1. This document describes libline, a library for producing lines and automation curves for audio-related purposes. Line segments produced are audio-rate and sample accurate. Libline will is designed to pair well with sound tools like Soundpipe and Sporth, where such gesture facilities do not exist.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#ifdef LL_SPORTH_UGEN
#include "plumber.h"
#endif
#include "line.h"
  ⟨ Top 45 ⟩
```

2. The Header File. This library has a single header file, containing all the necessary struct and function definitions for the API. This file should be installed alongside the generated library file in order to be used with other programs.

```
⟨ line.h 2 ⟩ ≡
#ifndef LINE_H
#define LINE_H
#ifdef LL_SPORTH_STANDALONE
#include <soundpipe.h>
#include <sporth.h>
#endif
  ⟨ Header Data 4 ⟩
#endif
```

3. Type Declarations. The following section describes the type declarations used in libline.

4. Line values use floating point precision. This precision is set using the macro definition *ll_flt* rather than a type declaration. By default, it is a floating point value. However, this value can be overridden at compile time.

⟨ Header Data 4 ⟩ ≡

```
#ifndef LLFLOAT
  typedef float ll_flt;
#else
  typedef LLFLOAT ll_flt;
#endif
#define UINTunsigned int
```

See also sections 5, 6, 7, 8, 9, 11, 12, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 42, 43, and 44.

This code is used in section 2.

5. The core unit is a point, which has two fundamental properties: a value, and a duration.

⟨ Header Data 4 ⟩ +≡

```
  typedef struct ll_point ll_point;
```

6. Points are tacked on sequentially, and then they are interpolated with some specified behavior. This collection of points forms a line.

⟨ Header Data 4 ⟩ +≡

```
  typedef struct ll_line ll_line;
```

7. A collection of lines is grouped in an interface known *ll_lines*.

⟨ Header Data 4 ⟩ +≡

```
  typedef struct ll_lines ll_lines;
```

8. Memory allocation functions are needed in some situations. By default, these are just wrappers around malloc free. However, this is designed so that they be overridden use custom memory handling functions.

⟨ Header Data 4 ⟩ +≡

```
  typedef void *(*ll_cb_malloc)(void *ud, size_t size);
  typedef void(*ll_cb_free)(void *ud, void *ptr);
```

9. A step function is a function which computes a line segment local to a point.

⟨ Header Data 4 ⟩ +≡

