Experimentos en la producción de forma para un escultura robótica

Emiliano Causa, Ariel Uzal

Introducción:

La propuesta discursiva: Extremófilos

El presente trabajo surge de un proyecto artístico llamado “Extremófilos”, que consiste en la realización de una serie de esculturas robóticas que representan organismos que se han adaptado a la hostilidad de nuestra cultura.
Por definición, los “Extremófilos” son seres que se han adaptado para vivir en condiciones extremas (consideradas hostiles para la vida en general), como la falta de oxígeno o agua, las temperaturas extremas, la exposición a la radiación, etc.
Nuestro proyecto propone la construcción de 4 esculturas robóticas que representan seres imaginarios que se han adaptado a vivir en las condiciones extremas de nuestra cultura, verdaderos extremófilos de nuestra forma de vida.
Nuestro bestiario se compone de los siguientes:

El *Consumófago* es un ser que se ha adaptado a la incesante “compulsión al consumo” que sufrimos. Es un animal que no puede reprimir su necesidad de consumir y acumular. Tiene unas inmensas mandíbulas que intentan morder y tragar cualquier cosa, sin ojos para distinguir lo que engulle. Este animal desea consumir objetos y poder integrarlos a su ser, pero su sistema digestivo es incapaz de metabolizar lo tragado, deja pasar de largo los objetos acumulándolos como desechos. Su consumo se transforma así en una actividad absurda.
En su ambiente, el Consumófago posee plantas/algas que en sus hojas que exponen ofertas de sitios de compra en Internet, lo que implican para éste una constante tentación.

El *Apocaliptópodo* es un organismo que se ha adaptado a la falta de proyección.
La sensación generalizada con la que vivimos, en un mundo arrasado por la crisis económica mundial que genera cada vez más exclusión, una alarma permanente de catástrofe ambiental, la sensación de inseguridad económica y física, frente al temor constante a ser agredidos, ha provocado que éste ser desarrolle una defensa para sobrevivir al extremo ambiente que habita.
Éste es un ser acorazado que repta buscando oportunidades en avisos laborales, en un ambiente repleto de estadísticas, reportes del clima y de la bolsa de valores.

El *Egófilo* se ha adaptado a la contemporánea necesidad de exponer públicamente la intimidad, frente a ese imperioso dictamen de construir una imagen de sí mismo, siempre espectacularizada.

Este ser nunca toca la tierra, por lo cual vive suspendido de un cable de internet, ansioso de encontrar la mirada de la gente para desplegar, cual pavo real, un plumaje repleto de avatares y perfiles de redes sociales.
Su cuerpo se contorsiona en forma extrema esforzándose por obtener el interés de los otros.

El *Infófilo* es un organismo constantemente interferido por el exceso de información presente en nuestra cultura.
Su cuerpo ha perdido la orientación bilateral para adquirir una simetría radial acorde a su indecisión, y desorientación constante.
Cada nuevo atisbo de información que llega a él le obliga a cambiar su foco de atención, por esto es incapaz de terminar cualquier actividad y llegar a resultados.
Su cuerpo se ha plagado de aparatos perceptivos, dotándolo de múltiples antenas, sensores y ocho patas que le permiten cambiar rápidamente su destino.

El presente trabajo muestra los primeros bocetos en la producción de una de estas esculturas robóticas, el Consumófago. Para este se ha pensado en realizar un robot zoomorfo que respeta la forma de una especie que es mezcla de tiburón y de organismo tubular, con inmensas mandíbulas móviles. El propósito del trabajo es el de un estudio de construcción de la forma escultórica y robótica mediante técnicas generativas.

Las etapas de desarrollo

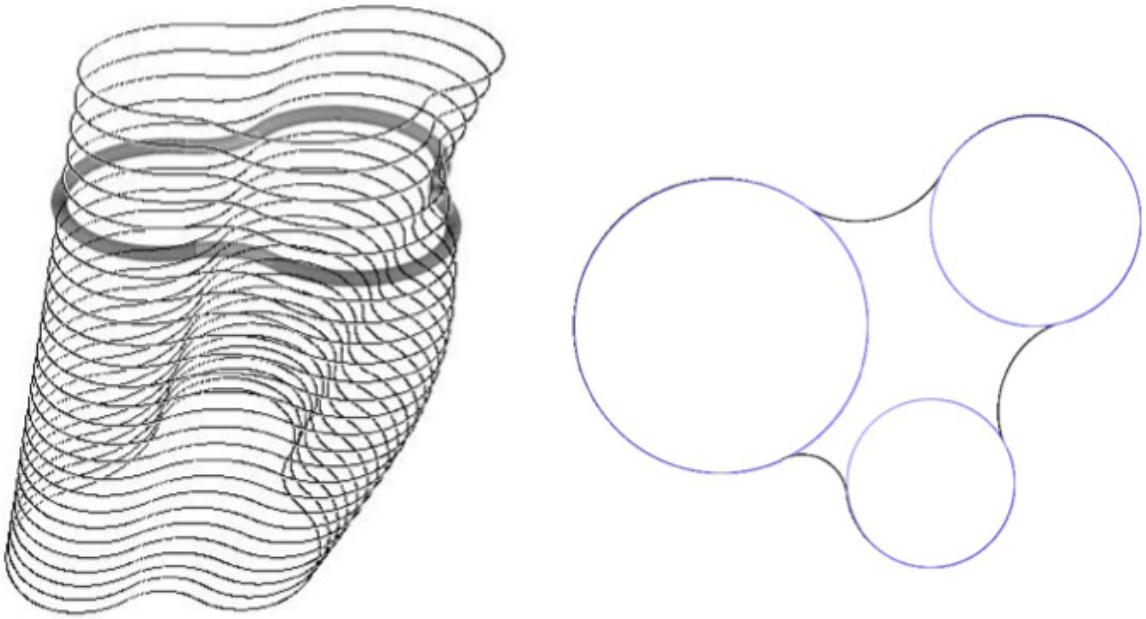
El contenido del presente trabajo está dividido en dos partes: la primera vinculada con el desarrollo de las técnicas y herramientas necesarias para el diseño digital de la forma de la escultura, y la segunda con el desarrollo de técnicas y dispositivos para la fabricación de la escultura y su motorización.

PARTE 1: El diseño de la forma

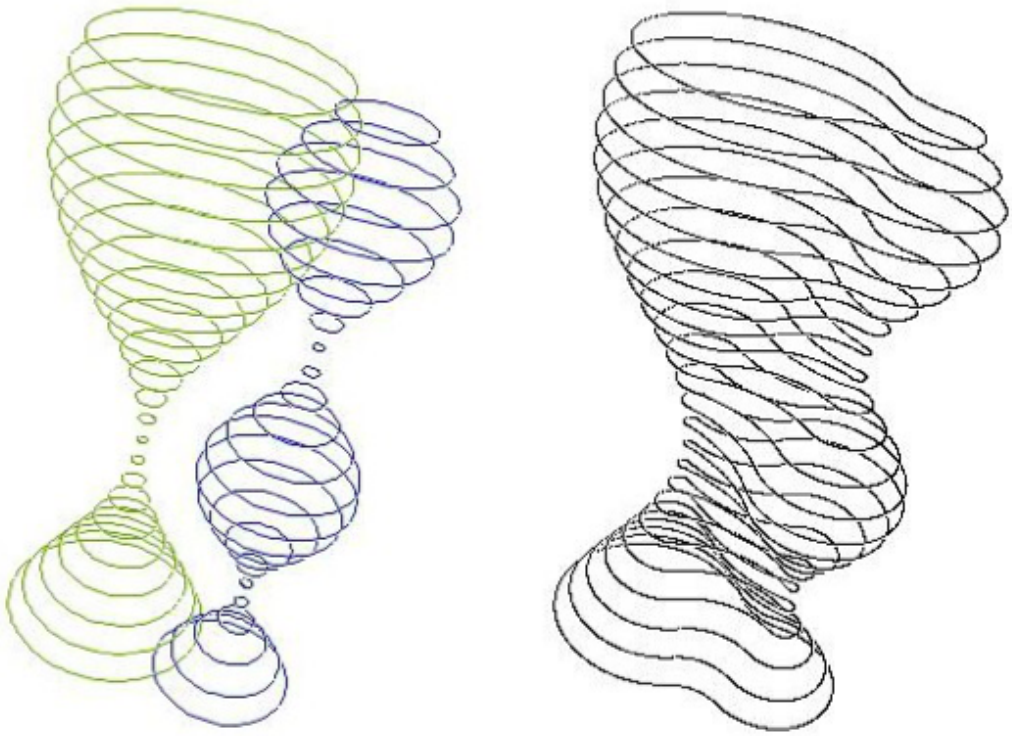
1.1 La propuesta morfológica, posibilidades: los planos seriados

En pos de diseñar y construir la forma de esta escultura robótica se buscó alguna técnica sencilla que permitiera alcanzar formas tridimensionales mediante técnicas de diseño generativo y prototipado alcanzables. Las principales técnicas de prototipado, y las más accesibles, son aquellas que permiten producir formas bidimensionales, como el corte láser o el corte por control numérico. A su vez, estas técnicas se puede combinar con el uso de software de gráfica vectorial tales como el InkScape. Una de las ventajas de la producción de formas bidimensionales es que se puede llegar hasta el uso de tijeras y papeles o cartones para su producción. Por todo esto, se pensó alguna forma de producir formas tridimesionales a partir del uso de figuras bidimensionales. Existen varias técnicas posibles para este tipo de

diseño, pero la producción de volúmenes por planos seriados nos pareció una de las más sencillas, al mismo tiempo que ya la habíamos abordado en proyectos anteriores.
En un proyecto para la producción de formas mediante algoritmos genéticos se investigó la construcción de una forma mediante planos seriados. Cada plano estaba conformado por círculos que estaban empalmados con otros círculos.
En la imagen que sigue se puede observar un volumen construído por planos y uno de sus planos compuesto por círculos empalmados.



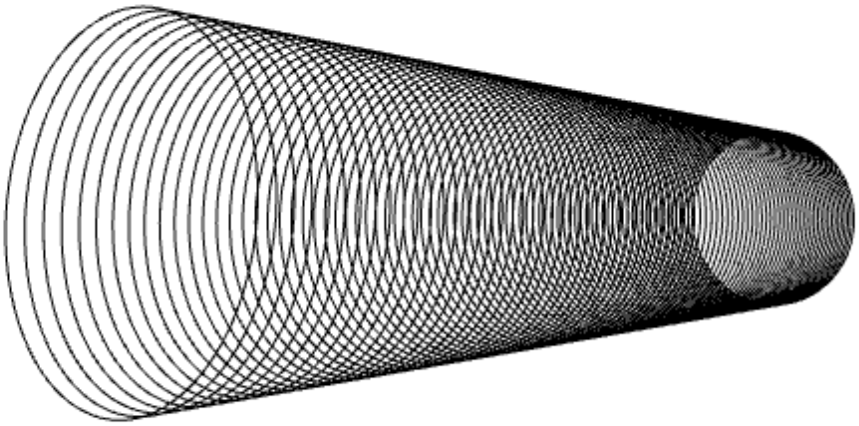
La clave para la construcción de la forma en estos volúmenes consistía en modificar la posición y tamaño de los círculos de plano en plano para lograr el modelado. En la figura siguiente se puede observar el cambio progresivo de los círculos y la forma en que incide en el volumen resultante:



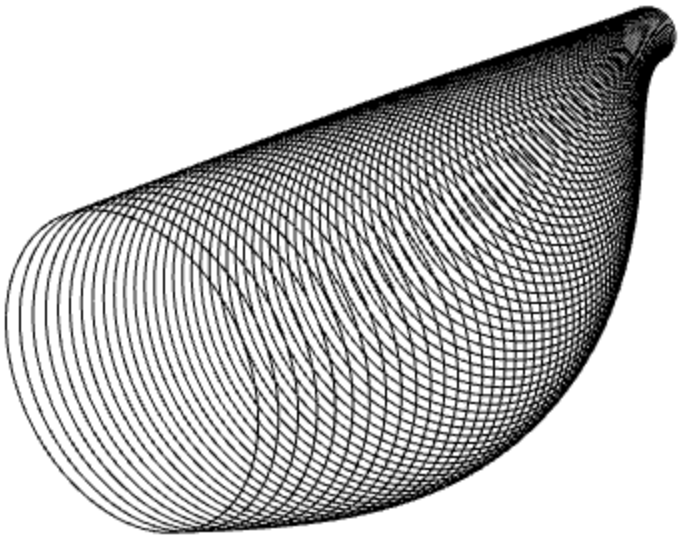
Por último, en la siguiente imagen se puede ver un prototipo real cortado con láser en acrílico. Un detalle más acabado de este proceso puede leerse en el texto “Algoritmos Genéticos aplicados a la generación y producción de formas escultóricas” (Causa, 2013).

A partir de la experiencia obtenida en el anterior trabajo se buscó la realización de formas más complejas, en vez de trabajar con círculos se utilizaron elipses a las que se le agregan o quitan apéndices. Con una secuencia de elipses, que varían en tamaño posiciones y apéndices, se construye un cuerpo. Por último los varios cuerpos pueden operarse entre sí para producir cuerpos más complejos.

