

# Voc

## A vocal tract physical model implementation.

By Paul Batchelor

Git Hash: 6a0996c2da5c26e5ffc7896620f7c020fb4ef4af



# Introduction

The following document describes *Voc*, an implementation of a vocal tract physical model.

## Motivations and Goals

The human voice is a powerful tool for any composer, second only to silence. Even an approximation of the voice can tap into the entire range of human emotion. This is why the wind howls, floorboards moan, or R2D2 pouts. For computer musicians and sound designers alike, creating sonic elements with vocal qualities can give cold digital sounds a human-like relatable quality; an excellent tool for engaging an audience.

The goal of *Voc* is to provide a low level model for producing utterances and phonemes. It will neither attempt to sing or talk, but it will babble and chatter. A program which is closely aligned with *Voc*'s scope is Neil Thapen's web application *Pink Trombone*. [pinktrombone] In this program, vocal phonemes are generated through directly manipulating a virtual vocal tract in continuous time.

## Literate Programming

As an experiment, the author has decided to use *literate programming* for this project. Literate programming, created by Donald Knuth [knuth1992literate], is the concept of melting documentation and code together. What you are reading is also a program!

The biggest advantage of using literate programming for this project is the ability to use mathematical notation to describe concepts that are implemented. The C-language does not lend itself well for comprehensibility when it comes to DSP, even with comments. Nobody ever learned about DSP from C code alone! A very successful example of literate programming is the book *Physically Based Rendering* [pbrt], which is both a textbook and software implementation of a physically accurate ray tracer.

The underlying technology used here is CWEB, the definitive literate programming tool developed by Donald Knuth, with some minor macro adjustments for formatting.

# Overview <sup>(1)</sup>

In a literate program, it is customary (and somewhat mandatory) to provide an "overview" section. This section serves as the entry point in generating the C amalgamation file *voc.c*. Complying with the constraints of CWEB, the corresponding sections will appear at the bottom of this section.

## The Core Voc Components

⟨Headers 3⟩ is the header section of the C file (not be confused with the separate header file ⟨*voc.h* 55⟩). This is where all the system includes, macros, global data, and structs are declared.

⟨The Glottis 28⟩ is the component of Voc concerned with producing the glottal excitation signal.

⟨The Vocal Tract 40⟩ is implementation of the physical waveguide of the vocal tract.

⟨Top Level Functions 8⟩ is the section consisting of all public functions for controlling Voc, from instantiation to parametric control.

## Supplementary Files

In addition to the main C amalgamation, there are a few other files that this literate program generates:

⟨*debug.c* 57⟩ is the debug utility used extensively through out the development of Voc, used to debug and test out features.

⟨*voc.h* 55⟩ is the user-facing header file that goes along with the C amalgamation. Anyone wishing to use this program will need this header file along with the C file.

⟨*plot.c* 58⟩ is a program that generates dat files, which can then be fed into gnuplot for plotting. It is used to generate the plots you see in this document.

⟨*ugen.c* 59⟩ provides an implementation of Voc as a Sporth unit generator, offering 5 dimensions of control. In addition the main Sporth plugin, there are also smaller unit generators implementing portions of Voc, such as the vocal tract filter.

⟨Headers 3⟩

⟨The Glottis 28⟩

⟨The Vocal Tract 40⟩

⟨Top Level Functions 8⟩

# Header Inclusion, Structs, and Macros <sup>(2)</sup>

## 3.

### Header of File

The header section consists of header inclusion, and definition of C-structs. The system-wide header files include *stdlib.h* for things like *malloc()*. Standard math library functions from *math.h* are used. Soundpipe/Sporth specific header files are *soundpipe.h* and *sporth.h*. It should be noted that due to the implementation of Sporth, the Soundpipe header file *must* be included before the Sporth header file.

ANSI C doesn't have the constant `M_PI`, so it has to be explicitly defined.

Both `MIN` and `MAX` macros are defined.

The header file *string.h* is included so that *memset* can be used to zero arrays.

There is exactly one local header file called *voc.h*, which is generated by CTANGLE. For more information about this header file, see `<voc.h 55>`

```
<Headers 3> ≡
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <soundpipe.h>
#include <sporth.h>
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif
#include "voc.h"
#ifndef MIN
#define MIN(A, B) ((A) < (B) ? (A) : (B))
#endif
#ifndef MAX
#define MAX(A, B) ((A) > (B) ? (A) : (B))
#endif
#define EPSILON 1.0 · 10-38
<Data Structures and C Structs 4>
```

This code is cited in section 1.

This code is used in section 1.

## 4.

### Structs

This subsection contains all the data structs needed by Voc.

```
<Data Structures and C Structs 4> ≡
<Glottis Data Structure 6>
<Tract Data 7>
<Voc Main Data Struct 5>
```

This code is used in section 3.

5. The top-most data structure is *sp\_voc*, designed to be an opaque struct containing all the variables needed for *Voc* to work. Like all Soundpipe modules, this struct has the prefix "sp".

⟨Voc Main Data Struct 5⟩ ≡

```
struct sp_voc {
    glottis glot;    /* The Glottis */
    tract tr;        /* The Vocal Tract */
    SPFLOAT buf[512];
    int counter;
};
```

This code is used in section 4.

6. The glottis data structure contains all the variables used by the glottis. See ⟨The Glottis 28⟩ to see the implementation of the glottal sound source.

- *freq* is the frequency
- *tenseness* is the tenseness of the glottis (more or less looks like a cross fade between voiced and unvoiced sound). It is a value in the range [0, 1].
- *Rd*
- *waveform\_length* provides the period length (in seconds) of the fundamental frequency, in seconds.
- The waveform position is kept track of in *time\_in\_waveform*, in seconds.
- *alpha*
- *EO*
- *epsilon*
- *shift*
- *delta*
- *Te*
- *omega*
- *T*

⟨Glottis Data Structure 6⟩ ≡

```
typedef struct {
    SPFLOAT freq;
    SPFLOAT tenseness;
    SPFLOAT Rd;
    SPFLOAT waveform_length;
    SPFLOAT time_in_waveform;
    SPFLOAT alpha;
    SPFLOAT EO;
    SPFLOAT epsilon;
    SPFLOAT shift;
    SPFLOAT delta;
    SPFLOAT Te;
    SPFLOAT omega;
    SPFLOAT T;
} glottis;
```

This code is used in section 4.

7. The Tract C struct contains all the data needed for the vocal tract filter.

⟨Tract Data 7⟩ ≡

```
typedef struct {
    int n;    n is the size, set to 44.
    SPFLOAT diameter[44];
    SPFLOAT rest_diameter[44];
    SPFLOAT target_diameter[44];
    SPFLOAT new_diameter[44];
    SPFLOAT R[44];    component going right
    SPFLOAT L[44];    component going left
    SPFLOAT reflection[45];
    SPFLOAT new_reflection[45];
    SPFLOAT junction_outL[45];
    SPFLOAT junction_outR[45];
    SPFLOAT A[44];
    SPFLOAT max_amplitude[44];
    int nose_length;    The original code here has it at floor(28 * n/44), and since n=44, it should be 28.
    int nose_start;    n - nose_length + 1, or 17
    tip_start is a constant set to 32
    int tip_start;
    SPFLOAT noseL[28];
    SPFLOAT noseR[28];
    SPFLOAT nose_junc_outL[29];
    SPFLOAT nose_junc_outR[29];
    SPFLOAT nose_reflection[29];
    SPFLOAT nose_diameter[28];
    SPFLOAT noseA[28];
    SPFLOAT nose_max_amp[28];
    SPFLOAT reflection_left;
    SPFLOAT reflection_right;
    SPFLOAT reflection_nose;
    SPFLOAT new_reflection_left;
    SPFLOAT new_reflection_right;
    SPFLOAT new_reflection_nose;
    SPFLOAT velum_target;
    SPFLOAT glottal_reflection;
    SPFLOAT lip_reflection;
    SPFLOAT last_obstruction;
    SPFLOAT fade;
    SPFLOAT movement_speed;
    15 cm/s SPFLOAT lip_output;
    SPFLOAT nose_output;
    SPFLOAT block_time;
} tract;
```

This code is used in section 4.

