - `./main repitch <audio><float>`
  - `./main repitch mi_audio.wav 0.5`

  donde `float`=0.5 vuelve al audio una octava más grave y 2 lo vuelve una octava más agudo (2^n vuelve al audio n octavas más agudo o grave, según si n es positivo o negativo). Aunque puede tomar como entrada cualquier `float` distinto de cero, un rango razonable de entrada para que funcione correctamente es una octava y media mas agudo o grave (aprox de 0.35 a 2.82)
- `Reverb`: aplica el efecto `Reverb` (sección 3.4) al audio, donde el eco está basado en un .wav `impulse_response`.
  - `./main reverb <audio><impulse>`
  - `./main reverb mi_audio.wav impulse.wav`
- `Vocoder`: aplica el efecto `Vocoder` (sección 3.5) dados un audio `modulator` y otro `carrier` que debe ser más largo que `modulator`. Toma un parámetro opcional `window_size` que debe ser potencia de 2 y menor a 2048. Por default es 2048.
  - `./main vocoder <modulator><carrier>[window_size]`
  - `./main vocoder modulator.wav carrier.wav 2048`

[5] https://github.com/erikd/libsndfile

Referencias

[1] Jan Onderka. Pitch shifting of audio signals in realtime using stft on a digital signal processor. 2018. https://dspace.cvut.cz/bitstream/handle/10467/77279/F8-BP-2018-Onderka-Jan-thesis.pdf.

[2] William Arthur Sethares. *Rhythm and Transforms*. Springer, 2007. p. 105, https://sethares.engr.wisc.edu/vocoders/Transforms.pdf.

[3] François Grondin. Guitar pitch shifter algorithm, 2009. http://www.guitarpitchshifter.com/algorithm.html.

[4] Amalia De Götzen Daniel Arfib, Nicola Bernardini. *Traditional implementation of a phase vocoder: the tricks of the trade*. 2000. https://pdfs.semanticscholar.org/f1ec/2695adfb65c439d75837b342d6d7b3cc642a.pdf.

[5] Michael A. peimani. *Pitch Correction for the Human Voice*. 2009. http://dave.ucsc.edu/physics195/thesis_2009/m_peimani.pdf.

[6] Jeffrey Brokish. Lab 5 - pitch synthesis. https://courses.engr.illinois.edu/ece420/fa2019/lab5/lab/#overlap-add-algorithm.

[7] Cort Lippe Richard Dudas. The phase vocoder - part i. https://cycling74.com/tutorials/the-phase-vocoder-%E2%80%93-part-i.

[8] Convolution reverb. https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/convolution.html.

[9] Sam Drazin. *A Real-time Channel Vocoder AudioUnits Plug-in Emulating a Vocal Harmonizer*. http://www.samdrazin.com/classes/mmi505/finalproject/finalpaper.pdf.

[10] Udo Zölzer. *DAFX: Digital Audio Effects*. Wiley, 2011.