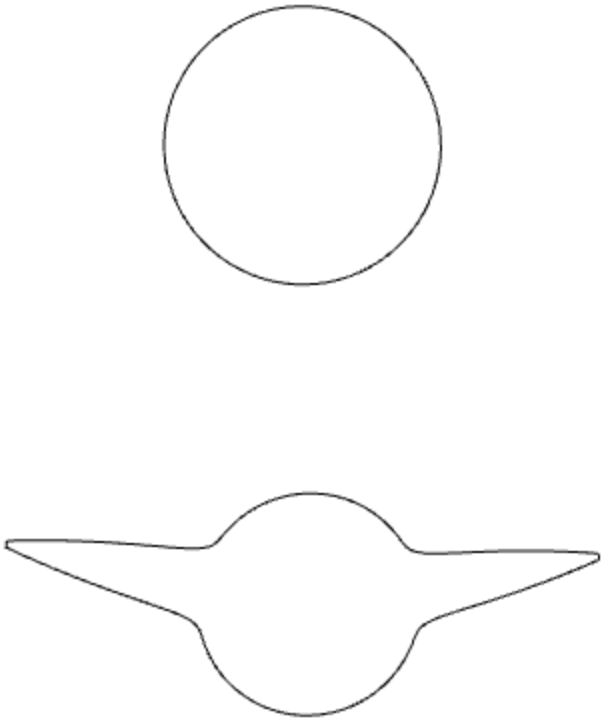
Ya que el objetivo del proyecto es producir estructuras zoomorfas, se buscó una figura de base que favoreciera la simetría bilateral. En este caso la forma deseada es la de los peces, así que se parte de la idea de construir un tubo, como una secuencia de planos elípticos, que varíen durante su desarrollo.
Por ejemplo, de una secuencia de elipses (en este caso elipses de igual diámetro vertical que horizontal, es decir: círculos) se puede obtener una forma tubular, un cilindro.



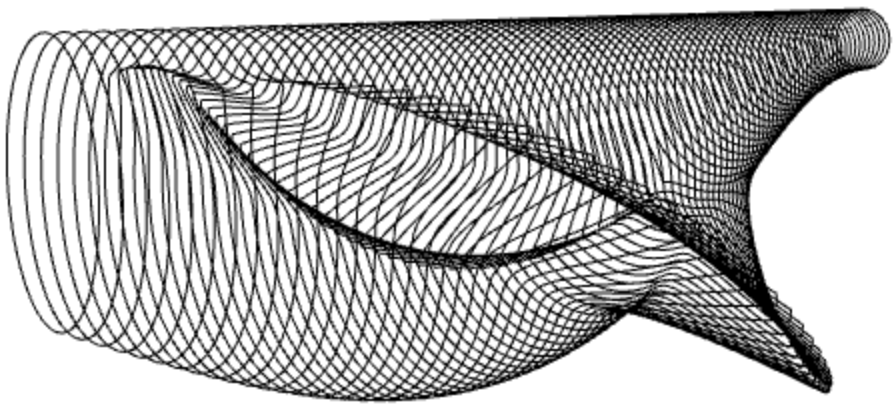
Pero si a esta forma le vamos variando sus diámetros durante el desarrollo, iremos obteniendo otras formas más sutiles:



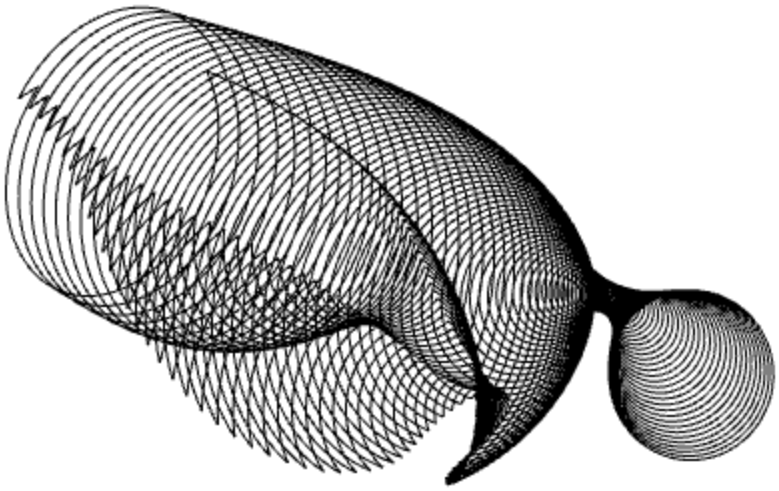
Cabe aclarar, que en el gráfico anterior, no sólo se variaron los diámetros, sino que las posición de los centros de la elipses, se variaron de forma tal de que estas estén alineadas en la parte superior.
Al ejemplo anterior podemos sumarles unos apéndices en cada elipse de la siguiente forma:



Si estos apéndices son aplicados a parte de la progresión, variándolos en sus parámetros (posición y prominencia), se obtiene la siguiente forma:

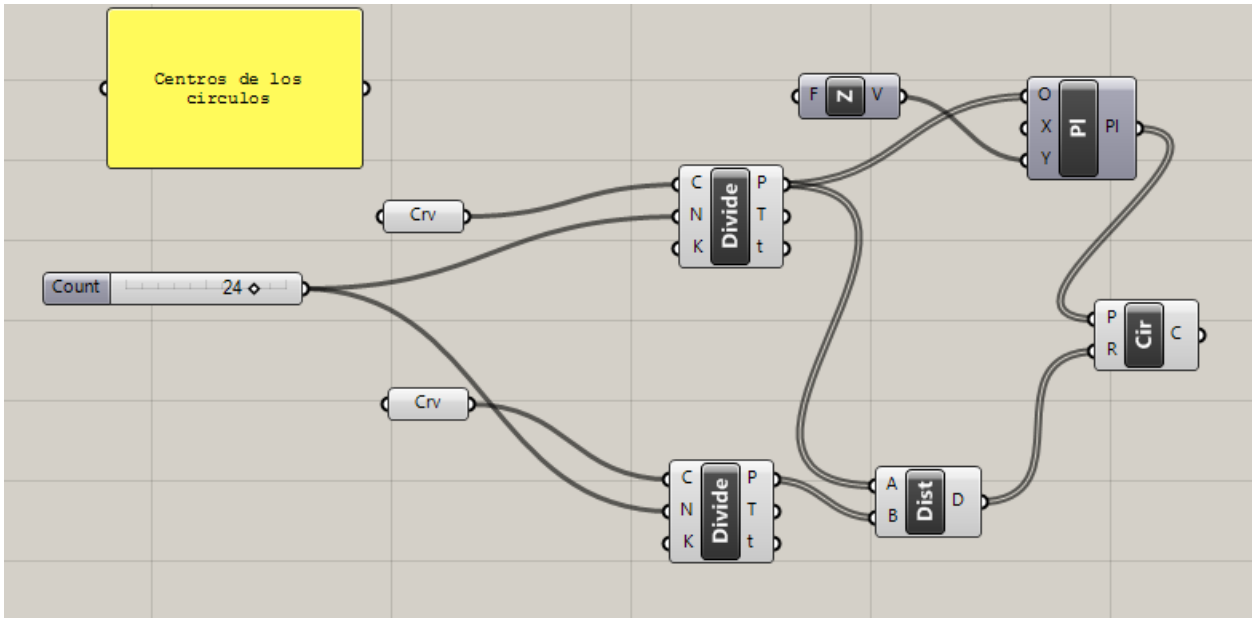


Por último, si a un cuerpo construido con esta lógica lo interceptamos con con otros cuerpos, generando substracciones y adiciones, obtenemos cuerpos más complejos:



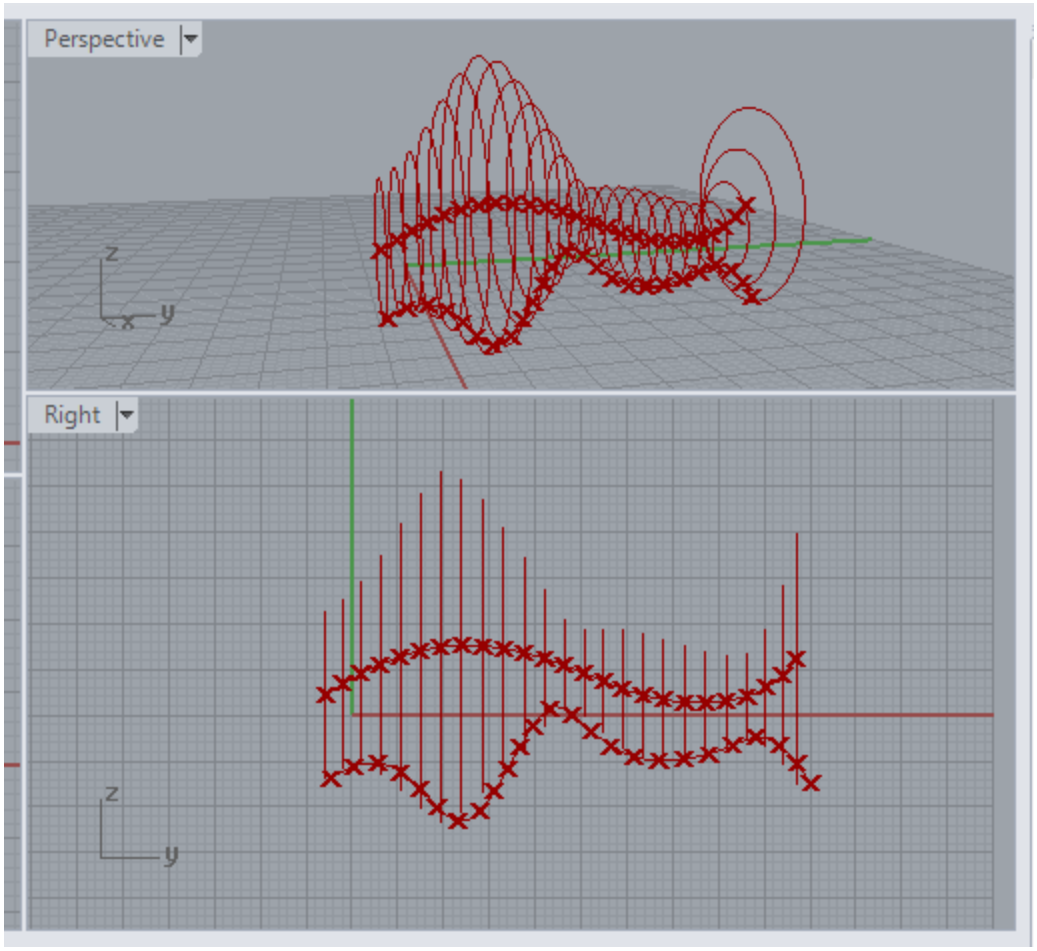
1.2 Los procesos de diseño: Grasshopper

La primeras pruebas, a la hora de producir la forma, se realizaron utilizando la aplicación Rhinoceros con el plugin Grasshoper. Este plugin funciona como un lenguaje programación para realizar Diseño Paramétrico, esto es, definir formas forma mediante algoritmos. Grasshoper trabaja con un entorno de programación visual, en donde se unen cajas con cables, para ir llevando la información de izquierda a derecha. Cada caja, recibe información, la cual procesa (funcionando como un comando) y envía por cables a otras cajas.
En el ejemplo de abajo se pueden observar que las cajas ubicadas más a la izquierda son las que funcionan como ingreso de información, y las del extremo opuesto como salida. Las de ingreso de información son dos cajas “Crv” y una “Count”. Las cajas “Crv” sirven para seleccionar curvas dibujadas en Rhinoceros. Esto es una de las grandes ventajas de este lenguaje, el hecho de combinar comandos con figuras de dibujo, ya que se puede avanzar mediante el dibujo y modelado, o la programación, de acuerdo a lo que resulte más sencillo en cada etapa del proceso.



En el gráfico de abajo se pueden observar dos curvas, que son las tomadas por las cajas “Crv”. En este ejemplo, la curva ubicada más arriba determina la posición de los centros de una secuencia de círculos (ubicados como planos seriados), la distancia entre las dos curvas determinan el radio de cada uno de los círculos. Esto permite que con sólo dos curvas se puede determinar la progresión de los círculos, tanto en posición como en tamaño.
Ubicado a la derecha del algoritmo (el esquema de arriba), se puede observar una caja “Cir” que es la encargada de dibujar los círculos.
Hagamos un seguimiento del algoritmo: las dos cajas “Divide” reciben información de las curvas (mediante las cajas “Crv”) y se encargan de dividir estas en una secuencia de puntos equidistantes (en el recorrido de la curva), la cantidad de puntos es 24 y está determinada

por el objeto (caja) “Count”. Un objeto “Dist” recibe la información de las cajas “Divide” y calcula la distancia entre los puntos de ambas curvas. Luego, este, envía la información al objeto “Cir” para usarla como radio de los círculos. Por la parte superior del esquema, un objeto “Pl” recibe información de un “Divide” y ubica planos que coinciden con las posición de los puntos de la curva, estos planos sirven para ubicar el centro y la orientación de los círculos.