## Top-level Functions <sup>(8)</sup>

Broadly speaking, the top-level functions are in charge of computing samples for the DSP inner-loop before, after, and during runtime. They get their name from the fact that they are the top level of abstraction in the program. These are the functions that get called in the Sporth Unit Generator implementation (`ugen.c` <sup>59</sup>).

```

< Top Level Functions 8 > ≡
  < Voc Create 9 >
  < Voc Destroy 10 >
  < Voc Initialization 11 >
  < Voc Compute 12 >
  < Voc Tract Compute 13 >
  < Voc Set Frequency 14 >
  < Voc Get Frequency 15 >
  < Voc Get Tract Diameters 16 >
  < Voc Get Current Tract Diameters 17 >
  < Voc Get Tract Size 18 >
  < Voc Get Nose Diameters 19 >
  < Voc Get Nose Size 20 >
  < Voc Set Diameters 21 >
  < Voc Set Tongue Shape 22 >
  < Voc Get Counter 23 >
  < Voc Set Tenseness 24 >
  < Voc Get Tenseness 25 >
  < Voc Set Velum 26 >
  < Voc Get Velum 27 >

```

This code is cited in sections [1](#) and [57](#).

This code is used in section [1](#).

**9.** In the function `sp_voc_create`, an instance of `Voc` is created via `malloc`.

```

< Voc Create 9 > ≡
  int sp_voc_create(sp_voc **voc)
  {
    *voc = malloc(sizeof(sp_voc));
    return SP_OK;
  }

```

This code is cited in sections [46](#) and [60](#).

This code is used in section [8](#).

**10.** As a counterpart to `sp_voc_compute`, `sp_voc_destroy` frees all data previous allocated.

```

< Voc Destroy 10 > ≡
  int sp_voc_destroy(sp_voc **voc)
  {
    free(*voc);
    return SP_OK;
  }

```

This code is cited in section [63](#).

This code is used in section [8](#).

11. After data has been allocated with *sp\_voc\_create*, it must be initialized with *sp\_voc\_init*.

⟨ Voc Initialization 11 ⟩ ≡

```
int sp_voc_init(sp_data * sp, sp_voc *voc)
{
    glottis_init(&voc->glot, sp->sr);    /* initialize glottis */
    tract_init(sp, &voc->tr);           /* initialize vocal tract */
    voc->counter = 0;
    return SP_OK;
}
```

This code is cited in section 61.

This code is used in section 8.

12. The function *sp\_voc\_compute* is called during runtime to generate audio. This computation function will generate a single sample of audio and store it in the SPFLOAT pointer *\*out*.

⟨ Voc Compute 12 ⟩ ≡

```
int sp_voc_compute(sp_data * sp, sp_voc *voc, SPFLOAT * out)
{
    SPFLOAT vocal_output, glot;
    SPFLOAT lambda1, lambda2;
    int i;
    if (voc->counter ≡ 0) {
        tract_reshape(&voc->tr);
        tract_calculate_reflections(&voc->tr);
        for (i = 0; i < 512; i++) {
            vocal_output = 0;
            lambda1 = (SPFLOAT)i/512;
            lambda2 = (SPFLOAT)(i + 0.5)/512;
            glot = glottis_compute(sp, &voc->glot, lambda1);
            tract_compute(sp, &voc->tr, glot, lambda1);
            vocal_output += voc->tr.lip_output + voc->tr.nose_output;
            tract_compute(sp, &voc->tr, glot, lambda2);
            vocal_output += voc->tr.lip_output + voc->tr.nose_output;
            voc->buf[i] = vocal_output * 0.125;
        }
    }
    *out = voc->buf[voc->counter];
    voc->counter = (voc->counter + 1) % 512;
    return SP_OK;
}
```

This code is cited in sections 13, 23, and 62.

This code is used in section 8.



**13.** The function *sp\_voc\_compute\_tract* computes the vocal tract component of Voc separately from the glottis. This provides the ability to use any input signal as an glottal excitation, turning the model into a formant filter. Compared to the main implementation in [〈Voc Compute 12〉](#), this function does not have the 512 sample delay.

```

〈Voc Tract Compute 13〉 ≡
int sp_voc_tract_compute(sp_data * sp, sp_voc *voc, SPFLOAT * in, SPFLOAT * out)
{
    SPFLOAT vocal_output;
    SPFLOAT lambda1, lambda2;
    if (voc->counter ≡ 0) {
        tract_reshape(&voc->tr);
        tract_calculate_reflections(&voc->tr);
    }
    vocal_output = 0;
    lambda1 = (SPFLOAT)voc->counter/512;
    lambda2 = (SPFLOAT)(voc->counter + 0.5)/512;
    tract_compute(sp, &voc->tr, *in, lambda1);
    vocal_output += voc->tr.lip_output + voc->tr.nose_output;
    tract_compute(sp, &voc->tr, *in, lambda2);
    vocal_output += voc->tr.lip_output + voc->tr.nose_output;
    *out = vocal_output * 0.125;
    voc->counter = (voc->counter + 1) % 512;
    return SP_OK;
}

```

This code is used in [section 8](#).

**14.** The function *sp\_voc\_set\_frequency* sets the fundamental frequency for the glottal wave.

```

〈Voc Set Frequency 14〉 ≡
void sp_voc_set_frequency(sp_voc *voc, SPFLOAT freq)
{
    voc->glot.freq = freq;
}

```

This code is used in [section 8](#).

**15.** The function *sp\_voc\_get\_frequency\_ptr* returns a pointer to the variable holding the frequency. This allows values to be set and read directly without the use of a helper function. This function was notably created for use in a demo using the GUI library Nuklear.

```

〈Voc Get Frequency 15〉 ≡
SPFLOAT * sp_voc_get_frequency_ptr(sp_voc *voc)
{
    return &voc->glot.freq;
}

```

This code is used in [section 8](#).

**16.** This getter function returns the cylindrical diameters representing tract.

```

⟨ Voc Get Tract Diameters 16 ⟩ ≡
  SPFLOAT * sp_voc_get_tract_diameters(sp_voc *voc)
  {
    return voc->tr.target_diameter;
  }

```

This code is cited in section 17.

This code is used in section 8.

**17.** Similar to *sp\_voc\_get\_tract\_diameters* in ⟨ Voc Get Tract Diameters 16 ⟩, the function *sp\_voc\_get\_current\_tract\_diameters* returns the diameters of the tract. The difference is that this function returns the actual slewed diameters used in ⟨ Reshape Vocal Tract 54 ⟩, rather than the target diameters.

```

⟨ Voc Get Current Tract Diameters 17 ⟩ ≡
  SPFLOAT * sp_voc_get_current_tract_diameters(sp_voc *voc)
  {
    return voc->tr.diameter;
  }

```

This code is used in section 8.

**18.** This getter function returns the size of the vocal tract.

```

⟨ Voc Get Tract Size 18 ⟩ ≡
  int sp_voc_get_tract_size(sp_voc *voc)
  {
    return voc->tr.n;
  }

```

This code is used in section 8.

**19.** This function returns the cylindrical diameters of the nasal cavity.

```

⟨ Voc Get Nose Diameters 19 ⟩ ≡
  SPFLOAT * sp_voc_get_nose_diameters(sp_voc *voc)
  {
    return voc->tr.nose_diameter;
  }

```

This code is used in section 8.

**20.** This function returns the nose size.

```

⟨ Voc Get Nose Size 20 ⟩ ≡
  int sp_voc_get_nose_size(sp_voc *voc)
  {
    return voc->tr.nose_length;
  }

```

This code is used in section 8.

**21.** The function `sp_voc_set_diameter()` is a function adopted from Neil Thapen’s Pink Trombone in a function he called `setRestDiameter`. It is the main function in charge of the ”tongue position” XY control. Modifications to the original function have been made in an attempt to make the function more generalized. Instead of relying on internal state, all variables used are parameters in the function. Because of this fact, there are quite a few function parameters:

- **voc**, the core Voc data struct
- **blade\_start**, index where the blade (?) starts. this is set to 10 in pink trombone
- **lip\_start**, index where lip starts. this constant is set to 39.
- **tip\_start**, this is set to 32.
- **tongue\_index**
- **tongue\_diameter**
- **diameters**, the floating point array to write to

For practical use cases, it is not ideal to call this function directly. Instead, it can be indirectly called using a more sane function `sp_voc_set_tongue_shape()`, found in the section [〈Voc Set Tongue Shape 22〉](#).