```
  typedef ll_flt(*ll_cb_step)(ll_point *pt, void *ud, UINTpos, UINTdur);
```

10. Memory Functions.

11. Default memory functions are implemented for line. They are simply wrappers for *malloc* and *free*.

⟨ Header Data 4 ⟩ +≡

void ∗*ll_malloc*(**void** ∗*ud*, **size_t** *size*);
void *ll_free*(**void** ∗*ud*, **void** ∗*ptr*);
void *ll_free_nothing*(**void** ∗*ud*, **void** ∗*ptr*);

12. Memory functions are embedded inside of the point data struct, and are exposed indirectly. *ll_point_destroy* specifically destroys data used by the interpolator.

⟨ Header Data 4 ⟩ +≡

void ∗*ll_point_malloc*(**ll_point** ∗*pt*, **size_t** *size*);
void *ll_point_free*(**ll_point** ∗*pt*, **void** ∗*ptr*);
void *ll_point_destroy*(**ll_point** ∗*pt*);

13. Size and Initialization.

14. Compilers are unable to tell what size opaque pointers are, so functions need to be written which return the size. This also shifts the burden of allocation onto the user.

⟨ Header Data 4 ⟩ +≡
size_t *ll_line_size*(**void**);
size_t *ll_point_size*(**void**);

15. Once memory is allocated, data types need to be initialized. These functions are safe to call multiple times, since no memory allocation happens here. After this, things can be done to the structs.

⟨ Header Data 4 ⟩ +≡
void *ll_point_init*(**ll_point** **pt*);
void *ll_line_init*(**ll_line** **ln*, **int** *sr*);

16. Point Declarations.

17. Points have two fundamental properties: a value, and a duration for that value.

⟨ Header Data 4 ⟩ +≡
void *ll_point_value*(**ll_point** ∗*pt*, **ll_flt** *val*);
void *ll_point_dur*(**ll_point** ∗*pt*, **ll_flt** *dur*);
ll_flt *ll_point_get_dur*(**ll_point** ∗*pt*);

18. Points have a *next* value, referencing the next point value.

⟨ Header Data 4 ⟩ +≡
void *ll_point_set_next_value*(**ll_point** ∗*pt*, **ll_flt** ∗*val*);

19. Points have a point A and point B.

⟨ Header Data 4 ⟩ +≡
ll_flt *ll_point_A*(**ll_point** ∗*pt*);
ll_flt *ll_point_B*(**ll_point** ∗*pt*);

20. In order to set the next value, there must be a function which is able to return the memory address of the previous point value (not the next value).

⟨ Header Data 4 ⟩ +≡
ll_flt ∗*ll_point_get_value*(**ll_point** ∗*pt*);

21. Points also act as a linked list, so they also contain a pointer to the next entry.

⟨ Header Data 4 ⟩ +≡
void *ll_point_set_next_point*(**ll_point** ∗*pt*, **ll_point** ∗*next*);

22. The linked list must be read as well written to, so a function is needed to retrieve the next point in the linked list.

⟨ Header Data 4 ⟩ +≡
ll_point ∗*ll_point_get_next_point*(**ll_point** ∗*pt*);

23. This is calls the step function inside of a point.

⟨ Header Data 4 ⟩ +≡
ll_flt *ll_point_step*(**ll_point** ∗*pt*, `UINT` *pos*, `UINT` *dur*);

24. These functions are needed to set up the step functions in point.

⟨ Header Data 4 ⟩ +≡
void *ll_point_data*(**ll_point** ∗*pt*, **void** ∗*data*);
void *ll_point_cb_step*(**ll_point** ∗*pt*, *ll_cb_step stp*);
void *ll_point_cb_destroy*(**ll_point** ∗*pt*, *ll_cb_free destroy*);

25. This function sets custom memory allocation functions for the point.

⟨ Header Data 4 ⟩ +≡
void *ll_point_mem_callback*(**ll_point** ∗*pt*, **ll_cb_malloc** *m*, *ll_cb_free f*);

26. Line Function Declarations.

27. A point, once it is set, can be tacked on to the end of a line. The value of this point becomes the end value of the previous point.

⟨ Header Data 4 ⟩ +≡
  void *ll_line_append_point*(**ll_line** ∗*ln*, **ll_point** ∗*p*);

28. The function *ll_line_append_point* assumes that points will be allocated and freed by the user. However, this is often not an ideal situation. The line has the ability to handle memory internally. This function will return a pointer to the value, for cases when further manipulations need to happen to the point.

⟨ Header Data 4 ⟩ +≡
  ll_point ∗*ll_line_append*(**ll_line** ∗*ln*, **ll_flt** *val*, **ll_flt** *dur*);

29. All things that must be allocated must be freed as well. This function frees all data allocated from functions like *ll_line_append*.

⟨ Header Data 4 ⟩ +≡
  void *ll_line_free*(**ll_line** ∗*ln*);

30. For situations where custom memory allocation is desired, the default callbacks for memory can be overridden.

⟨ Header Data 4 ⟩ +≡
  void *ll_line_mem_callback*(**ll_line** ∗*ln*, **ll_cb_malloc** *m*, *ll_cb_free f*);

31. Once all points have been added, the line is finalized and rewound to the beginning, where it can be ready to be computed as an audio-rate signal in time.

⟨ Header Data 4 ⟩ +≡
  void *ll_line_done*(**ll_line** ∗*ln*);
  void *ll_line_reset*(**ll_line** ∗*ln*);

32. This function gets every sample inside of the audio loop, generating a single sample and moving forward in time by a single sample.

⟨ Header Data 4 ⟩ +≡
  ll_flt *ll_line_step*(**ll_line** ∗*ln*);

33. This function will print all the points in a given line.

⟨ Header Data 4 ⟩ +≡
  void *ll_line_print*(**ll_line** ∗*ln*);

34. This function sets a point to be a linear point.

⟨ Header Data 4 ⟩ +≡
  void *ll_linpoint*(**ll_point** ∗*pt*);

35. Sets a point to be an exponential point.

⟨ Header Data 4 ⟩ +≡
  void *ll_exppoint*(**ll_point** ∗*pt*, **ll_flt** *curve*);

36. Sets a point to be a tick.

⟨ Header Data 4 ⟩ +≡
  void *ll_tick*(**ll_point** ∗*pt*);

37. Sets a point to be a bezier curve.

⟨ Header Data 4 ⟩ +≡
void *ll_bezier*(**ll_point** ∗*pt*, **ll_flt** *cx*, **ll_flt** *cy*);

38. Sets time scale of the line.

⟨ Header Data 4 ⟩ +≡
void *ll_line_timescale*(**ll_line** ∗*ln*, **ll_flt** *scale*);

39. The function *ll_line_bind_float* binds a float value to a line.

⟨ Header Data 4 ⟩ +≡
void *ll_line_bind_float* (**ll_line** ∗*ln*, **ll_flt** ∗ **line**) ;

40. The functions below are needed to be able to access points on the line.

⟨ Header Data 4 ⟩ +≡
ll_point ∗*ll_line_top_point*(**ll_line** ∗*ln*);
int *ll_line_npoints*(**ll_line** ∗*ln*);

41. Lines Function Declarations.

42. These are the functions used for **ll_lines**. More words for this will be added later if needed.

⟨ Header Data 4 ⟩ +≡

size_t *ll_lines_size*( );
void *ll_lines_init*(**ll_lines** $*l$, **int** *sr*);
void *ll_lines_mem_callback*(**ll_lines** $*l$, **void** $*ud$, **ll_cb_malloc** m, *ll_cb_free* f); **void** *ll_lines_append* (**ll_lines** $*l$, **ll_line** $**$ **line** , **ll_flt** $**val$) ;
void *ll_lines_step*(**ll_lines** $*l$);
void *ll_lines_free*(**ll_lines** $*l$);
ll_line $*$*ll_lines_current_line*(**ll_lines** $*l$);

43. High-level Interface Declarations. These functions provide a convenient interface for constructing lines.

⟨ Header Data 4 ⟩ +≡