Grasshoper se mostró como una herramienta muy eficiente en muchos aspectos, principalmente por la fluidez de pasaje de la situación de dibujo y modelado a la de programación. Sin embargo existen algunas cuestiones que implican ciertas desventajas, estas no siempre responden a la herramienta en sí, sino que a veces se relacionan con la forma de trabajo elegida, así como a factores externos. Una de las desventajas se relaciona con que en Grasshopper resulta sencillo tomar geometrías modeladas en Rhinoceros, y usualmente es la forma más sencilla de tomar geometrías sutiles, por el contrario, producir geometrías de base (sutiles) generadas en forma paramétricas puede resultar bastante engorroso. Es decir, parece una herramienta que requiere mucho del dibujo y modelado como forma de ingresar información y no está del todo pensada para crear forma de manera

estrictamente algorítmica. Por ejemplo, crear una curva interesante e irregular en forma numérica (y que esta resulte parametrizable) requiere de una batería de objetos que a veces se hace inmanejables en pantalla. Si bien Grasshopper cuenta con una gran cantidad de elementos, categorías y operadores, a veces ciertas problemáticas requieren recorrer ciertos caminos, que no siempre son del todo intuitivos, a veces esto se relaciona con cosas están pensadas para ser post-procesadas en Rhinoceros y no mediante algoritmos.
Es importante destacar que la investigación que hicimos de la herramienta ha sido veloz y en el marco de nuestro proyecto y es probable que un experto en su uso seguramente pondría en duda algunas de nuestras críticas. Quizás, en ese sentido, nuestras críticas puedan ser tomadas en cuentas desde el punto de vista de aquel que desea iniciarse en este tipo de herramientas. Es importante, también, destacar que Grasshopper es una herramienta que posee una verdadera autoridad en el tema, y es difícil entrar en este campo sin cruzarse inmediatamente con alguna referencia a este.
Por último, nos pareció una desventaja importante, el hecho de que, si bien Grasshopper es una herramienta libre, corre en la aplicación Rhinoceros que es privativa, lo que casi la convierte en privativa. A partir de esto último y de algunas de las desventajas antes planteadas, decidimos cambiar de estrategia, realizando un software a medida para nuestro diseño, pero que pudiera aprovechar las ventajas de Grasshopper. Por esto, se nos ocurrió realizar un software de generación de forma paramétrico en Processing, que luego pudiera trasladar su resultado a Grasshopper para poder continuar en este.

1.3 Desarrollo de los algoritmos de la forma

A partir de las necesidades detalladas en los párrafos anteriores, se decidió construir un conjunto de clases y métodos en Processing (un lenguaje de programación libre que utilizamos con frecuencia) que permitiera construir una forma de manera paramétrica. La estrategia que se eligió es la que ya fue expuesta:

- Una serie de elipses construyen un “cuerpo”. A estas elipses se les puede llamar “costillas”
- A estas elipse se le puede agregar o quitar “apéndices”
- Tanto las elipses como sus apéndices pueden variar durante la progresión siguiendo ciertas “curvas”
- Varios cuerpos puede ser operados mediantes adiciones y substracciones para crear cuerpos más complejos

Para esto se programó la clase Cuerpo que posee un conjunto de métodos combinables con algunas funciones:

Cuerpo: es la clase principal y encargada de construir el cuerpo. Posee un conjunto de métodos para determinar la forma de este:

- `Cuerpo( int cantidad )`: el método constructor de la clase en el que el parámetro “cantidad” determina la cantidad de costillas (plano) que tendrá en cuerpo.

- `setColumna( float[ ] columnaX_, float[ ] columnaY_, float[ ] columnaZ_ )`: este método determina la posición de la “columna vertebral” es decir, cómo se ubican los centros de los planos (las costillas) en la progresión.

- `setElipses( float anchos[ ], float altos[ ] )`: este método determina los tamaños de los diámetros verticales y horizontales de las elipses.

- `apendiceSuave( int resolucion, int desdeCostilla, int hastaCostilla, float[ ] angDesde, float[ ] angHasta, float[ ] agregado )`: genera un apéndice que se ubica (en forma angular por coordenadas polares) entre el ángulo “angDesde” hasta el “angHasta”. El parámetro “agregado” determina el nivel de prominencia del apéndice.

- `apendiceAgudo( int resolucion, int desdeCostilla, int hastaCostilla, float[ ] angDesde, float[ ] angHasta, float[ ] agregado )`: este método es como el anterior pero la forma del apéndice que genera es más aguda.

- `operar( Cuerpo otro, float dx, float dy, int indiceDesde, int indiceHasta, int indiceOffSet, float angulo, boolean horarioUno, boolean horarioDos )`: etse método permite operar dos cuerpos entre sí para obtener uno resultado de la adición o subtracción entre los dos. Se determina la posición relativa entre los cuerpos (con “dx” y “dy”) así como los índice que determinan en cuáles planos se ejecuta la operación.

- `cambiarCostillasCon( Cuerpo otro, float dx, float dy, int indiceDesde, int indiceHasta, int indiceOffSet )`: este método permite que un cuerpo reemplace sus costillas con las de otro.

Como puede observarse, en gran parte de los métodos antes citados, los parámetros son arreglos (aquellos que terminan con corchetes: “[ ]”), esto se debe a que el parámetro no se aplica a una sola elipse, sino a una serie de elipses (el cuerpo) y por ende debe ser una serie de valores. Para complementar estos métodos, se usaron un conjunto de funciones que construyen arreglos que respetan ciertas progresiones:

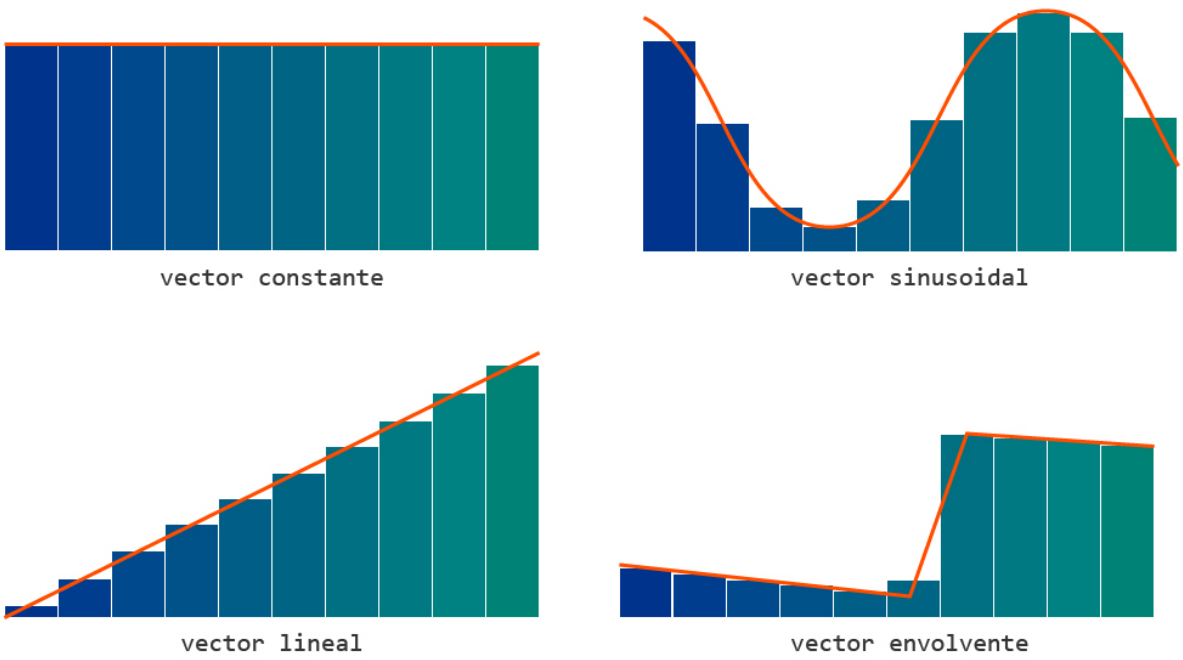
- `float[ ] vecConst( int cantidad, float valorC )`: esta función devuelve un arreglo de un tamaño determinado por cantidad, en donde todos los valores posee el valor constante “valorC”.

- `float[ ] vecLinea( int cantidad, float desde, float hasta ):` en este caso, la función devuelve un arreglo con una progresión lineal que va entre los dos valores límites (“desde”, “hasta”).

- `float[ ] vecSeno( int cantidad, float desde, float hasta, float ang1, float ang2 ):` la progresión de esta función es sinusoidal, moviendo la función entre dos valores límites (“desde”, “hasta”) y moviendo la fase del seno entre dos ángulos.

- `float[ ] vecEnvolvente( int cantidad, float[ ] t, float[ ] v ):` en función genera una envolvente, una curva que transita entre una secuencia de puntos, los puntos están definidos por dos arreglos “t” y “v” que describen los tiempo y valores de los puntos por los que pasa la curva.

La siguiente figura grafica los cuatro tipos de arreglos descriptos.

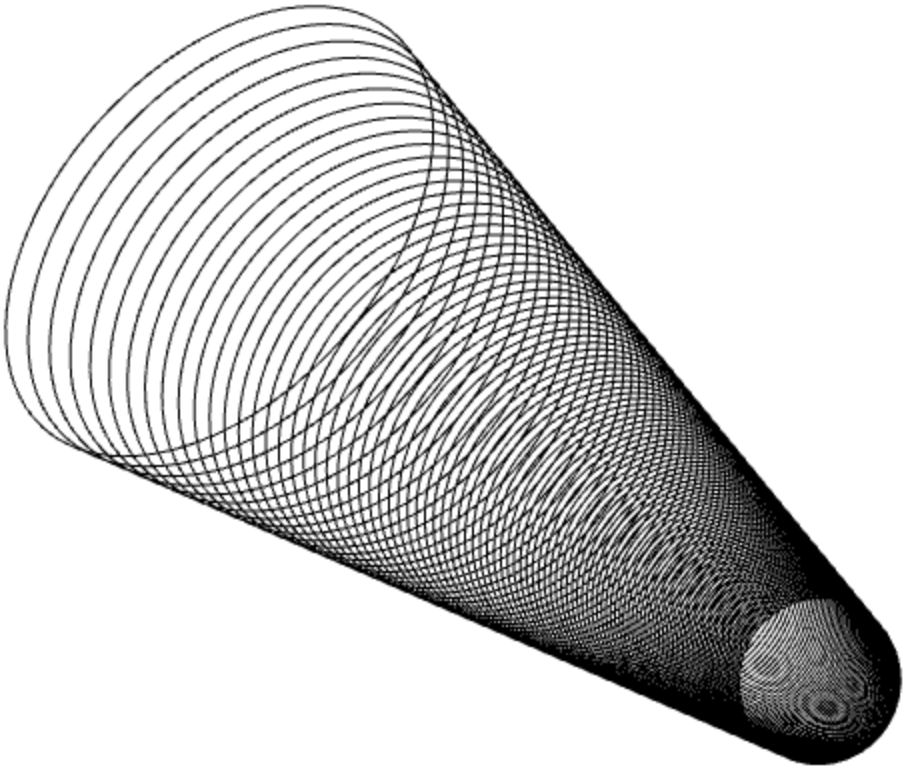


Ya hemos revisado la clase “Cuerpo” y las principales funciones para generar sus parámetros, pasemos, entonces, a ver ejemplos concretos de su uso.
Por ejemplo, en el código que se expone debajo se puede observar (en la segunda línea) que se declara un objeto “k” de tipo Cuerpo al que se le pasa como parámetro una variable “cantidad” con el valor 100. Esto significa que el cuerpo es una secuencia de 100 planos seriados, estos planos son elipses. Debajo del código se observa un gráfico con la secuencia de elipses conformando el volumen.

Debajo de la declaración del Cuerpo se pueden verse tres líneas que declaran tres arreglos (series de números), d[], p[] y z[], que son cargados en el caso de d[] y p[] con valores constantes (60 para d[] y 0 para p[]) y en el caso de z[] con una progresión lineal de 0 a 600.
La línea “k.setElipses(d, d);” determina que los diámetros horizontales y verticales de las elipses toman el valor 60 extraído del arreglo d[] (el cual posee 60 en toda su serie).
Por último, la línea “k.setColumna(p, d, z);” organiza las posiciones de los planos, asignandoles los parámetros (X,Y,Z) que en este caso las posiciones en X son tomadas de la serie p[] (que posee 0 en todas sus posiciones), los valores de Y se toman de d[] (que posee 60 en todas sus posiciones) y a Z se le asigna el arreglo z[] que avanza progresivamente de 0 a 600 en 100 pasos (es decir: 0,6,12,18,...), este último determina la separación en profundidad de los planos.

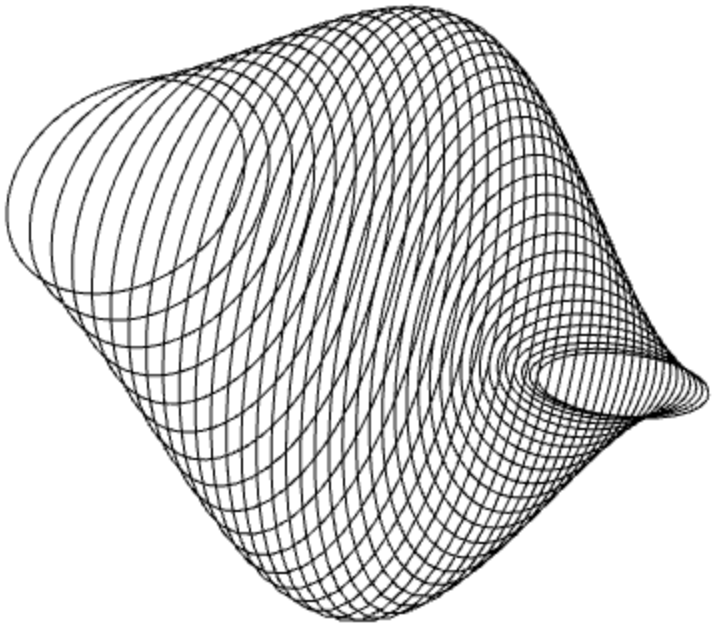
```
int cantidad = 100;

k = new Cuerpo( cantidad );

float d[] = vecConst( cantidad, 60 );
float p[] = vecConst( cantidad, 0 );
float z[] = vecLinea( cantidad, 0, 600 );

k.setElipses( d, d );
k.setColumna( p, d, z );
```

La siguiente figura muestra el volumen resultante del código anterior.