〈Voc Set Diameters 21〉  $\equiv$

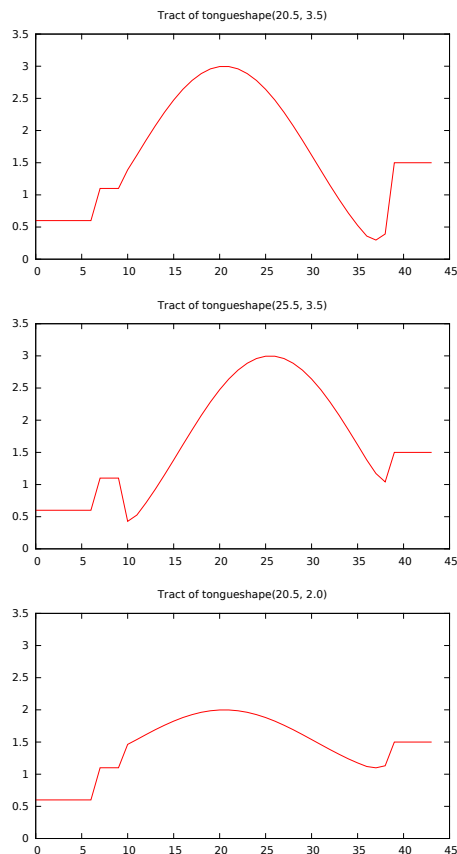
```
void sp_voc_set_diameters(sp_voc *voc,
int blade_start,
int lip_start,
int tip_start,
SPFLOAT tongue_index,
SPFLOAT tongue_diameter,
SPFLOAT *diameters)
{
    int i;
    SPFLOAT t;
    SPFLOAT fixed_tongue_diameter;
    SPFLOAT curve;
    int grid_offset = 0;
    for (i = blade_start; i < lip_start; i++) {
        t = 1.1 * M_PI * (SPFLOAT)(tongue_index - i) / (tip_start - blade_start);
        fixed_tongue_diameter = 2 + (tongue_diameter - 2) / 1.5;
        curve = (1.5 - fixed_tongue_diameter + grid_offset) * cos(t);
        if (i == blade_start - 2 || i == lip_start - 1) curve *= 0.8;
        if (i == blade_start || i == lip_start - 2) curve *= 0.94;
        diameters[i] = 1.5 - curve;
    }
}
```

This code is cited in section [22](#).

This code is used in section [8](#).

**22.** The function `sp_voc_set_tongue_shape()` will set the shape of the tongue using the two primary arguments `tongue_index` and `tongue_diameter`. It is a wrapper around the function described in ⟨Voc Set Diameters 21⟩, filling in the constants used, and thereby making it simpler to work with.

A few tract shapes shaped using this function have been generated below:



⟨Voc Set Tongue Shape 22⟩ ≡

```
void sp_voc_set_tongue_shape(sp_voc *voc, SPFLOAT tongue_index,
SPFLOAT tongue_diameter)
{
    SPFLOAT *diameters;
    diameters = sp_voc_get_tract_diameters(voc);
    sp_voc_set_diameters(voc, 10, 39, 32, tongue_index, tongue_diameter, diameters);
}
```

This code is cited in section 21.

This code is used in section 8.

**23.** Voc keeps an internal counter for control rate operations called inside of the audio-rate compute function in [〈Voc Compute 12〉](#). The function `sp_voc_get_counter()` gets the current counter position. When the counter is 0, the next call to `sp_voc_compute` will compute another block of audio. Getting the counter position before the call allows control-rate variables to be set before then.

```

< Voc Get Counter 23 > ≡
    int sp_voc_get_counter(sp_voc *voc)
    {
        return voc->counter;
    }

```

This code is used in [section 8](#).

**24.** The function `sp_voc_set_tenseness` is used to set the tenseness variable, used when calculating glottal time coefficients in [〈Set up Glottis Waveform 31〉](#), and is the main factor in calculating aspiration noise in [〈Glottis Computation 30〉](#). Typically this is a value between 0 and 1. A value of 1 gives a full vocal sound, while a value of 0 is all breathy. It is ideal to have a little bit of aspiration noise. Empirically good values tend to be in the range of [0.6, 0.9].

```

< Voc Set Tenseness 24 > ≡
    void sp_voc_set_tenseness(sp_voc *voc, SPFLOAT tenseness)
    {
        voc->glot.tenseness = tenseness;
    }

```

This code is used in [section 8](#).

**25.** The function `sp_voc_get_tenseness_ptr` returns an `SPFLOAT` pointer to the parameter value directly controlling tenseness. This function is useful for GUI frontends that use direct pointer manipulation like Nuklear, the cross-platform UI framework used to make a demo for Voc.

```

< Voc Get Tenseness 25 > ≡
    SPFLOAT * sp_voc_get_tenseness_ptr(sp_voc *voc)
    {
        return &voc->glot.tenseness;
    }

```

This code is used in [section 8](#).

**26.** The function `sp_voc_set_velum` sets the *velum*, or soft palette of tract model. In the original implementation, the default value is 0.01, and set to a value of 0.04 to get a nasally sound.

```

< Voc Set Velum 26 > ≡
    void sp_voc_set_velum(sp_voc *voc, SPFLOAT velum)
    {
        voc->tr.velum_target = velum;
    }

```

This code is used in [section 8](#).

**27.** The function `sp_voc_get_velum_ptr` returns the pointer associated with the velum, allowing direct control of the velum parameter. This function was created for use with a demo requiring direct access.

```

< Voc Get Velum 27 > ≡
    SPFLOAT * sp_voc_get_velum_ptr(sp_voc *voc)
    {
        return &voc->tr.velum_target;
    }

```

This code is used in [section 8](#).

## The Glottis (28)

This is where the synthesis of the glottal source signal will be created.

While the implementation comes directly from Pink Trombone’s JavaScript code, it should be noted that the glottal model is based on a modified LF-model[[lu2000glottal](#)].

⟨ The Glottis [28](#) ⟩ ≡  
 ⟨ Set up Glottis Waveform [31](#) ⟩  
 ⟨ Glottis Initialization [29](#) ⟩  
 ⟨ Glottis Computation [30](#) ⟩

This code is cited in sections [1](#) and [6](#).

This code is used in section [1](#).

**29.** Initialization of the glottis is done inside of *glottis\_init*.

⟨ Glottis Initialization [29](#) ⟩ ≡  

```
static void glottis_init(glottis *glot, SPFLOAT sr)
{
    glot->freq = 140;    /* 140Hz frequency by default */
    glot->tenseness = 0.6; /* value between 0 and 1 */
    glot->T = 1.0/sr;    /* big T */
    glot->time_in_waveform = 0;
    glottis_setup_waveform(glot, 0);
}
```

This code is used in section [28](#).

**30.** This is where a single sample of audio is computed for the glottis

⟨Glottis Computation 30⟩ ≡

```
static SPFLOAT glottis_compute(sp_data * sp, glottis * glot, SPFLOAT lambda)
{
    SPFLOAT out;
    SPFLOAT aspiration;
    SPFLOAT noise;
    SPFLOAT t;
    SPFLOAT intensity;
    out = 0;
    intensity = 1.0;
    glot->time_in_waveform += glot->T;
    if (glot->time_in_waveform > glot->waveform_length) {
        glot->time_in_waveform -= glot->waveform_length;
        glottis_setup_waveform(glot, lambda);
    }
    t = (glot->time_in_waveform / glot->waveform_length);
    if (t > glot->Te) {
        out = (-exp(-glot->epsilon * (t - glot->Te)) + glot->shift) / glot->delta;
    }
    else {
        out = glot->E0 * exp(glot->alpha * t) * sin(glot->omega * t);
    }
    noise = 2.0 * ((SPFLOAT)sp_rand(sp) / SP_RANDOM_MAX) - 1;
    aspiration = intensity * (1 - sqrt(glot->tenseness)) * 0.3 * noise;
    aspiration *= 0.2;
    out += aspiration;
    return out;
}
```

This code is cited in sections 24 and 33.

This code is used in section 28.