```
void ll_add_linpoint(ll_lines *l, ll_flt val, ll_flt dur);
void ll_add_exppoint(ll_lines *l, ll_flt val, ll_flt dur, ll_flt curve);
void ll_add_step(ll_lines *l, ll_flt val, ll_flt dur);
void ll_add_tick(ll_lines *l, ll_flt dur);
void ll_end(ll_lines *l);
void ll_timescale(ll_lines *l, ll_flt scale);
void ll_timescale_bpm(ll_lines *l, ll_flt bpm);
```

44. Sporth Function Declarations. An optional feature of libline is to have hooks into the Sporth programming language.

⟨ Header Data 4 ⟩ +≡

```
#ifdef LL_SPORTH
  void ll_sporth_ugen(ll_lines *l, plumber_data *pd, const char *ugen);
  ll_line *ll_sporth_line(ll_lines *l, plumber_data *pd, const char *name);
  void ll_sporth_reset_ugen(ll_lines *l, plumber_data *pd, const char *ugen);
#endif
```

45. The Point. The point is the atomic value used inside of libline.

⟨ Top 45 ⟩ ≡
  ⟨ The Point 47 ⟩

See also sections 71, 106, 122, 127, 132, 139, 142, and 143.
This code is used in section 1.

46. The ll_point struct declaration.

47. A libline point can be best thought of as a line chunk going from point A to point B over a given duration in seconds.

⟨ The Point 47 ⟩ ≡
  struct ll_point {
  ll_flt *A*;
  ll_flt *dur*;
  ll_flt ∗*B*;

See also sections 48, 49, 50, 51, 53, 54, 56, 57, 58, 59, 60, 61, 62, 64, 65, 69, and 70.

This code is used in section 45.

48. The line is built around a linked list data structure, so the struct has a reference to the next entry in the list.

⟨ The Point 47 ⟩ +≡
  ll_point ∗*next*;

49. Points have various styles of interpolation, and with that comes custom user data for the allocator, and memory allocation.

⟨ The Point 47 ⟩ +≡
  ll_cb_malloc *malloc*;
  ll_cb_free free;
  void ∗*ud*;

50. Custom data needs to be freed in a general kind of way. This is another free callback called destroy. This should be called before free.

⟨ The Point 47 ⟩ +≡
  void ∗*data*;
  ll_cb_free destroy;

51. A step function computes a line segment local to the point. By default, this is set to return point A¿

⟨ The Point 47 ⟩ +≡
  ll_cb_step step;
  } ;

52. Point Initialization.

53. The size of the point struct is implemented as a function.

⟨ The Point 47 ⟩ +≡
```
size_t ll_point_size(void)
{
  return sizeof(ll_point);
}
```

54. Initialization. Add some words here.

⟨ The Point 47 ⟩ +≡
```
⟨ Default Step Function 67 ⟩void ll_point_init(ll_point *pt)
    {
      pt→A = 1.0;    /* A reasonable default value */
      pt→dur = 1.0;    /* A one-second duration by default */
      pt→B = &pt→A;    /* Point B points to point A by default */
      pt→ud = Λ;
      pt→free = ll_free;
      pt→malloc = ll_malloc;
      pt→data = Λ;
      pt→destroy = ll_free_nothing;
      pt→step = step;
    }
```

55. Point Setters and Getters. The following describes setter and getter functions needed for the **ll_point** opaque pointer type.

56. Set the initial "A" value.

⟨ The Point 47 ⟩ +≡
```
void ll_point_value(ll_point *pt, ll_flt val)
{
  pt->A = val;
}
```

57. This sets the point of the "B" value. Note that this is a pointer value.

⟨ The Point 47 ⟩ +≡
```
void ll_point_set_next_value(ll_point *pt, ll_flt *val)
{
  pt->B = val;
}
```

58. Set the point duration.

⟨ The Point 47 ⟩ +≡
```
void ll_point_dur(ll_point *pt, ll_flt dur)
{
  pt->dur = dur;
}
ll_flt ll_point_get_dur(ll_point *pt)
{
  return pt->dur;
}
```

59. The following function is used to set the next entry in the linked list.

⟨ The Point 47 ⟩ +≡
```
void ll_point_set_next_point(ll_point *pt, ll_point *next)
{
  pt->next = next;
}
```

60. The following function is used to retrive the next entry in the linked list.

⟨ The Point 47 ⟩ +≡
```
ll_point *ll_point_get_next_point(ll_point *pt)
{
  return pt->next;
}
```

61. In order to set a B value, there needs to be a way to get the memory address of another points A value. This function returns the memory address of a points A value.

⟨ The Point 47 ⟩ +≡
```
ll_flt *ll_point_get_value(ll_point *pt)
{
  return &pt->A;
}
```

62. These functions return the A and B values in the point struct, and are particularly useful for interpolation functions.

⟨ The Point 47 ⟩ +≡
```
ll_flt ll_point_A(ll_point *pt)
{
  return pt→A;
}
ll_flt ll_point_B(ll_point *pt)
{
  return *pt→B;
}
```

63. Point Memory Handling.

64. Various interpolation styles will require the ability to allocate memory. For this reason, the memory allocation functions must be exposed.

⟨ The Point 47 ⟩ +≡