Para intentar comprender un poco mejor la estrategia, en el ejemplo siguiente modificamos los diámetros de las elipses agregando un nuevo arreglo llamado d2[], al que se le carga una serie que sigue una curva sinusoidal, declarado en la cuarta línea del código. En la anteúltima línea se puede ver como en “k.setElipses(d, d2);” se usa d2[] para determinar los diámetros verticales. Esto hace que en el gráfico que le sigue el cilindro se engrosa en altura hacia la mitad de su recorrido, también se puede ver que dicha progresión sigue la curva sinusoidal antes descripta:

```
int cantidad = 40;

 k = new Cuerpo( cantidad );

 float d[] = vecConst( cantidad, 60 );
 float d2[] = vecSeno( cantidad, 20 , 140 , radians(-45) , radians(270) );
 float p[] = vecConst( cantidad, 0 );
 float z[] = vecLinea( cantidad, 0, 400 );

 k.setElipses( d, d2 );
 k.setColumna( p, d, z );
```

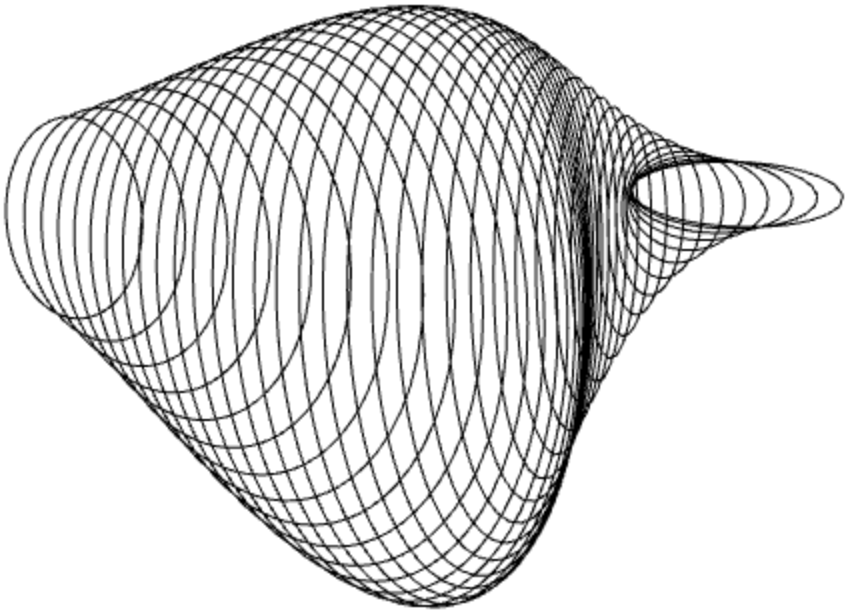


Dando un paso más, en el siguiente ejemplo se reemplazó a d[] por d1[], un arreglo al que se le asignó una serie que también sigue una sinusoidal (en la tercer línea del código). Si bien d1[] y d2[] son sinusoides que organizan valores en un rango que va de 20 a 140 (como puede verse en sus parámetros) la principal diferencia es que está corridos en fase, ya que d1[] copia la forma de la sinusoide entre los 0 y los 360 grados, mientras que d2[] lo hace entre los -45 y los 270 grados. El nuevo arreglo, d1[], determina los diámetros horizontales de las elipses, lo que puede verse en el nuevo gráfico que este cambio produce. Al mover ambos diámetros, la forma tubular de la serie se pierde y da lugar a una forma más plástica y sutil:

```
int cantidad = 40;

 k = new Cuerpo( cantidad );

 float d1[] = vecSeno( cantidad, 20, 140, radians(0), radians(360) );
 float d2[] = vecSeno( cantidad, 20, 140, radians(-45), radians(270) );
 float p[] = vecConst( cantidad, 0 );
 float y[] = vecSeno( cantidad, 20, 70, radians(-45), radians(270) );
 float z[] = vecLinea( cantidad, 0, 400 );

 k.setElipses( d1, d2 );
```

```
k.setColumna( p, y, z );
```

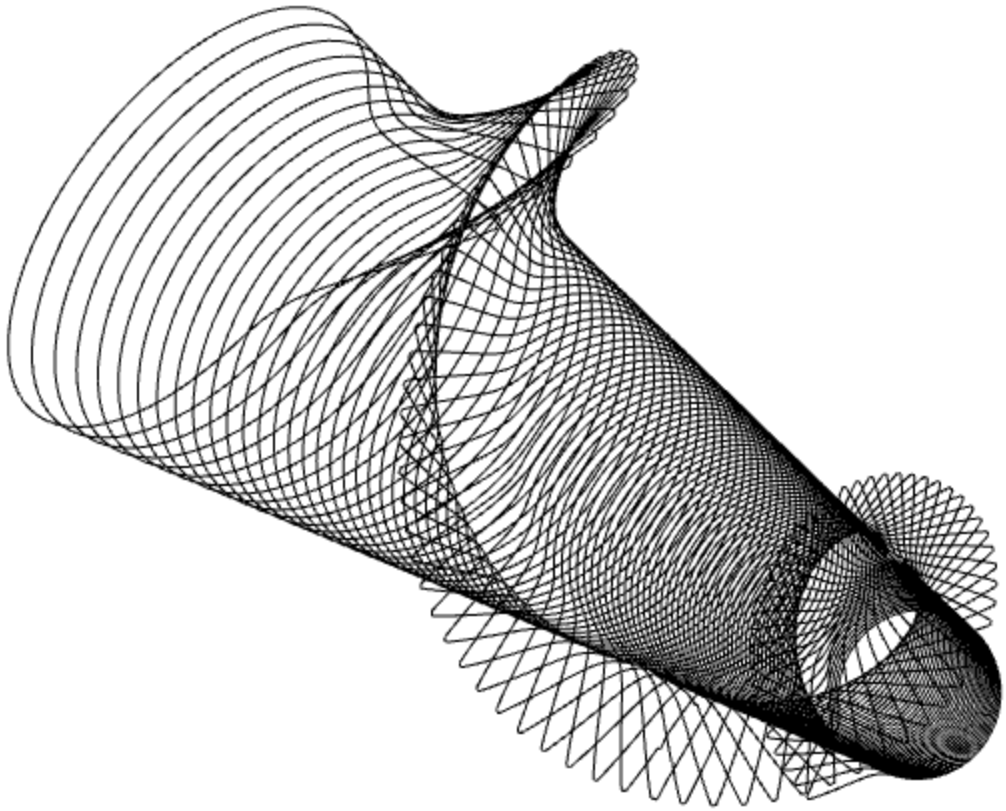
La siguiente figura muestra el volumen resultante del código anterior.



En el siguiente ejemplo se muestra cómo funcionan los apéndices. Volvimos al cuerpo del primer ejemplo, en el que los diámetros se comportan en forma constante, usando un arreglo d[] que posee el valor 60 en todas sus posiciones. Al final del código se pueden apreciar cuatro líneas en las que primero se declaran tres arreglos (angDesde[], angHasta[] y agregado[]) y que luego son usados en el método que tienen los cuerpos para agregar apéndices: "k.apendiceAgudo(resolucion, 0, cantidad-1, angDesde, angHasta, agregado);". La idea es la siguiente: los apéndices son salientes que se generan en las elipses, la posición de estas se determinan por el ángulo, por ejemplo el ángulo 0 se ubica en la parte superior, el de 90 grados en el costado derecho, el de 180 grados abajo y así siguiendo el giro. Por eso en el gráfico se puede observar que los apéndices de las elipses se van corriendo siguiendo una progresión, lo que hace que efectúen un giro alrededor del tubo, como en una helicoide.

```
int cantidad = 100;

 k = new Cuerpo( cantidad );

 float d[] = vecConst( cantidad, 60 );
 float p[] = vecConst( cantidad, 0 );
 float z[] = vecLinea( cantidad, 0, 600 );
```

```
  k.setElipses( d, d );
  k.setColumna( p, d, z );

  float angDesde[] = vecLinea( cantidad, radians(0), radians(600) );
  float angHasta[] = vecLinea( cantidad, radians(80), radians(600+80) );
  float agregado[] = vecConst( cantidad, 50 );

   k.apendiceAgudo( resolucion, 0, cantidad-1, angDesde, angHasta, agregado
);
```

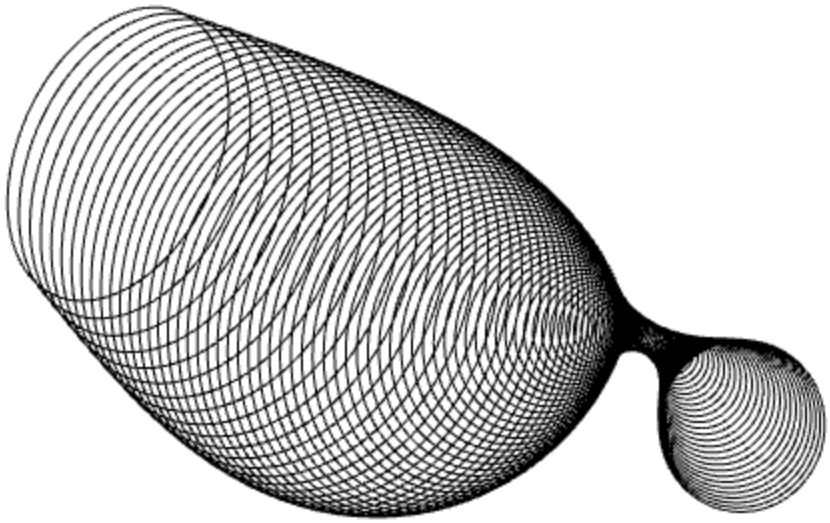


En este último ejemplo mostraremos la última acción posible que es la de operar cuerpos entre sí. El código está dividido en tres partes por gráficos que muestran cada una de las partes:

La primer parte del código muestra cómo se construye un cuerpo siguiendo los lineamientos explicados en los anteriores ejemplos. Este cuerpo se llama “k” y debido a la variación de sus

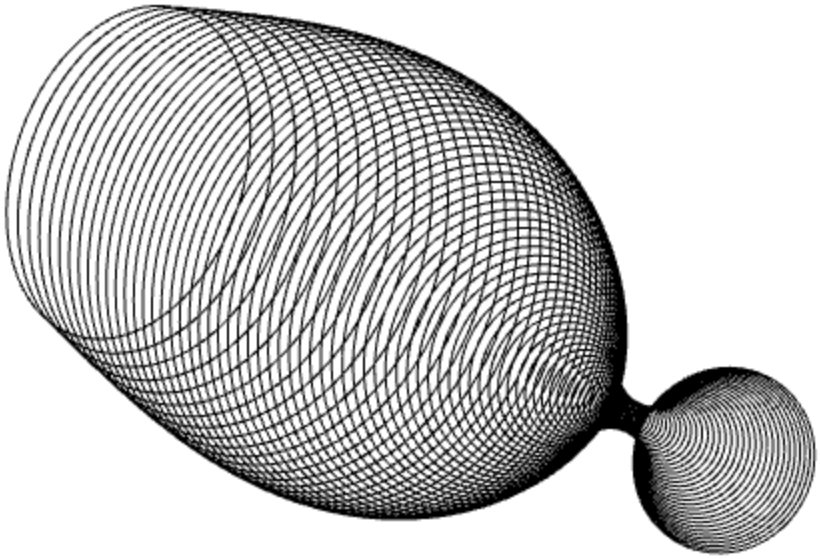
dos diámetros produce una forma similar a una curva recostada con cierto corrimiento de su eje central, como puede observarse en el gráfico:

```
 k = new Cuerpo( cantidad );
float d[] = vecSeno( cantidad, 10, 80, radians(0), radians( 400) );
float p[] = vecConst( cantidad, 0 );
float z[] = vecLinea( cantidad, 0, 600 );
float y[] = vecSeno( cantidad, 10, 50, radians(0), radians( 400) );
k.setElipses( d, d );
k.setColumna( p, y, z );
```



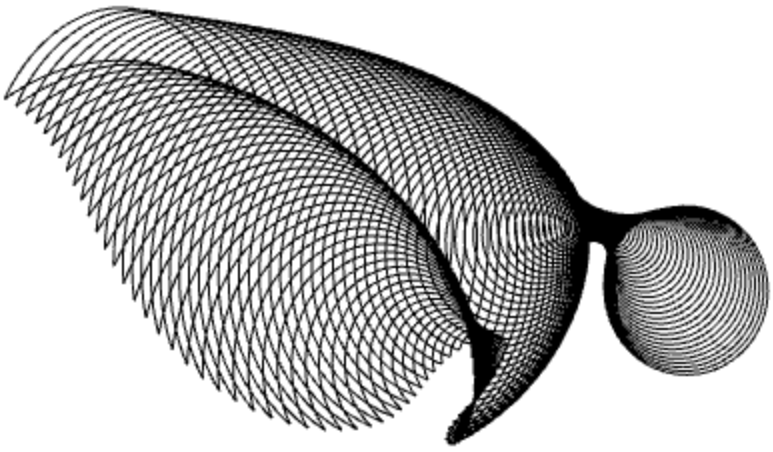
En la segunda parte se construye otro cuerpo llamado “l” que es de forma muy similar al anterior aunque con un eje más centrado, como puede observarse en la figura:

```
l = new Cuerpo( cantidad );
float d2[] = vecSeno( cantidad, 10, 90, radians(0), radians( 400) );
float p2[] = vecConst( cantidad, 80 );
float y2[] = vecSeno( cantidad, 10, 50, radians(0), radians( 400) );
l.setElipses( d2, d2 );
l.setColumna( p, p2, z );
```



En esta tercera y última parte el cuerpo “k” es operado con el cuerpo “l”, en este caso mediante una substracción. Es decir que la cuerpo “k” se le sustrae el cuerpo “l” generando la figura que se puede observar debajo:

```
k.operar( l, 0, 10, 0, 70, 20, radians(180), true, false );
```



Para sintetizar, la nueva estrategia de producción de forma consiste en la generación de cuerpos, mediante planos seriados de elipses a las que se le aplican progresiones (arreglos) que siguen curvas constantes, lineales, sinusoidales o de envolventes (esta será vista más adelante). Estas progresiones se aplican a los diámetros horizontales, verticales, y a las posiciones en el espacio de estos planos. A su vez, a estos planos (elipses) se le pueden agregar o quitar apéndices que también pueden seguir progresiones para determinar sus posiciones o nivel de prominencia. Por último, se pueden construir diferentes cuerpos y operarlos mediante adición o subtracción entre sí.