**31.** The function *glottis\_setup\_waveform* is tasked with setting the variables needed to create the glottis waveform. The glottal model used here is known as the LF-model, as described in Lu and Smith[lu2000glottal].■

⟨Set up Glottis Waveform 31⟩ ≡

```
static void glottis_setup_waveform(glottis * glot, SPFLOAT lambda) {
    ⟨Set up local variables 32⟩
    ⟨Derive waveform_length and R_d 33⟩
    ⟨Derive R_a, R_k, and R_g 34⟩
    ⟨Derive T_a, T_p, and T_e 35⟩
    ⟨Calculate epsilon, shift, and delta 36⟩
    ⟨Calculate Integrals 37⟩
    ⟨Calculate E_0 38⟩
    ⟨Update variables in glottis data structure 39⟩
}
```

This code is cited in section 24.

This code is used in section 28.

**32.** A number of local variables are used for intermediate calculations. They are described below.

⟨Set up local variables 32⟩ ≡

```
SPFLOAT Rd;
SPFLOAT Ra;
SPFLOAT Rk;
SPFLOAT Rg;
SPFLOAT Ta;
SPFLOAT Tp;
SPFLOAT Te;
SPFLOAT epsilon;
SPFLOAT shift;
SPFLOAT delta;
SPFLOAT rhs_integral;
SPFLOAT lower_integral;
SPFLOAT upper_integral;
SPFLOAT omega;
SPFLOAT s;
SPFLOAT y;
SPFLOAT z;
SPFLOAT alpha;
SPFLOATE0;
```

This code is used in section 31.

**33.** To begin, both *waveform\_length* and *R<sub>d</sub>* are calculated.

The variable *waveform\_length* is the period of the waveform based on the current frequency, and will be used later on in ⟨Glottis Computation 30⟩.

*R<sub>d</sub>* is part of a set of normalized timing parameters used to calculate the time coefficients described in the LF model [fant1997voice]. The other timing parameters *R<sub>a</sub>*, *R<sub>g</sub>*, and *R<sub>k</sub>* can be computed in terms of *R<sub>d</sub>*, which is why this gets computed first. *R<sub>d</sub>* is derived from the parameter *glot-tenseness*.

*R<sub>d</sub>* is then clamped to be in between 0.5 and 2.7, as these are good approximations[lu2000glottal].

⟨Derive *waveform\_length* and *R<sub>d</sub>* 33⟩ ≡

```
glot-Rd = 3 * (1 - glot-tenseness);
glot-waveform_length = 1.0/glot-freq;
Rd = glot-Rd;
if (Rd < 0.5) Rd = 0.5;
if (Rd > 2.7) Rd = 2.7;
```

This code is used in section 31.



**34.**  $R_d$  can be used to calculate approximations for  $R_a$ ,  $R_g$ , and  $R_k$ . The equations described below have been derived using linear regression.

$$R_{ap} = \frac{(-1 + 4.8R_d)}{100}$$

$$R_{kp} = \frac{(22.4 + 11.8R_d)}{100}$$

$R_{gp}$  is derived using the results from  $R_{ap}$  and  $R_{kp}$  in the following equation described in Fant 1997:

$$R_d = (1/0.11)(0.5 + 1.2R_k)(R_k/4R_g + R_a)$$

Which yields:

$$R_{gp} = \frac{(R_{kp}/4)(0.5 + 1.2R_{kp})}{(0.11R_d - R_{ap} * (0.5 + 1.2R_{kp}))}$$

⟨ Derive  $R_a$ ,  $R_k$ , and  $R_g$  34 ⟩ ≡

$$Ra = -0.01 + 0.048 * Rd;$$

$$Rk = 0.224 + 0.118 * Rd;$$

$$Rg = (Rk/4) * (0.5 + 1.2 * Rk) / (0.11 * Rd - Ra * (0.5 + 1.2 * Rk));$$

This code is used in section 31.

**35.** The parameters approximating  $R_a$ ,  $R_g$ , and  $R_k$  can be used to calculate the timing parameters  $T_a$ ,  $T_p$ , and  $T_e$  in the LF model:

$$T_a = R_{ap}$$

$$T_p = 2R_{gp}^{-1}$$

$$T_e = T_p + T_p R_{kp}$$

⟨ Derive  $T_a$ ,  $T_p$ , and  $T_e$  35 ⟩ ≡

$$Ta = Ra;$$

$$Tp = (\text{SPFLOAT})1.0 / (2 * Rg);$$

$$Te = Tp + Tp * Rk;$$

This code is used in section 31.

**36.** ⟨ Calculate epsilon, shift, and delta 36 ⟩ ≡

$$\epsilon = (\text{SPFLOAT})1.0 / Ta;$$

$$\text{shift} = \exp(-\epsilon * (1 - Te));$$

$$\text{delta} = 1 - \text{shift};$$

This code is used in section 31.

**37.** ⟨ Calculate Integrals 37 ⟩ ≡

$$\text{rhs\_integral} = (\text{SPFLOAT})(1.0 / \epsilon) * (\text{shift} - 1) + (1 - Te) * \text{shift};$$

$$\text{rhs\_integral} = \text{rhs\_integral} / \text{delta};$$

$$\text{lower\_integral} = -(Te - Tp) / 2 + \text{rhs\_integral};$$

$$\text{upper\_integral} = -\text{lower\_integral};$$

This code is used in section 31.

**38.**

$$E_0 = -\frac{E_e}{e^{\alpha T} \sin \omega_g T_e}$$

$$\omega = \frac{\pi}{T_p}$$

$$\epsilon T_a = 1 - e^{-\epsilon(T_c - T_e)}$$

$\langle \text{Calculate } E_0 \text{ 38} \rangle \equiv$   
 $\text{omega} = \text{M\_PI} / T_p;$   
 $s = \sin(\text{omega} * T_e);$   
 $y = -\text{M\_PI} * s * \text{upper\_integral} / (T_p * 2);$   
 $z = \log(y);$   
 $\text{alpha} = z / (T_p / 2 - T_e);$   
 $\text{E0} = -1 / (s * \exp(\text{alpha} * T_e));$

This code is used in section 31.

**39.**  $\langle \text{Update variables in glottis data structure 39} \rangle \equiv$

$\text{glot} \rightarrow \text{alpha} = \text{alpha};$   
 $\text{glot} \rightarrow \text{E0} = \text{E0};$   
 $\text{glot} \rightarrow \text{epsilon} = \text{epsilon};$   
 $\text{glot} \rightarrow \text{shift} = \text{shift};$   
 $\text{glot} \rightarrow \text{delta} = \text{delta};$   
 $\text{glot} \rightarrow T_e = T_e;$   
 $\text{glot} \rightarrow \text{omega} = \text{omega};$

This code is used in section 31.

## The Vocal Tract <sup>(40)</sup>

The vocal tract is the part of the vocal model which takes the excitation signal (the glottis) and produces the vowel formants from it.

The two main functions for the vocal tract consist of an initialization function *tract\_init* called once before runtime, and a computation function *tract\_compute* called at twice the sampling rate. See ⟨Vocal Tract Initialization 41⟩ and ⟨Vocal Tract Computation 46⟩ for more detail.

```
⟨The Vocal Tract 40⟩ ≡
  ⟨Calculate Vocal Tract Reflections 52⟩
  ⟨Calculate Vocal Tract Nose Reflections 53⟩
  ⟨Reshape Vocal Tract 54⟩
  ⟨Vocal Tract Initialization 41⟩
  ⟨Vocal Tract Computation 46⟩
```

This code is cited in sections 1 and 58.

This code is used in section 1.

**41.** The function *tract\_init* is responsible for zeroing out variables and buffers, as well as setting up constants.

```
⟨Vocal Tract Initialization 41⟩ ≡
static void tract_init(sp_data *sp, tract *tr)
{
    int i;
    SPFLOAT diameter, d;    /* needed to set up diameter arrays */
    ⟨Initialize Tract Constants and Variables 42⟩
    ⟨Zero Out Tract Buffers 43⟩
    ⟨Set up Vocal Tract Diameters 44⟩
    ⟨Set up Nose Diameters 45⟩
    tract_calculate_reflections(tr);
    tract_calculate_nose_reflections(tr);
    tr->nose_diameter[0] = tr->velum_target;
    tr->block_time = 512.0/(SPFLOAT)sp->sr;
}
```

This code is cited in section 40.

This code is used in section 40.