1.4 Desarrollo de la GUI

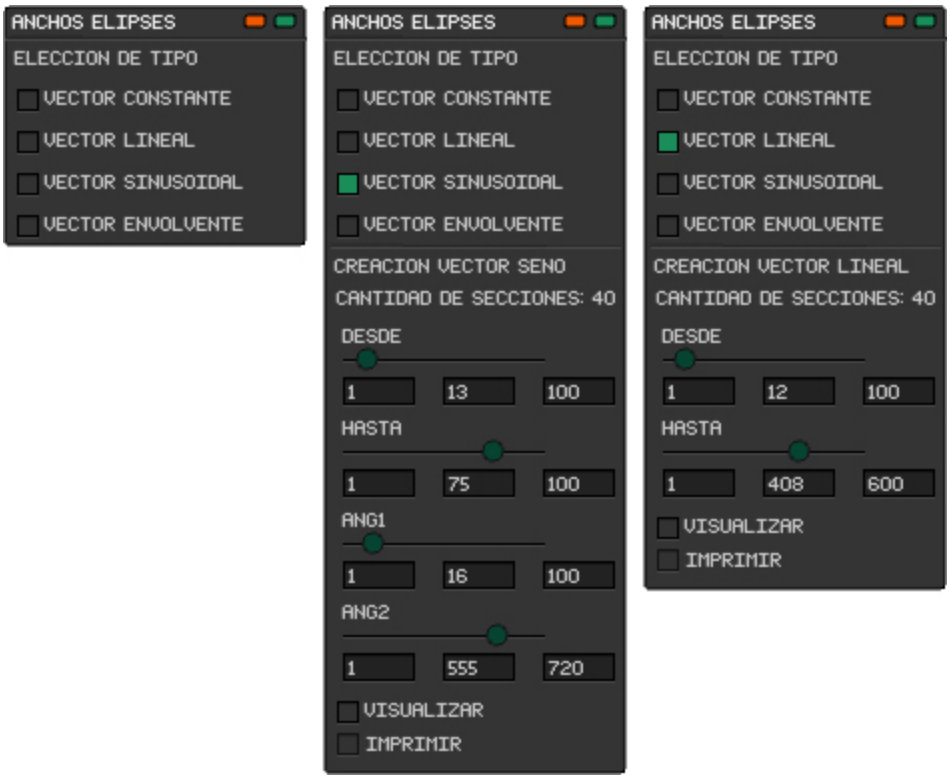
Una vez desarrollados los algoritmos de software descritos arriba, se hizo necesario el desarrollo de un sistema de interfaz gráfica que permita utilizarlos de manera práctica.

Como ya fue mencionado anteriormente, los arreglos que definen la forma de los cuerpos pueden ser definidos manualmente o a través de funciones de código específicas para la generación de los mismos. Ambas opciones, al ser implementadas en el código de la aplicación, no permiten la realización de cambios en tiempo real. El código debe modificarse, se debe ejecutar la aplicación, y si se desean realizar cambios en la configuración formal del volumen, se debe modificar nuevamente el código y volver a ejecutar la aplicación para visualizar dichos cambios.

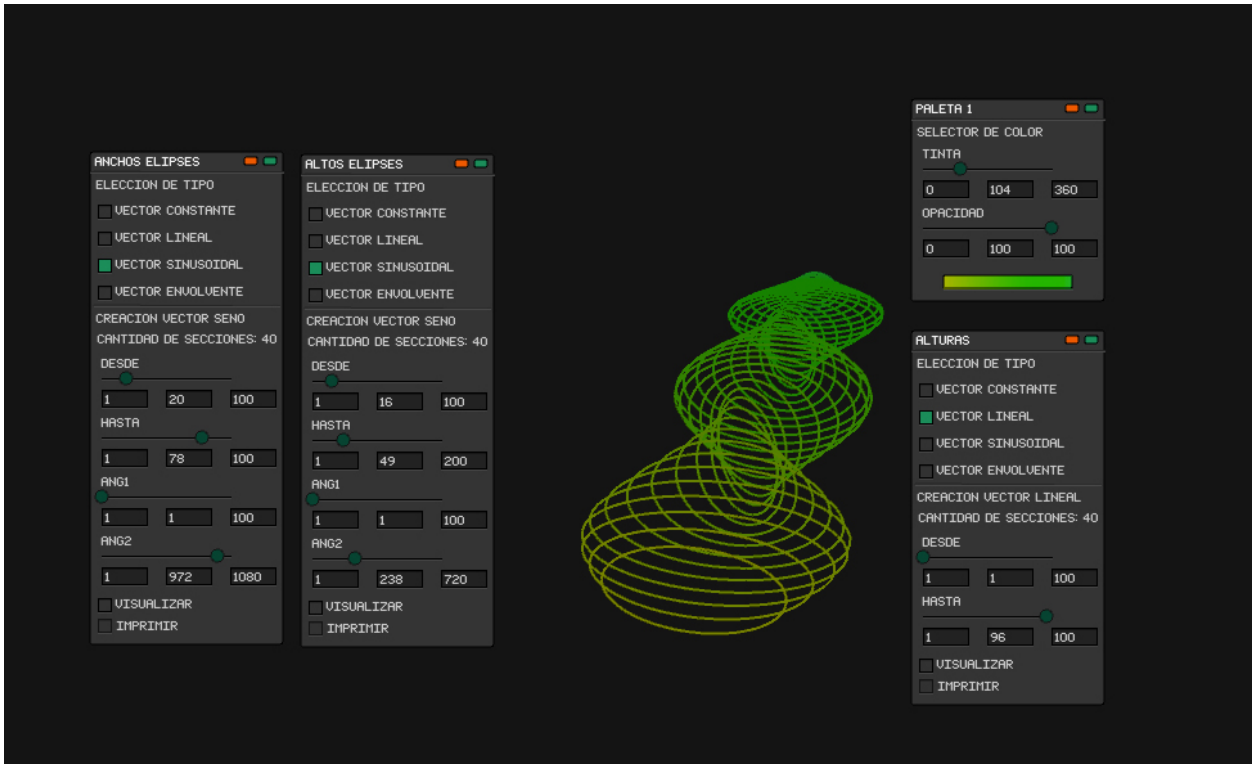
Dado que en la creación del tipo de volumen que planteamos hay varios parámetros y vectores de control en juego(esto es, un volumen está definido por varios vectores, incluso puede estar definido a partir de la operación de varios cuerpos), el tener que cambiar el código cada vez que quiera modificarse una variable resulta engorroso.

Con esto en cuenta, se desarrolló una aplicación de software, denominada *extremoForm* que permite el control de la forma del cuerpo, mediante la implementación de módulos de interfaz gráfica. Estos módulos permiten la configuración detallada de un vector numérico (de los cuatros tipos preestablecidos detallados anteriormente). Al mismo tiempo, la aplicación grafica en tiempo real el efecto que los cambios que dicha configuración provoca sobre el desarrollo del volúmen diseñado.

La siguiente figura muestra un módulo de interfaz gráfica, cuyo nombre asignado (en este caso) es “anchos elipses”. La instancia de la izquierda muestra el módulo en su estado inicial, en el que permite elegir un tipo de vector. La instancia del centro muestra las opciones de configuración para un vector del tipo sinusoidal (nótese que todos sus parámetros pueden ser modificados). La instancia de la derecha muestra las opciones de configuración para un vector del tipo lineal. Es importante notar que se puede saltar de un tipo de vector a otro en tiempo real.



La siguiente figura muestra un volúmen diseñado con esta estrategia. Todos los parámetros que definen su forma (menos las posiciones de los elipses en el eje z) están siendo controlados a través de objetos de interfaz: el ancho de los elipses responde a una progresión sinusoidal, al igual que el alto de los mismos (aunque no a la misma función sinusoidal), mientras que la altura de los elipses responde a una progresión lineal. El módulo de interfaz “paleta 1” es auxiliar y permite controlar la paleta de colores con la que se visualiza el volumen. La gran ventaja de la implementación de los módulos de interfaz es que conviven con la visualización del volumen, por lo tanto, cualquier modificación de parámetros (que a su vez hace efecto de manera instantánea) puede realizarse con la visualización como referencia, facilitando de gran manera el proceso de diseño.



Estos módulos de interfaz gráfica pueden ser reemplazados completamente por código. Una vez que un vector ha sido configurado, el módulo de interface que lo controla puede generar el código equivalente a dicha configuración. Este código puede implementarse en el código mismo de la aplicación, y así liberar espacio en el entorno de la misma, para luego configurar otros parámetros a través de nuevos módulos de interfaz gráfica.

De esta forma, podemos entender al software *extremoForm* como un híbrido entre configuración por código y configuración por interfaz gráfica.

En cuanto a la estructura del código, todo lo relacionado con el módulo de interfaz gráfica se encuentra encapsulado dentro de un objeto llamado *interfaz.* Este mismo objeto a su vez contiene el vector numérico que emerge de su propia configuración. Dentro de este objeto interfaz, se encuentra una serie de objetos del tipo *menú* (hay uno específico para cada tipo de vector, más algunos auxiliares), que a su vez contienen los objetos de interfaz gráfica necesarios (sliders, checkboxes, campos de texto, botones, etc). Al mismo tiempo, el objeto interfaz posee métodos para exportar el código equivalente al vector configurado y para visualizar el mismo.

Resultaba importante que la implementación de estos objetos de interfaz sea lo más sencilla posible (de forma tal de poder reemplazarlos por su código equivalente de manera rápida), por lo que toda la funcionalidad del objeto interfaz está controlada por un sólo método. Este

método devuelve el vector configurado a través de la interfaz. Su implementación es la siguiente:

```
float vector[] = interfaz.procesar("NOMBRE INTERFAZ", numero_elementos);
```

De esta forma, el método *procesar* (cuyos argumentos son el nombre de la interfaz (dato únicamente de referencia para el usuario) y el número de elementos que tendrá el vector devuelto), se encarga de mostrar la interfaz en la pantalla, procesar las cambios realizados en la misma y reflejarlos en el vector que devuelve. Esta estrategia permite que al momento de reemplazar la interfaz por su código equivalente, tan solo haya que reemplazar una línea de código.

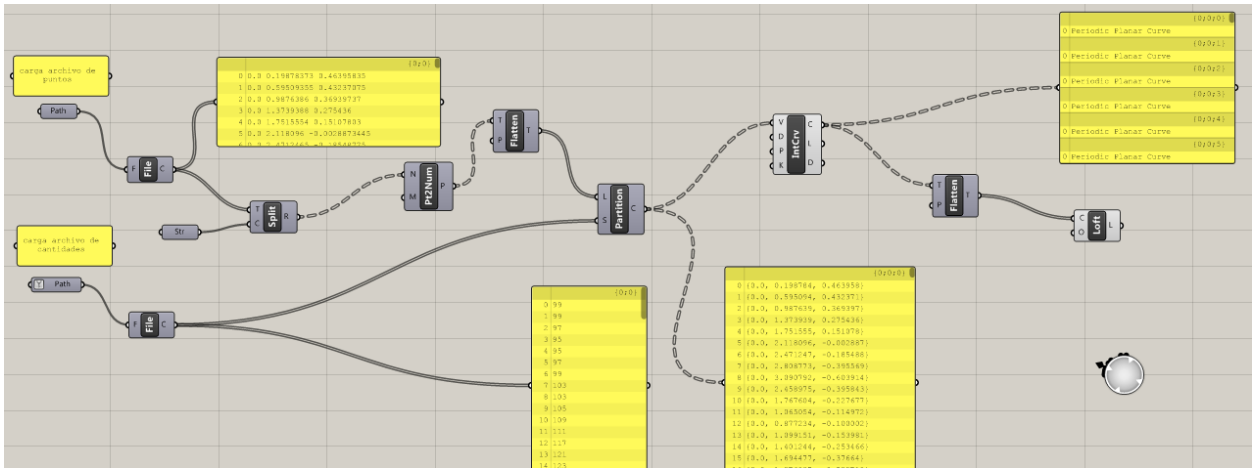
1.5 El algoritmo de traducción a Grasshopper

En esta etapa del proceso, contábamos con la implementación de una serie de clases y funciones para abordar la producción de formas tridimensionales, así como con procedimientos para vincular estas con una interfaz gráfica (GUI). Como una forma de potenciar los logros obtenidos, decidimos realizar un procedimiento para poder exportar las formas generadas con nuestra aplicación (en Processing) a Grasshopper (en Rhinoceros), y de esta forma, poder aprovechar las ventajas de este último.
La estrategia para realizar dicho procedimiento consistió exportar los puntos generados en las costillas, para reproducir dichas costillas en Grasshoper. Para realizar esto se implementó un método que escribe un archivo de texto una secuencia de tripletes de números, donde cada terna representa las coordenadas de un punto en el espacio tridimensional. Así el cuerpo se traduce como una lista de números. A continuación se puede ver una muestra del formato del archivo con los tripletes de números que representan los puntos en el espacio 3D:

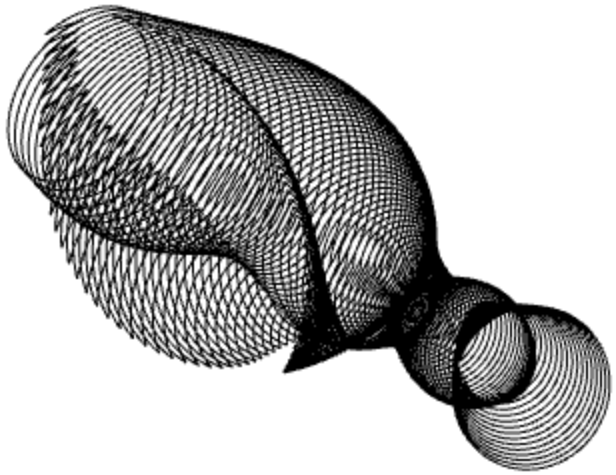
```
0.0 -2.1855695E-7 -10.0
0.0 -0.39725077 -9.984195
0.0 -0.7919904 -9.936877
0.0 -1.1817222 -9.858347
0.0 -1.5639833 -9.749101
0.0 -1.9363558 -9.60983
…
```

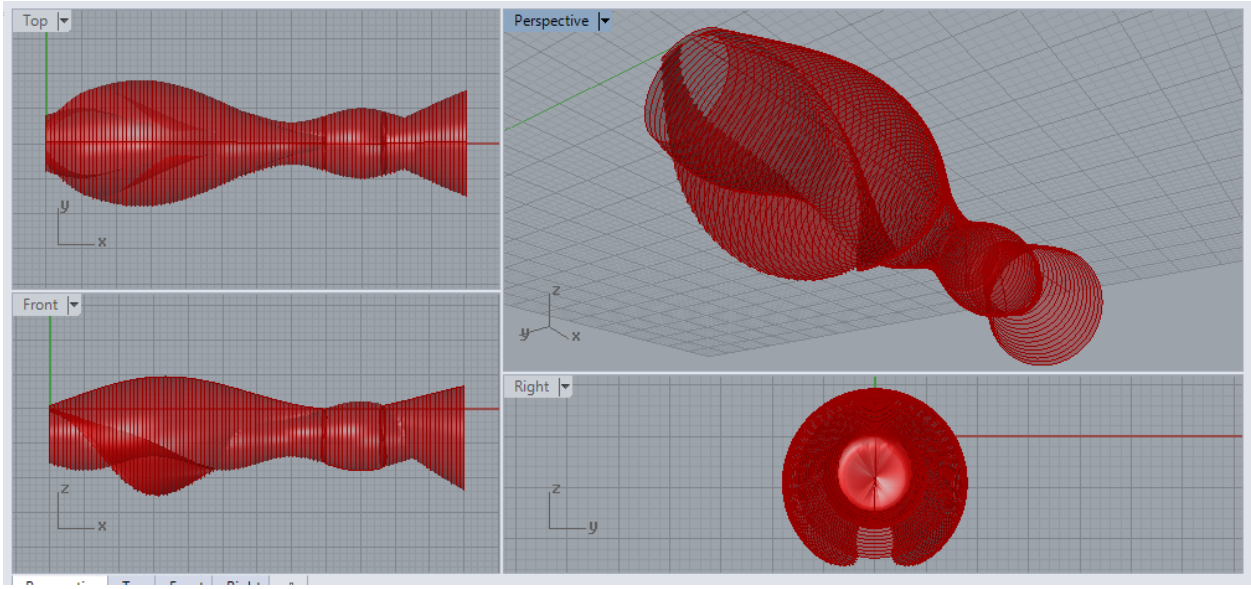
En el algoritmo de abajo (que es el receptor de los datos en Grasshopper) se puede observar dos objetos “File”, el de la parte superior abre el archivo con la lista de puntos. Este objeto se conecta con uno del tipo “Split” que divide los tripletes que luego son enviados a un objeto “Flatten” y posteriormente a “Pt2Num” que transforma estos tripletes en puntos en el espacio. En esta etapa los datos se transformaron en puntos. Posteriormente un objeto “Partition” se

encarga de separar esos puntos en grupos que luego serán transformados en curvas con el objeto “IntCrv”, luego de este punto tenemos una serie de curvas. Por último, un objeto “Loft” construye un superficie que representa el volumen diseñado en Processing.



Una parte importante del proceso es la separación en grupos que hace el objeto “Partition”, en esta separación en grupos determina cuáles puntos pertenecen a cuál curva. Este objeto obtiene datos de un segundo archivo que detalla cuántos puntos tiene cada una de las curvas.
Debajo se puede observar un cuerpo diseñado en la aplicación de Processing y su traducción en los subsiguientes gráficos a Rhinoceros usando el algoritmo de GrassHopper.



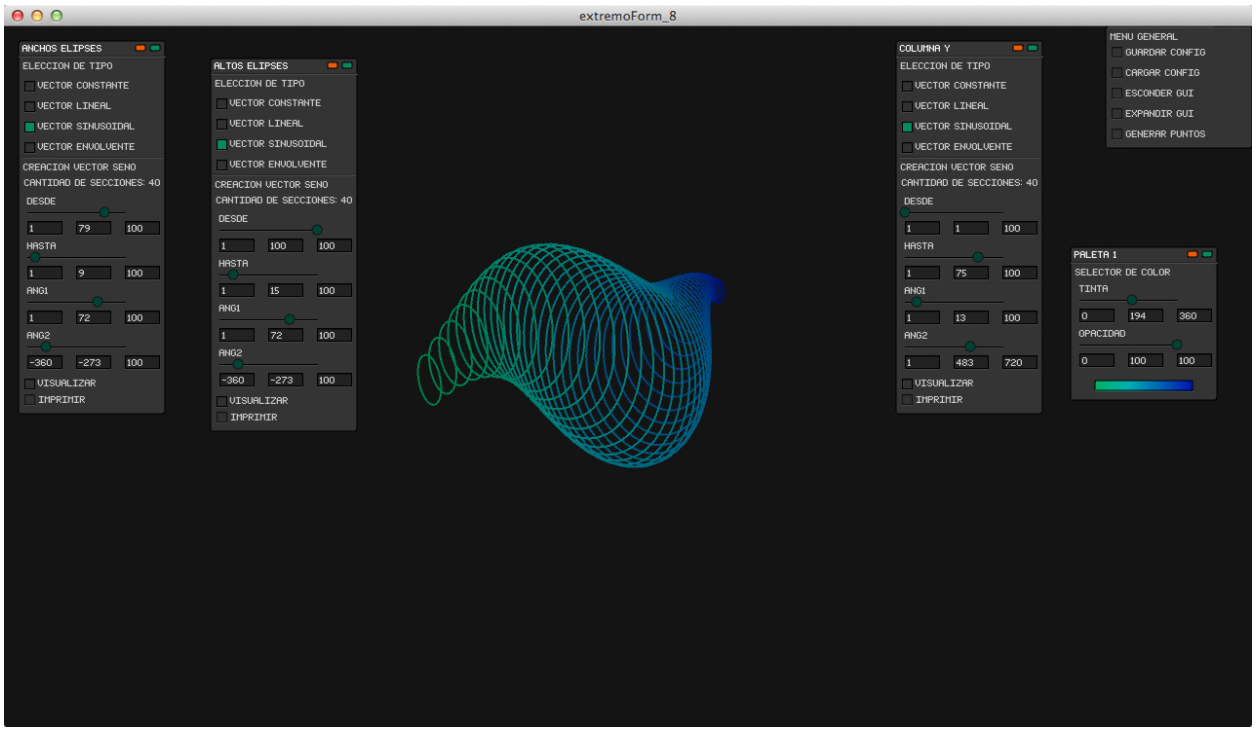


El traspaso del cuerpo de nuestra aplicación desarrollada en Processing a Grasshopper responde a que en éste pueden realizarse modificaciones pertinentes a la fabricación del volúmen, como será explicado más adelante.

1.6 El diseño de la forma mediante el uso de la herramienta de diseño

A continuación se detalla un ejemplo del flujo de trabajo posible con el software desarrollado:

La siguiente figura muestra el entorno gráfico de *extremoForm*, con un posible diseño de un cuerpo. En este caso particular, la configuración del mismo está generada a partir de tres módulos de interfaz gráfica (o GUI, según sus siglas en inglés), que controlan el ancho, el alto y el desplazamiento en el eje Y de los elipses que lo componen, además de dos vectores definidos directamente en el código, que dictan la distancia entre elipses (o sus posiciones en el eje Z) y su desplazamiento en el eje X.



El siguiente fragmento muestra el código necesario para lograr la configuración visible en la figura anterior. Nótese que los objetos **interfaz** fueron organizados en un arreglo (en este caso denominado **in**) para facilitar su control.

```
Cuerpo k;
int cuantos = 40;
k = new Cuerpo( cuantos );
float w[] = in[0].procesar(“ANCHOS ELIPSES”, cuantos);
float h[] = in[0].procesar(“ALTOS ELIPSES”, cuantos);
float px[] = vecConst(cuantos, 0);
float py[] = in[0].procesar(“COLUMNA Y”, cuantos);
float z[] = vecLinea(cuantos, 0, -500);
k.setElipses( w, h );
k.setColumna( px, py, z );
k.dibujar(); // este es un método propio de la clase Cuerpo, se ocupa de
 dibujarlo en pantalla
```

Como puede observarse, cada módulo de GUI posee un botón denominado “IMPRIMIR”. El mismo copia al portapapeles del sistema operativo el código equivalente al vector creado a partir de la actual configuración del módulo. Una vez adquiridos los códigos equivalentes para todos los vectores, el fragmento de código necesario para la descripción completa de este volumen es el siguiente:

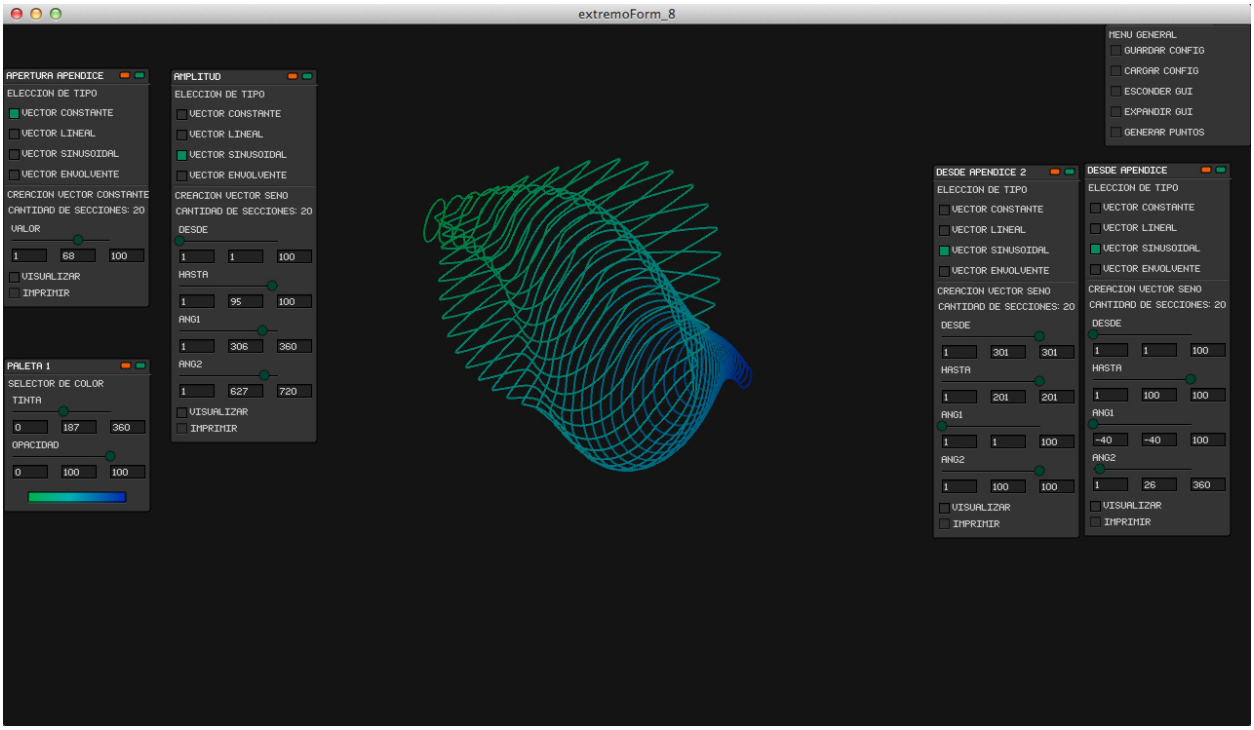
```
Cuerpo k;
```

```
int cuantos = 40;
k = new Cuerpo( cuantos );
float w[] = vecSeno(40, 79, 9, radians(72), radians(-273));
float h[] = vecSeno(40, 100, 15, radians(72), radians(-273));
float px[] = vecConst(cuantos, 0);
float py[] = vecSeno(40, 1, 75, radians(13), radians(483));
float z[] = vecLinea(cuantos, 0, -500);
k.setElipses( w, h );
k.setColumna( px, py, z );
k.dibujar(); // este es un método propio de la clase Cuerpo, se ocupa de
 dibujarlo en pantalla
```

Donde los arreglos *w, h* y *py* fueron generados a partir de módulos de GUI, y los demás arreglos (*px* y *z*) fueron generados previamente en el código.