**42.**  $\langle$  Initialize Tract Constants and Variables 42  $\rangle \equiv$ 

```
|n = 44;
|nose_length = 28;
|nose_start = 17;
|reflection_left = 0.0;
|reflection_right = 0.0;
|reflection_nose = 0.0;
|new_reflection_left = 0.0;
|new_reflection_right = 0.0;
|new_reflection_nose = 0.0;
|velum_target = 0.01;
|glottal_reflection = 0.75;
|lip_reflection = -0.85;
|last_obstruction = -1;
|movement_speed = 15;
|lip_output = 0;
|nose_output = 0;
|tip_start = 32;

















|  |

```

This code is used in section 41.

**43.** Several floating-point arrays are needed for the scattering junctions. C does not zero these out by default. Below, the standard function *memset()* from *string.h* is used to zero out each of the blocks of memory.

 $\langle$  Zero Out Tract Buffers 43  $\rangle \equiv$ 

```

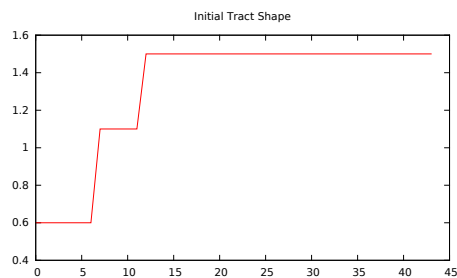
memset(tr_diameter, 0, tr_n * sizeof (SPFLOAT));
memset(tr_rest_diameter, 0, tr_n * sizeof (SPFLOAT));
memset(tr_target_diameter, 0, tr_n * sizeof (SPFLOAT));
memset(tr_new_diameter, 0, tr_n * sizeof (SPFLOAT));
memset(tr_L, 0, tr_n * sizeof (SPFLOAT));
memset(tr_R, 0, tr_n * sizeof (SPFLOAT));
memset(tr_reflection, 0, (tr_n + 1) * sizeof (SPFLOAT));
memset(tr_new_reflection, 0, (tr_n + 1) * sizeof (SPFLOAT));
memset(tr_junction_outL, 0, (tr_n + 1) * sizeof (SPFLOAT));
memset(tr_junction_outR, 0, (tr_n + 1) * sizeof (SPFLOAT));
memset(tr_A, 0, tr_n * sizeof (SPFLOAT));
memset(tr_max_amplitude, 0, tr_n * sizeof (SPFLOAT));
memset(tr_noseL, 0, tr_nose_length * sizeof (SPFLOAT));
memset(tr_noseR, 0, tr_nose_length * sizeof (SPFLOAT));
memset(tr_nose_junc_outL, 0, (tr_nose_length + 1) * sizeof (SPFLOAT));
memset(tr_nose_junc_outR, 0, (tr_nose_length + 1) * sizeof (SPFLOAT));
memset(tr_nose_diameter, 0, tr_nose_length * sizeof (SPFLOAT));
memset(tr_noseA, 0, tr_nose_length * sizeof (SPFLOAT));
memset(tr_nose_max_amp, 0, tr_nose_length * sizeof (SPFLOAT));

```

This code is used in section 41.

44. The cylindrical diameters approximating the vocal tract are set up below. These diameters will be modified and shaped by user control to shape the vowel sound.

The initial shape of the vocal tract is plotted below:



```

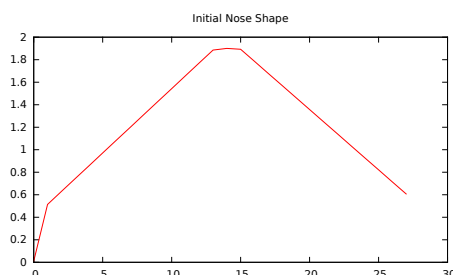
⟨ Set up Vocal Tract Diameters 44 ⟩ ≡
  for (i = 0; i < tr-n; i++) {
    diameter = 0;
    if (i < 7 * (SPFLOAT)tr-n/44 - 0.5) {
      diameter = 0.6;
    }
    else if (i < 12 * (SPFLOAT)tr-n/44) {
      diameter = 1.1;
    }
    else {
      diameter = 1.5;
    }
    tr-diameter[i] = tr-rest_diameter[i] = tr-target_diameter[i] = tr-new_diameter[i] = diameter;
  }

```

This code is used in section 41.

**45.** The cylindrical diameters representing nose are set up. These are only set once, and are immutable for the rest of the program.

The shape of the nasal passage is plotted below:



```

⟨ Set up Nose Diameters 45 ⟩ ≡
for (i = 0; i < tr→nose.length; i++) {
    d = 2 * ((SPFLOAT)i/tr→nose.length);
    if (d < 1) {
        diameter = 0.4 + 1.6 * d;
    }
    else {
        diameter = 0.5 + 1.5 * (2 - d);
    }
    diameter = MIN(diameter, 1.9);
    tr→nose_diameter[i] = diameter;
}

```

This code is used in section 41.

**46.** The vocal tract computation function computes a single sample of audio. As the original implementation describes it, this function is designed to run at twice the sampling rate. For this reason, it is called twice in the top level call back (see ⟨ Voc Create 9 ⟩).

At the moment, *tract\_compute* has two input arguments. The variable *in* is the glottal excitation signal. The *lambda* variable is a coefficient for a linear crossfade along the buffer block, used for parameter smoothing. In future iterations, the linear crossfade will be removed in place of one-pole smoothing filters.

```

⟨ Vocal Tract Computation 46 ⟩ ≡
static void tract_compute(sp_data * sp, tract * tr, SPFLOAT in, SPFLOAT lambda)
{
    SPFLOAT r, w;
    int i;
    ⟨ Calculate Scattering Junctions 47 ⟩
    ⟨ Calculate Scattering for Nose 48 ⟩
    ⟨ Update Left/Right delay lines and set lip output 49 ⟩
    ⟨ Calculate Nose Scattering Junctions 50 ⟩
    ⟨ Update Nose Left/Right delay lines and set nose output 51 ⟩
}

```

This code is cited in section 40.

This code is used in section 40.

**47.** A derivation of  $w$  can be seen in section 2.5.2 of Jack Mullens PhD dissertation *Physical Modelling of the Vocal Tract with the 2D Digital Waveguide Mesh*. [mullen2006physical]

```

⟨ Calculate Scattering Junctions 47 ⟩ ≡
  tr-junction_outR[0] = tr-L[0] * tr-glottal_reflection + in;
  tr-junction_outL[tr-n] = tr-R[tr-n - 1] * tr-lip_reflection;
  for (i = 1; i < tr-n; i++) {
    r = tr-reflection[i] * (1 - lambda) + tr-new_reflection[i] * lambda;
    w = r * (tr-R[i - 1] + tr-L[i]);
    tr-junction_outR[i] = tr-R[i - 1] - w;
    tr-junction_outL[i] = tr-L[i] + w;
  }

```

This code is used in section 46.

```

48.  ⟨ Calculate Scattering for Nose 48 ⟩ ≡
  i = tr-nose_start;
  r = tr-new_reflection_left * (1 - lambda) + tr-reflection_left * lambda;
  tr-junction_outL[i] = r * tr-R[i - 1] + (1 + r) * (tr-noseL[0] + tr-L[i]);
  r = tr-new_reflection_right * (1 - lambda) + tr-reflection_right * lambda;
  tr-junction_outR[i] = r * tr-L[i] + (1 + r) * (tr-R[i - 1] + tr-noseL[0]);
  r = tr-new_reflection_nose * (1 - lambda) + tr-reflection_nose * lambda;
  tr-nose_junc_outR[0] = r * tr-noseL[0] + (1 + r) * (tr-L[i] + tr-R[i - 1]);

```

This code is used in section 46.

```

49.  ⟨ Update Left/Right delay lines and set lip output 49 ⟩ ≡
  for (i = 0; i < tr-n; i++) {
    tr-R[i] = tr-junction_outR[i] * 0.999;
    tr-L[i] = tr-junction_outL[i + 1] * 0.999;
  }
  tr-lip_output = tr-R[tr-n - 1];

```

This code is used in section 46.

```

50.  ⟨ Calculate Nose Scattering Junctions 50 ⟩ ≡
  tr-nose_junc_outL[tr-nose_length] = tr-noseR[tr-nose_length - 1] * tr-lip_reflection;
  for (i = 1; i < tr-nose_length; i++) {
    w = tr-nose_reflection[i] * (tr-noseR[i - 1] + tr-noseL[i]);
    tr-nose_junc_outR[i] = tr-noseR[i - 1] - w;
    tr-nose_junc_outL[i] = tr-noseL[i] + w;
  }

```

This code is used in section 46.

```

51.  ⟨ Update Nose Left/Right delay lines and set nose output 51 ⟩ ≡
  for (i = 0; i < tr-nose_length; i++) {
    tr-noseR[i] = tr-nose_junc_outR[i];
    tr-noseL[i] = tr-nose_junc_outL[i + 1];
  }
  tr-nose_output = tr-noseR[tr-nose_length - 1];

```

This code is used in section 46.

**52.** The function *tract.calculate\_reflections* computes reflection coefficients used in the scattering junction. Because this is a rather computationally expensive function, it is called once per render block, and then smoothed.

First, the cylindrical areas of tract section are computed by squaring the diameters, they are stored in the struct variable *A*.

Using the areas calculated, the reflections are calculated using the following formula:

$$R_i = \frac{A_{i-1} - A_i}{A_{i-1} + A_i}$$

To prevent some divide-by-zero edge cases, when  $A_i$  is exactly zero, it is set to be 0.999.

From there, the new coefficients are set.

⟨ Calculate Vocal Tract Reflections 52 ⟩ ≡

```
static void tract.calculate_reflections(tract *tr)
{
    int i;
    SPFLOAT sum;
    for (i = 0; i < tr-n; i++) {
        tr-A[i] = tr-diameter[i] * tr-diameter[i];    /* Calculate area from diameter squared */
    }
    for (i = 1; i < tr-n; i++) {
        tr-reflection[i] = tr-new_reflection[i];
        if (tr-A[i] == 0) {
            tr-new_reflection[i] = 0.999;    /* to prevent bad behavior if 0 */
        }
        else {
            tr-new_reflection[i] = (tr-A[i - 1] - tr-A[i]) / (tr-A[i - 1] + tr-A[i]);
        }
    }
    tr-reflection_left = tr-new_reflection_left;
    tr-reflection_right = tr-new_reflection_right;
    tr-reflection_nose = tr-new_reflection_nose;
    sum = tr-A[tr-nose_start] + tr-A[tr-nose_start + 1] + tr-noseA[0];
    tr-new_reflection_left = (SPFLOAT)(2 * tr-A[tr-nose_start] - sum) / sum;
    tr-new_reflection_right = (SPFLOAT)(2 * tr-A[tr-nose_start + 1] - sum) / sum;
    tr-new_reflection_nose = (SPFLOAT)(2 * tr-noseA[0] - sum) / sum;
}
```

This code is cited in section 53.

This code is used in section 40.



**53.** Similar to *tract.calculate\_reflections*, this function computes reflection coefficients for the nasal scattering junction. For more information on the math that is happening, see ⟨Calculate Vocal Tract Reflections 52⟩.

⟨Calculate Vocal Tract Nose Reflections 53⟩ ≡

```
static void tract_calculate_nose_reflections(tract *tr)
{
    int i;
    for (i = 0; i < tr->nose_length; i++) {
        tr->noseA[i] = tr->nose_diameter[i] * tr->nose_diameter[i];
    }
    for (i = 1; i < tr->nose_length; i++) {
        tr->nose_reflection[i] = (tr->noseA[i - 1] - tr->noseA[i]) / (tr->noseA[i - 1] + tr->noseA[i]);
    }
}
```

This code is used in section 40.

54.

⟨Reshape Vocal Tract 54⟩ ≡

```

static SPFLOAT move_towards(SPFLOAT current, SPFLOAT target, SPFLOAT amt_up, SPFLOAT amt_down)
{
    SPFLOAT tmp;
    if (current < target) {
        tmp = current + amt_up;
        return MIN(tmp, target);
    }
    else {
        tmp = current - amt_down;
        return MAX(tmp, target);
    }
    return 0.0;
}

static void tract_reshape(tract *tr)
{
    SPFLOAT amount;
    SPFLOAT slow_return;
    SPFLOAT diameter;
    SPFLOAT target_diameter;
    int i;
    amount = tr->block_time * tr->movement_speed;
    for (i = 0; i < tr->n; i++) {
        slow_return = 0;
        diameter = tr->diameter[i];
        target_diameter = tr->target_diameter[i];
        if (i < tr->nose_start) slow_return = 0.6;
        else if (i ≥ tr->tip_start) slow_return = 1.0;
        else {
            slow_return = 0.6 + 0.4 * (i - tr->nose_start) / (tr->tip_start - tr->nose_start);
        }
        tr->diameter[i] = move_towards(diameter, target_diameter, slow_return * amount, 2 * amount);
    }
    tr->nose_diameter[0] = move_towards(tr->nose_diameter[0], tr->velum_target, amount * 0.25, amount * 0.1);
    tr->noseA[0] = tr->nose_diameter[0] * tr->nose_diameter[0];
}

```

This code is cited in section 17.

This code is used in section 40.

## Header File (55)

CTANGLE will end up generating two files: a single C amalgamation and this header file.

This header file exists for individuals who wish to use Voc in their own programs. Voc follows Soundpipe’s hardware-agnostic design, and should be trivial to throw in any DSP inner loop.

The contents of the header is fairly minimal. Following a standard header guard, the contents consist of:

- a **typedef** around the opaque struct **sp\_voc**
- function declarations which adhere to the 4-stage Soundpipe module lifecycle model.
- a collection of setter/getter functions to allow to get and set data from the opaque struct.

Since *Voc* makes use of opaque struct pointers, this header file will need to declare setter/getter functions for any user parameters.

```

< voc.h 55 > ≡
#ifndef SP_VOC
#define SP_VOC
    typedef struct sp_voc sp_voc;
    int sp_voc_create(sp_voc **voc);
    int sp_voc_destroy(sp_voc **voc);
    int sp_voc_init(sp_data * sp, sp_voc *voc);
    int sp_voc_compute(sp_data * sp, sp_voc *voc, SPFLOAT * out);
    int sp_voc_tract_compute(sp_data * sp, sp_voc *voc, SPFLOAT * in, SPFLOAT * out);
    void sp_voc_set_frequency(sp_voc *voc, SPFLOAT freq);
    SPFLOAT * sp_voc_get_frequency_ptr(sp_voc *voc);
    SPFLOAT * sp_voc_get_tract_diameters(sp_voc *voc);
    SPFLOAT * sp_voc_get_current_tract_diameters(sp_voc *voc);
    int sp_voc_get_tract_size(sp_voc *voc);
    SPFLOAT * sp_voc_get_nose_diameters(sp_voc *voc);
    int sp_voc_get_nose_size(sp_voc *voc);
    void sp_voc_set_tongue_shape(sp_voc *voc, SPFLOAT tongue_index, SPFLOAT tongue_diameter);
    void sp_voc_set_tenseness(sp_voc *voc, SPFLOAT breathiness);
    SPFLOAT * sp_voc_get_tenseness_ptr(sp_voc *voc);
    void sp_voc_set_velum(sp_voc *voc, SPFLOAT velum);
    SPFLOAT * sp_voc_get_velum_ptr(sp_voc *voc);
    void sp_voc_set_diameters(sp_voc *voc, int blade_start, int lip_start, int tip_start,
        SPFLOAT tongue_index, SPFLOAT tongue_diameter, SPFLOAT * diameters);
    int sp_voc_get_counter(sp_voc *voc);
#endif

```

This code is cited in sections [1](#) and [3](#).

## Small Applications and Examples <sup>(56)</sup>

It has been fruitful investment to write small applications to assist in the debugging process. Such programs can be used to generate plots or visuals, or to act as a simple program to be used with GDB. In addition to debugging, these programs are also used to quickly try out concepts or ideas.

57.

## A Program for Non-Realtime Processing and Debugging

The example program below is a C program designed out of necessity to debug and test Voc. It a program with a simple commandline interface, where the user gives a "mode" along with set of optional arguments.

The following modes are as follows:

- **audio:** writes an audio file called "test.wav". You must supply a duration (in samples).
- **plot:** Uses `sp_process_plot` to generate a matlab/octave compatible program that plots the audio output.
- **tongue:** Will be a test program that experiments with parameters manipulating tongue position. It takes in tongue index and diameter parameters, to allow for experimentation without needing to recompile.