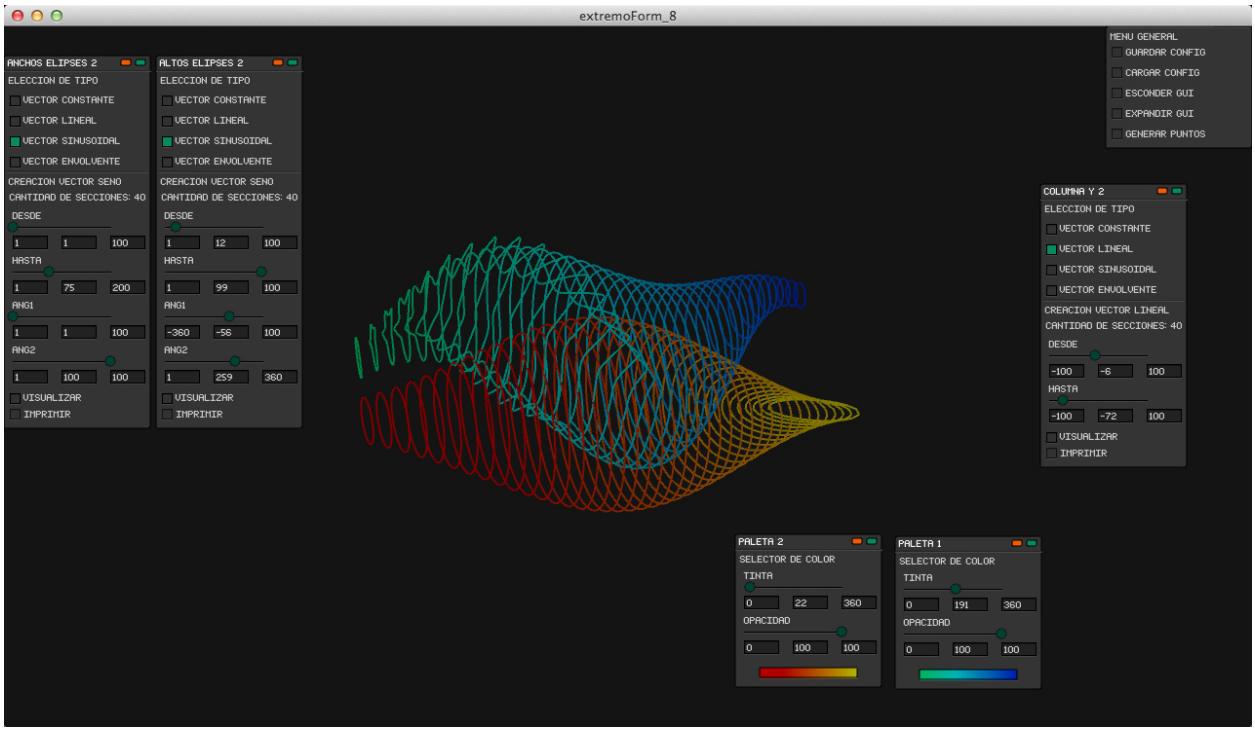
Dados que los algoritmos desarrollados para el modelado del volumen permiten la adición de apéndices al cuerpo diseñado y también la intersección, suma y resta con otros cuerpos; y dado que todas estas operaciones respetan una lógica de funcionamiento basada en arreglos numéricos (y que por lo tanto pueden ser controladas mediante los módulos de GUI de *extremoForm*), resulta de gran importancia la habilidad de poder reemplazar dichos módulos por su código equivalente, de forma tal de liberar espacio en pantalla para poder implementar nuevos módulos de interfaz.

En la siguiente figura pueden verse el mismo cuerpo, pero con la adición de dos “apéndices”, en forma de aletas. Todos los módulos de GUI implementados aquí están relacionados con parámetros de control de los apéndices.

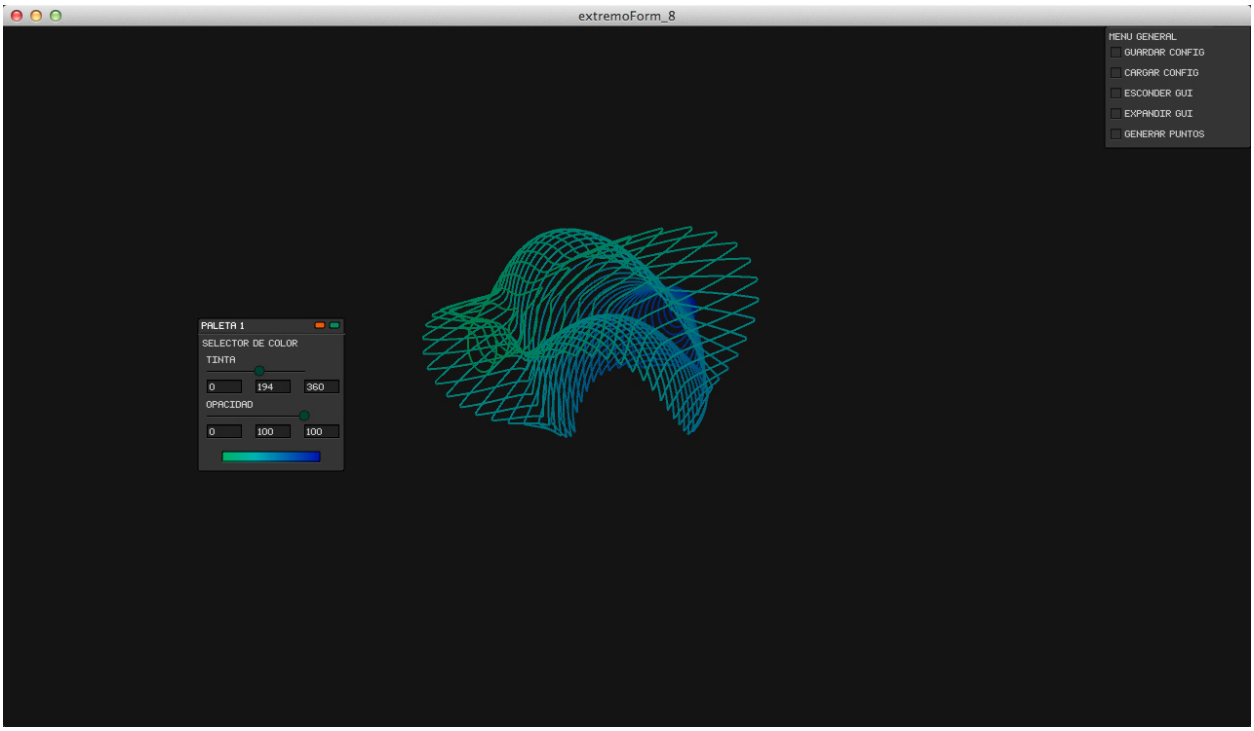


Estos módulos fueron nuevamente reemplazados por sus equivalentes en código, a fin de liberar espacio en el entorno para nuevas operaciones de configuración.

En la siguiente figura, se ha añadido un segundo “cuerpo”, y todos los módulos de GUI implementados están vinculados a la definición de la forma del mismo. Todos los parámetros relacionados con la forma del cuerpo inicial han sido reemplazados por código.



Como fue mencionado previamente, es posible “operar” dos cuerpos entre sí. En este caso, el segundo cuerpo (graficado en rojo y amarillo en la figura anterior), es sustraído del primer cuerpo (graficado en verde y azul). La siguiente figura muestra el resultado de la operación, junto con todos los módulos de GUI convertidos a sus equivalentes en código. De esta forma, el cuerpo en su estado actual se encuentra íntegramente definido a través de código, permitiendo la utilización de nuevos módulos de interfaz para nuevas operaciones (es posible agregar nuevos apéndices, nuevos cuerpos para operar, etc).



Por último, *extremoForm* tiene la capacidad de exportar la forma del cuerpo diseñado de forma tal de poder ser importado por otras aplicaciones de software, como ya fue explicado en la sección anterior, a fin de continuar su configuración y preparlo para su fabricación.

PARTE II: La construcción y motorización de la forma

2.1 Los servos como forma de animación/motorización.

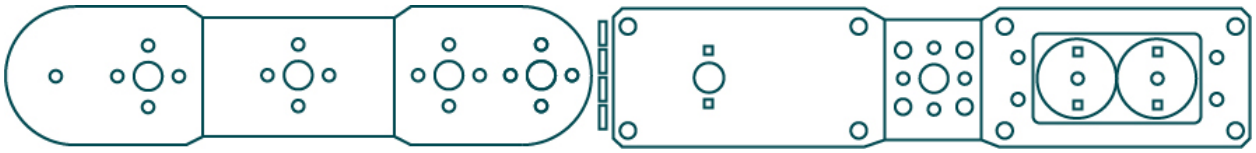
Producir piezas de arte robótico (como la que se busca lograr con este proyecto) requiere el control preciso de la posición y el movimiento de piezas y mecanismos. Para este propósito, es conveniente la utilización de servomotores. Los servomotores son motores de alto torque que poseen un sensor de rotación acoplado a su eje. Generalmente están limitados a un rango de movimiento de 180 grados, por lo que (gracias al sensor de rotación) es posible conocer el ángulo de rotación del eje en todo momento. Esto permite un control muy preciso de la posición de elementos que estén acoplados al eje del servomotor (por ejemplo, una de las aplicaciones más comunes de este tipo de motores es el control de alerones en aeromodelismo). Al mismo tiempo, el acoplamiento de varios servomotores permite crear movimientos complejos con mucha exactitud. Otra ventaja de los servomotores es su relativo bajo costo (en comparación a tecnologías como actuadores neumáticos, por ejemplo). Además, existen en un gran rango de tamaños y capacidades de torque.

Todas estas características hacen que los servomotores sean ideales para el desarrollo de piezas artísticas robotizadas.

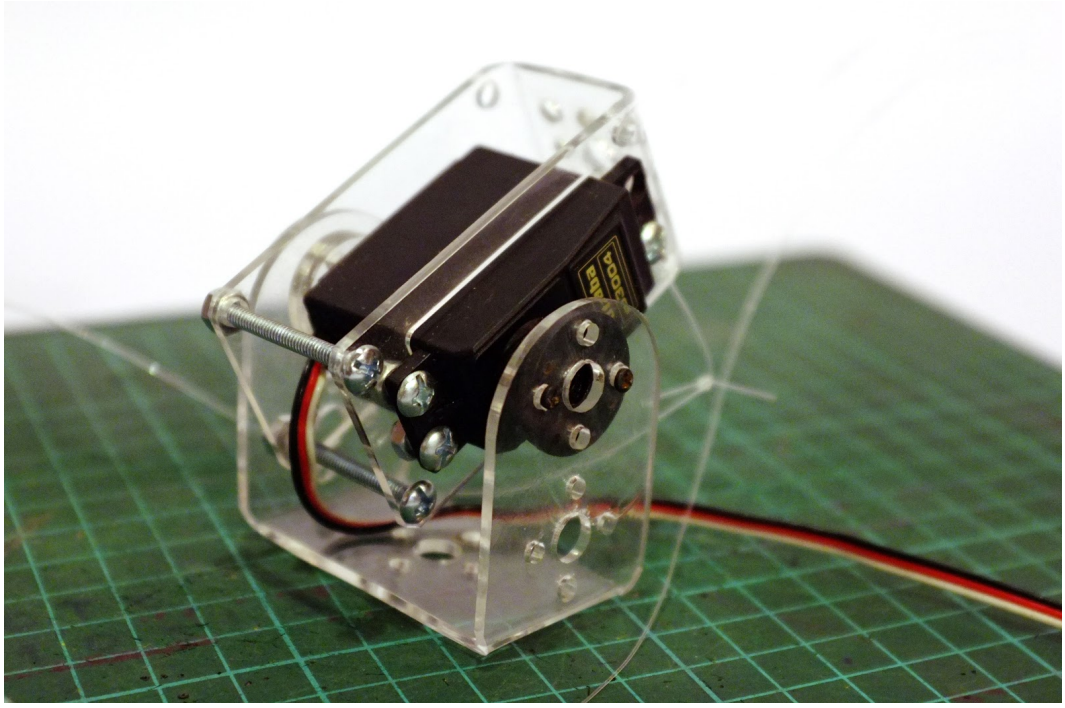
2.2 El diseño de la rótulas

En función de lo anterior, y a fines de poder dotar de movimiento a un cuerpo generado por nuestro software, fue necesario el desarrollo de una articulación robotizada. La misma permite la vinculación de dos piezas a un servomotor, permitiendo un giro de 180° de una pieza sobre otra.