The functions needed to call Voc from C in this way are found in the section [\(Top Level Functions 8\)](#).

```
<debug.c 57> ≡
#include <soundpipe.h>
#include <string.h>
#include <stdlib.h>
#include "voc.h"

static void process(sp_data * sp, void * ud)
{
    SPFLOAT out;
    sp_voc * voc = ud;
    sp_voc_compute(sp, voc, &out);
    sp_out(sp, 0, out);
}

static void run_voc(long len, int type)
{
    sp_voc * voc;
    sp_data * sp;
    sp_create(&sp);
    sp-len = len;
    sp_voc_create(&voc);
    sp_voc_init(sp, voc);
    if (type == 0) {
        sp_process_plot(sp, voc, process);
    }
    else {
        sp_process(sp, voc, process);
    }
    sp_voc_destroy(&voc);
    sp_destroy(&sp);
}

static void run_tongue(SPFLOAT tongue_index, SPFLOAT tongue_diameter)
{
    sp_voc * voc;
    sp_data * sp;
    sp_create(&sp);
    sp_voc_create(&voc);
    sp_voc_init(sp, voc);
    fprintf(stderr, "Tongue_index: %g. Tongue_diameter: %g\n", tongue_index, tongue_diameter);
    sp_voc_set_tongue_shape(voc, tongue_index, tongue_diameter);
    sp_process(sp, voc, process);
    sp_voc_destroy(&voc);
}
```

```

    sp_destroy(&sp);
}
int main(int argc, char *argv[])
{
    if (argc == 1) {
        fprintf(stderr, "Pick a mode!\n");
        exit(0);
    }
    if (!strcmp(argv[1], "plot")) {
        if (argc < 3) {
            fprintf(stderr, "Usage: %s plot duration (samples)\n", argv[0]);
            exit(0);
        }
        run_voc(atoi(argv[2]), 0);
    }
    else if (!strcmp(argv[1], "audio")) {
        if (argc < 3) {
            fprintf(stderr, "Usage: %s audio duration (samples)\n", argv[0]);
            exit(0);
        }
        run_voc(atoi(argv[2]), 1);
    }
    else if (!strcmp(argv[1], "tongue")) {
        if (argc < 4) {
            fprintf(stderr, "Usage: %s tongue tongue_index tongue_diameter\n", argv[0]);
            exit(0);
        }
        run_tongue(atof(argv[2]), atof(argv[3]));
    }
    else {
        fprintf(stderr, "Error: invalid type %s\n", argv[1]);
    }
    return 0;
}

```

This code is cited in section [1](#).

58.

## A Utility for Plotting Data

The following program below is used to write data files to be read by GNUplot. The primary use of this program is for generating use plots in this document, such as those seen in the section [The Vocal Tract 40](#).

```

<plot.c 58> ≡
#include <soundpipe.h>
#include <string.h>
#include <stdlib.h>
#include "voc.h"
static void plot_tract()
{
    sp_voc *voc;
    sp_data *sp;
    SPFLOAT
        *tract;
    int size;
    int i;
    sp_create(&sp);
    sp_voc_create(&voc);
    sp_voc_init(sp, voc);
    tract = sp_voc_get_tract_diameters(voc);
    size = sp_voc_get_tract_size(voc);
    for (i = 0; i < size; i++) {
        printf("%i\t%g\n", i, tract[i]);
    }
    sp_voc_destroy(&voc);
    sp_destroy(&sp);
}
static void plot_nose()
{
    sp_voc *voc;
    sp_data *sp;
    SPFLOAT *nose;
    int size;
    int i;
    sp_create(&sp);
    sp_voc_create(&voc);
    sp_voc_init(sp, voc);
    nose = sp_voc_get_nose_diameters(voc);
    size = sp_voc_get_nose_size(voc);
    for (i = 0; i < size; i++) {
        printf("%i\t%g\n", i, nose[i]);
    }
    sp_voc_destroy(&voc);
    sp_destroy(&sp);
}
static void plot_tongue_shape(int num)
{
    sp_voc *voc;

```

```

    sp_data * sp;
    SPFLOAT
        *tract;
        int size;
        int i;
    sp_create(&sp);
    sp_voc_create(&voc);
    sp_voc_init(sp, voc);
    tract = sp_voc_get_tract_diameters(voc);
    size = sp_voc_get_tract_size(voc);
    switch (num) {
    case 1: sp_voc_set_tongue_shape(voc, 20.5, 3.5);
        break;
    case 2: sp_voc_set_tongue_shape(voc, 25.5, 3.5);
        break;
    case 3: sp_voc_set_tongue_shape(voc, 20.5, 2.0);
        break;
    case 4: sp_voc_set_tongue_shape(voc, 24.8, 1.4);
        break;
    }
    for (i = 0; i < size; i++) {
        printf("%i\t%g\n", i, tract[i]);
    }
    sp_voc_destroy(&voc);
    sp_destroy(&sp);
}

int main(int argc, char **argv)
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s plots/name.dat\n", argv[0]);
        exit(1);
    }
    if (!strcmp(argv[1], "plots/tract.dat", 100)) {
        plot_tract();
    }
    else if (!strcmp(argv[1], "plots/nose.dat", 100)) {
        plot_nose();
    }
    else if (!strcmp(argv[1], "plots/tongueshape1.dat", 100)) {
        plot_tongue_shape(1);
    }
    else if (!strcmp(argv[1], "plots/tongueshape2.dat", 100)) {
        plot_tongue_shape(2);
    }
    else if (!strcmp(argv[1], "plots/tongueshape3.dat", 100)) {
        plot_tongue_shape(3);
    }
    else if (!strcmp(argv[1], "plots/tongueshape4.dat", 100)) {
        plot_tongue_shape(4);
    }
    else {

```



```
    fprintf(stderr, "Plot: could not find plot %s\n", argv[1]);  
    exit(1);  
}  
return 0;  
}
```

This code is cited in section [1](#).

## External Sporth Plugins <sup>(59)</sup>

Sporth, a stack-based synthesis language, is the preferred tool of choice for sound design experimentation and prototyping with Voc. A version of Voc has been ported to Sporth as third party plugin, known as an *external Sporth Plugin*.

### Sporth Plugins as Seen from Sporth

In Sporth, one has the ability to dynamically load custom unit-generators or, *ugens*, into Sporth. Such a unit generator can be seen here in Sporth code:

```
_voc ". /voc.so" fl
# frequency
170 8 1 5 jitter +
# tongue position
0 1 5 randi
# tongue diameter
0 1 20 randi
# breathiness
0.4 0.7 9 randi
# velum amount
0
_voc fe

# Add reverberation using the Zita reverberator
dup dup 1 2 8000 zrev drop -3 ampdb * +

# close the plugin
_voc fc
```

In the code above, the plugin file is loaded via `fl` (function load) and saved into the table `_voc`. An instance of `_voc` is created with `fe` (function execute). Finally, the dynamic plugin is closed with `fc` (function close).

### Sporth plugins as seen from C.

Custom unit generators are written in C using a special interface provided by the Sporth API. The functionality of an external sporth ugen is nearly identical to an internal one, with exceptions being the function definition and how custom user-data is handled. Besides that, they can be seen as equivalent.

The entirety of the Sporth unit generator is contained within a single subroutine, declared **static** so as to not clutter the global namespace. The crux of the function is a case switch outlining four unique states of operation, which define the *lifecycle* of a Sporth ugen. This design concept comes from Soundpipe, the music DSP library that Sporth is built on top of.

These states are executed in this order:

1. Create: allocates memory for the DSP module
2. Initialize: zeros out and sets up default values
3. Compute: Computes an audio-rate sample (or samples)
4. Destroy: frees all memory previously allocated in Create

Create and init are called once during runtime, compute is called as many times as needed while the program is running, and destroy is called once when the program is stopped.