La articulación está compuesta de dos piezas de acrílico. El diseño de la misma fue realizado digitalmente, utilizando una aplicación de dibujo vectorial, teniendo en cuenta las medidas exactas de los servomotores que luego serían utilizados. La siguiente figura muestra la totalidad de las piezas de acrílico necesarias para fabricar una articulación.

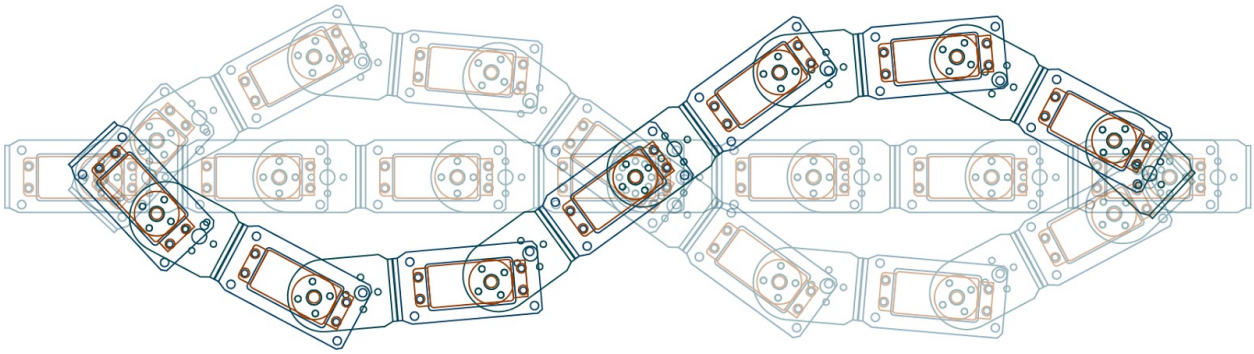


Las piezas fueron diseñadas de forma tal de poder ser cortadas en una lámina de acrílico de 2mm de espesor, mediante una máquina de corte láser. Una vez cortadas, las mismas fueron plegadas mediante un proceso de termomoldeado. Para montar las piezas al servomotor se utilizaron bulones con tuercas y sellador de roscas anaeróbico (este último impide el desajuste de las piezas por consecuencia de las vibraciones de los motores). La siguiente figura muestra una articulación completa, montada sobre un servomotor.



La articulación está preparada para ser acoplada tanto a otras piezas mecánicas o estructurales, como a otros servomotores (para lograr movimiento en dos ejes, por ejemplo), como también a otras articulaciones del mismo tipo. Para todo esto, las piezas cuentan con perforaciones de montaje que respetan las dimensiones estándar de ejes de servomotores. Al mismo tiempo, hay perforaciones en todas las caras de las piezas, de forma tal de proveer la mayor cantidad de posiciones de montaje posibles.

En el caso particular de este proyecto, se realizaron 7 articulaciones, y se montaron en línea (una con el extremo de otra), de forma tal de lograr una configuración similar a una “columna vertebral” articulada para el cuerpo de nuestro “Consumófago”, como puede observarse en la siguiente figura.



En este caso particular, las costillas diseñadas con nuestro software y exportadas desde grasshopper irían montadas entre las articulaciones.

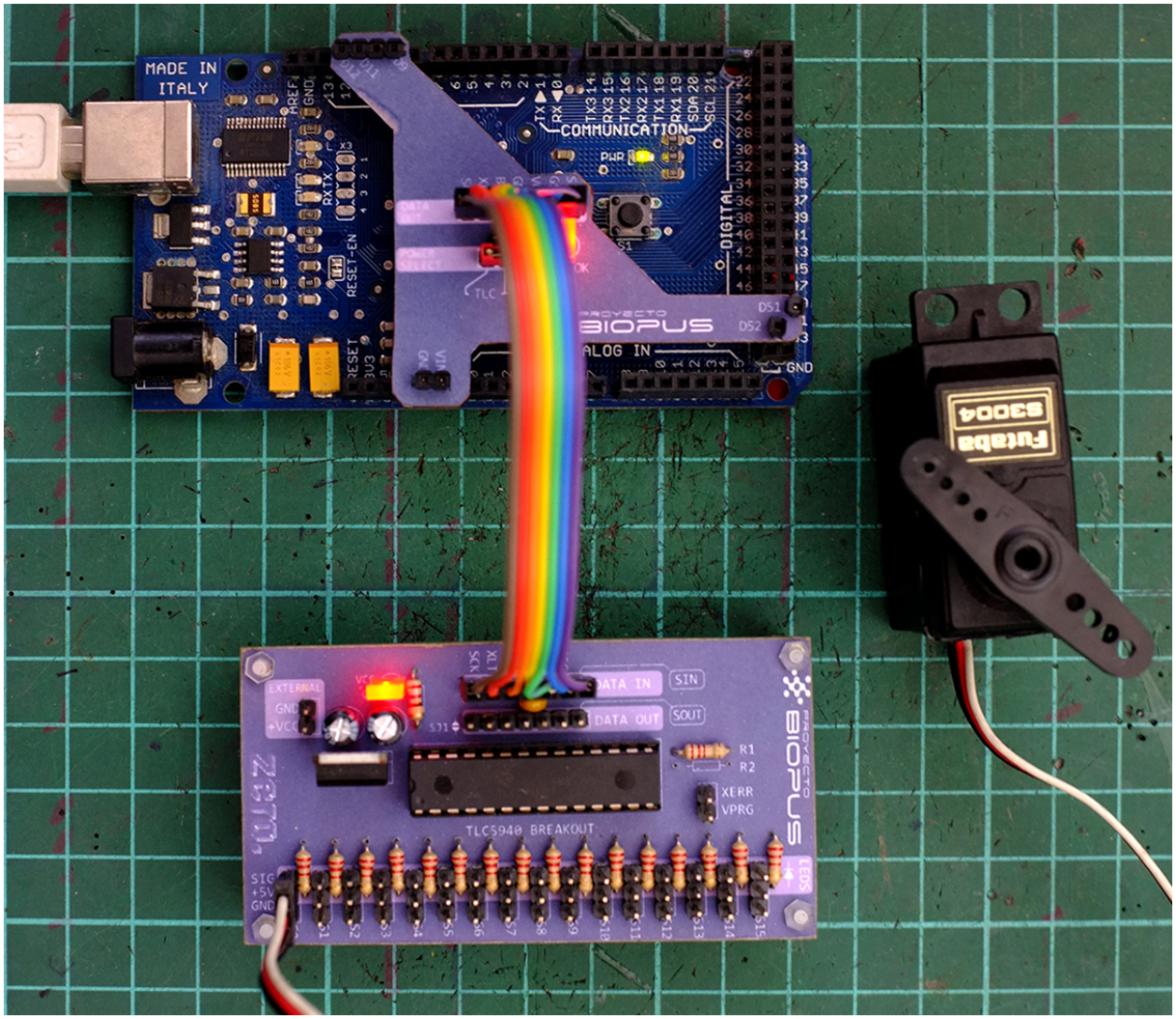
2.3 El diseño de la electrónica de control

Para poder controlar todos los servomotres de la columna descrita anteriormente, se requiere requiere de la utilización de un microcontrolador (al menos, esto resulta lo más práctico y versátil). La plataforma de microcontrolador elegida para este proyecto fue *Arduino*, una plataforma de código y hardware abierto. Dado que la columna vertebral desarrollada cuenta con varios puntos de articulación, y que estos deben poder ser operados de manera simultánera, se hace necesaria la utilización de un elevado número de puertos de salida/entrada de la placa Arduino, que podrían ser útiles para controlar otro tipo de actuadores o sensores (luces, dispositivos de sonido, sensores de proximidad, etc). Por esto, se buscó una solución que permita el control de un gran número de servomotores sin comprometer todos los puertos de salida/entrada de la placa Arduino. Para esto se desarrolló un circuito electrónico basado en el chip driver de modulación por ancho de pulso (PWM, según sus siglas en inglés) de 16 canales TLC5940. El circuito fue basado en un diseño de hardware abierto (disponible en https://www.sparkfun.com/products/10616), y el mismo fue fabricado como una placa de circuito impreso.

Esta placa permite el control de 16 canales (a los cuales pueden conectarse servomotores, LEDs o cualquier dispositivo que requiera señal PWM), con tan solo 5 puertos digitales de entrada/salida del Arduino (sin la utilización de esta placa, cada servomotor requeriría 3 puertos de entrada/salida para ser controlado directamente, por lo que para controlar 16 servomotores sin utilizar electrónica adicional, serían necesarios 48 puertos de entrada/salida). Al mismo tiempo, estas placas pueden ser conectadas secuencialmente, hasta un límite de aproximadamente 40 placas (esto significa aproximadamente 640 canales de PWM), sin la utilización de puertos adicionales.

Se fabricaron tres placas (si bien este caso particular solo necesita de una), más una placa adaptadora que permite la fácil conexión del circuito a una placa Arduino modelo Mega.

La siguiente figura muestra una placa Arduino Mega, conectada mediante el adaptador a una de las placas construidas, con un servomotor conectado al primero de sus 16 canales de salida.

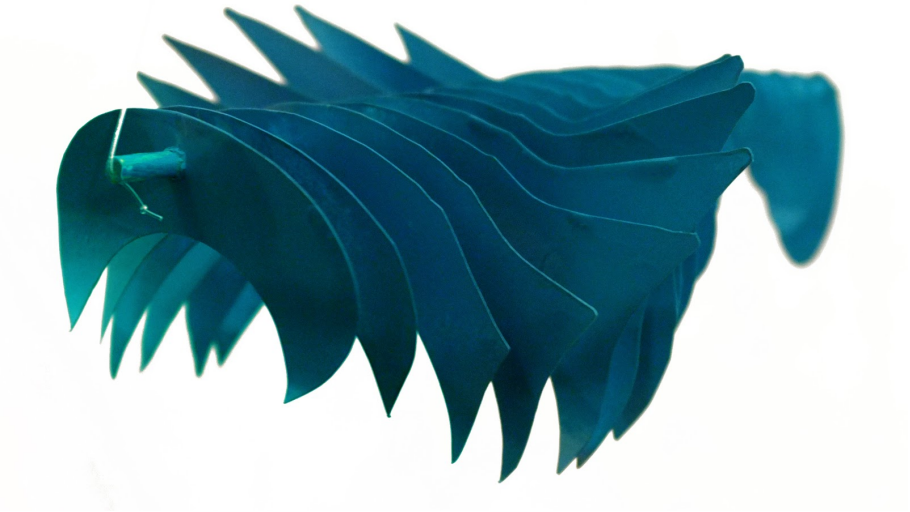


2.4 El corte y ensamblado de la piezas con la estructura de motores

Una vez que diseñamos (utilizando nuestro software, extremoForm) una forma de cuerpo con la que estuvimos conformes, y una vez importadas sus costillas en Grasshopper, se atravesó a todas con un eje de referencia, para asegurar el correcto posicionamiento de las mismas posteriormente.
Las costillas fueron exportadas de Grasshopper como curvas vectoriales e importadas en un software de dibujo vectorial. A partir de aquí, las costillas fueron organizadas para ocupar la

menor cantidad de superficie posible (en función de reducir la cantidad de material necesario para el corte de las mismas).

Una versión a escala de las costillas fue impresa en papel de alto gramaje y montada sobre un eje, de forma tal de lograr una maqueta a escala del volumen resultante del proceso descrito previamente, como puede observarse en la siguiente figura.



De vuelta en el software de dibujo vectorial, se utilizaron los diseños vectoriales para las articulaciones robotizadas, para agregar las perforaciones de montaje de las mismas a las costillas (de forma tal de poder montar las piezas sobre la columna creada con las articulaciones). Las mismas fueron impresas en papel a escala real, montadas sobre cartón y cortadas. Luego fueron montadas en la columna de articulaciones, para realizar pruebas de movimiento del conjunto en su totalidad (es decir, las costillas diseñadas digitalmente, montadas sobre la columna motorizada, controlada con nuestra electrónica).

Una vez comprobado el correcto funcionamiento del sistema, los mismos diseños vectoriales pueden ser utilizados para realizar cortes de las piezas en acrílico, para su ensamblaje final.

Bibliografía

Causa, E (2013) Texto “Algoritmos Genéticos aplicados a la generación y producción de formas escultóricas” en “Invasión Generativa. Fronteras de la generatividad en las tres dimensiones, la robótica y la realidad aumentada.”
Autores: Compilador: Emiliano Causa. Autores: Francisco Alvarez Lojo, David Bedoian, Paula Castillo, Emiliano Causa, Joaquin Ibarlucia, Federico Joselevich Puiggros, Ezequiel Rivero, Christian Silva, Leonardo Solaas, Ariel Uzal
Editorial: Editorial Invasores de la Generatividad Año: 2013 - 2014
ISSN: 2362-3381

GcodeTools - http://kalyaev.com/2010/20100423/gcodetools.html

Arduino - http://www.arduino.cc/

Grasshopper, algorithmic modeling for rhino -http://www.grasshopper3d.com/