The code below shows the outline for the main Sporth Ugen.

```
<ugen.c 59> =
#include <stdlib.h>
#include <math.h>
#include <string.h>
```

```

#include <soundpipe.h>
#include <sporth.h>
#include "voc.h"
static int sporth_voc(plumber_data * pd, sporth_stack * stack, void **ud)
{
    sp_voc *voc;
    SPFLOAT out;
    SPFLOAT freq;
    SPFLOAT pos;
    SPFLOAT diameter;
    SPFLOAT tenseness;
    SPFLOAT nasal;
    switch (pd->mode) {
    case PLUMBER_CREATE:
        < Creation 60 >;
        break;
    case PLUMBER_INIT:
        < Initialization 61 >;
        break;
    case PLUMBER_COMPUTE:
        < Computation 62 >;
        break;
    case PLUMBER_DESTROY:
        < Destruction 63 >;
        break;
    }
    return PLUMBER_OK;
}
< Return Function 64 >

```

See also sections 65 and 66.

This code is cited in sections 1 and 8.

**60.** The first state executed is **creation**, denoted by the macro `PLUMBER_CREATE`. This is the state where memory is allocated, tables are created and stack arguments are checked for validity.

It is here that the top-level function < Voc Create 9 > is called.

```

< Creation 60 > ≡
    sp_voc_create(&voc);
    *ud = voc;
    if (sporth_check_args(stack, "fffff") ≠ SPORTH_OK) {
        plumber_print(pd, "Voc: not enough arguments!\n");
    }
    nasal = sporth_stack_pop_float(stack);
    tenseness = sporth_stack_pop_float(stack);
    diameter = sporth_stack_pop_float(stack);
    pos = sporth_stack_pop_float(stack);
    freq = sporth_stack_pop_float(stack);
    sporth_stack_push_float(stack, 0.0);

```

This code is used in section 59.

**61.** The second state executed is **initialization**, denoted by the macro `PLUMBER_INIT`. This is the state where variables get initialised or zeroed out. It should be noted that auxiliary memory can be allocated here for things involving delay lines with user-specified sizes. For this reason, it is typically not safe to call this twice for reinitialization. (The author admits that this is not an ideal design choice.)

It is here that the top-level function `<Voc Initialization 11>` is called.

```
<Initialization 61> ≡
voc = *ud;
sp_voc_init(pd-sp, voc);
nasal = sporth_stack_pop_float(stack);
tenseness = sporth_stack_pop_float(stack);
diameter = sporth_stack_pop_float(stack);
pos = sporth_stack_pop_float(stack);
freq = sporth_stack_pop_float(stack);
sporth_stack_push_float(stack, 0.0);
```

This code is used in section 59.

**62.** The third state executed is **computation**, denoted by the macro `PLUMBER_COMPUTE`. This state happens during Sporth runtime in the audio loop. Generally speaking, this is where a Ugen will process audio. In this state, strings in this callback are ignored; only floating point values are pushed and popped.

It is here that the top-level function `<Voc Compute 12>` is called.

```
<Computation 62> ≡
voc = *ud;
nasal = sporth_stack_pop_float(stack);
tenseness = sporth_stack_pop_float(stack);
diameter = sporth_stack_pop_float(stack);
pos = sporth_stack_pop_float(stack);
freq = sporth_stack_pop_float(stack);
sp_voc_set_frequency(voc, freq);
sp_voc_set_tenseness(voc, tenseness);
if (sp_voc_get_counter(voc) ≡ 0) {
    sp_voc_set_velum(voc, 0.01 + 0.8 * nasal);
    sp_voc_set_tongue_shape(voc, 12 + 16.0 * pos, diameter * 3.5);
}
sp_voc_compute(pd-sp, voc, &out);
sporth_stack_push_float(stack, out);
```

This code is used in section 59.

**63.** The fourth and final state in a Sporth ugen is **Destruction**, denoted by `PLUMBER_DESTROY`. Any memory allocated in `PLUMBER_CREATE` should be consequently freed here.

It is here that the top-level function `<Voc Destroy 10>` is called.

```
<Destruction 63> ≡
voc = *ud;
sp_voc_destroy(&voc);
```

This code is used in section 59.

**64.** A dynamically loaded sporth unit-generated such as the one defined here needs to have a globally accessible function called *sporth\_return\_ugen*. All this function needs to do is return the ugen function, which is of type *plumber\_dyn\_func*.

```

⟨Return Function 64⟩ ≡
  plumber_dyn_func sporth_return_ugen()
  {
    return sporth_voc;
  }

```

This code is used in section 59.

65.

## A Ugen for the Vocal Tract Model

```

⟨ ugen.c 59 ⟩ +≡
static int sporth_tract(plumber_data * pd, sporth_stack * stack, void **ud)
{
    sp_voc *voc;
    SPFLOAT out;
    SPFLOAT pos;
    SPFLOAT diameter;
    SPFLOAT nasal;
    SPFLOAT in;
    switch (pd→mode) {
    case PLUMBER_CREATE:
        sp_voc_create(&voc);
        *ud = voc;
        if (sporth_check_args(stack, "ffff") ≠ SPORTH_OK) {
            plumber_print(pd, "Voc: ␣not␣enough␣arguments!␣\n");
        }
        nasal = sporth_stack_pop_float(stack);
        diameter = sporth_stack_pop_float(stack);
        pos = sporth_stack_pop_float(stack);
        in = sporth_stack_pop_float(stack);
        sporth_stack_push_float(stack, 0.0);
        break;
    case PLUMBER_INIT:
        voc = *ud;
        sp_voc_init(pd→sp, voc);
        nasal = sporth_stack_pop_float(stack);
        diameter = sporth_stack_pop_float(stack);
        pos = sporth_stack_pop_float(stack);
        in = sporth_stack_pop_float(stack);
        sporth_stack_push_float(stack, 0.0);
        break;
    case PLUMBER_COMPUTE:
        voc = *ud;
        nasal = sporth_stack_pop_float(stack);
        diameter = sporth_stack_pop_float(stack);
        pos = sporth_stack_pop_float(stack);
        in = sporth_stack_pop_float(stack);
        if (sp_voc_get_counter(voc) ≡ 0) {
            sp_voc_set_velum(voc, 0.01 + 0.8 * nasal);
            sp_voc_set_tongue_shape(voc, 12 + 16.0 * pos, diameter * 3.5);
        }
        sp_voc_tract_compute(pd→sp, voc, &in, &out);
        sporth_stack_push_float(stack, out);
        break;
    case PLUMBER_DESTROY:
        voc = *ud;
        sp_voc_destroy(&voc);
        break;
    }
}

```

```

    return PLUMBER_OK;
}

```

66.

## A multi ugen plugin implementation

New Sporth developments contemporary with the creation of Voc have lead to the development of Sporth plugins with multiple ugens.

```

⟨ugen.c 59⟩ +≡
static const plumber_dyn_func sporth_functions[] = {sporth_voc, sporth_tract, };
int sporth_return_ugen_multi(int n, plumber_dyn_func * f)
{
    if (n < 0 ∨ n > 1) {
        return PLUMBER_NOTOK;
    }
    *f = sporth_functions[n];
    return PLUMBER_OK;
}

```

# Sporoth Code Examples <sup>(67)</sup>

Here are some sporoth code examples.

## Chant

```
_voc ". /voc.so" fl
36 0.3 1 4 jitter + mtof
0.1 1 sine 0 1 biscale
0.9
0.9
0.3 1 sine 0 1 biscale
_voc fe 36 mtof 70 5 eqfil
dup dup 0.97 10000 revsc drop -14 ampdb * dcbk +
_voc fc
```

## Rant

```
# It kind of sounds like an angry rant
_voc ". /voc.so" fl
100
8 metro 0.3 maygate 200 * + 0.1 port
30 1 10 jitter +
0 1 3 randi
0 1 3 20 1 randi randi
0.7
0
_voc fe
1 metro 0.7 maygate 0.03 port *
dup dup 1 2 8000 zrev drop -10 ampdb * +
_voc fc
```

## Unya

```
_voc ". /voc.so" fl
_rate var
_seq "0 2 4 7 9 11 12" gen_vals
15 inv 1 sine 0.3 3 biscale _rate set
_rate get metro 1 _seq tseq 48 + 5 6 1 randi 1 sine 0.3 * + mtof
_rate get metro 0.1 0.01 0.1 tenv 0.0 0.3 scale
_rate get metro 0.1 0.1 0.3 tenv 0.0 _rate get metro 0.3 0.9 trand scale
0.8
_rate get metro tog
_voc fe
dup dup 0.9 8000 revsc drop -14 ampdb * dcbk +
_voc fc
```



## References <sup>(68)</sup>