```
void *ll_point_malloc(ll_point *pt, size_t size)
{
  return pt->malloc(pt->ud, size);
}
void ll_point_free(ll_point *pt, void *ptr)
{
  pt->free(pt->ud, ptr);
}
```

65. Data allocated by the interpolator is *destroyed* using the internal free function.

⟨ The Point 47 ⟩ +≡

```
void ll_point_destroy(ll_point *pt)
{
  pt->destroy(pt, pt->data);
}
```

66. Point Step Function. Every point has a "step" function, which computes the current points value at that moment in time.

67. The default step function simply returns point A.

⟨ Default Step Function 67 ⟩ ≡

```
static ll_flt step(ll_point *pt, void *ud, UINT pos, UINT dur)
{
  return ll_point_A(pt);
}
```

See also section 68.

This code is used in section 54.

68. These functions set the internal variables for the step function, step function data, and the destroy function, respectively.

⟨ Default Step Function 67 ⟩ +≡

```
void ll_point_data(ll_point *pt, void *data)
{
  pt->data = data;
}
void ll_point_cb_step(ll_point *pt, ll_cb_step stp)
{
  pt->step = stp;
}
void ll_point_cb_destroy(ll_point *pt, ll_cb_free destroy)
{
  pt->destroy = destroy;
}
```

69. This calls the step function inside of the point.

⟨ The Point 47 ⟩ +≡

```
ll_flt ll_point_step(ll_point *pt, UINT pos, UINT dur)
{
  return pt->step(pt, pt->data, pos, dur);
}
```

70. The function *ll_point_mem_callback* sets the memory allocation callbacks. This function may be called implicitely when setting memory allocation functions from higher abstractions.

⟨ The Point 47 ⟩ +≡

```
void ll_point_mem_callback(ll_point *pt, ll_cb_malloc m, ll_cb_free f)
{
  pt->malloc = m;
  pt->free = f;
}
```

71. The Line. A line is a sequence of points in time. A line smoothly steps through the points with some sort of interpolation.

⟨ Top 45 ⟩ +≡
  ⟨ The Line 73 ⟩

72. The ll_line Declaration.

73. The line is mostly a linked list, with a root value, a pointer to the value last appended, and the size.

⟨ The Line 73 ⟩ ≡
  struct ll_line {
  ll_point ∗*root*;
  ll_point ∗*last*;
  int *size*;
  int *curpos*; /∗ the current point position ∗/

See also sections 74, 75, 76, 77, 78, 79, 80, 81, 82, 84, 85, 87, 88, 89, 90, 92, 94, 95, 96, 97, 98, 99, 100, 102, 103, 104, and 105.
This code is used in section 71.

74. Since the line generated is a digital audio signal, it must have a sampling rate *sr*.

⟨ The Line 73 ⟩ +≡
  int *sr*;

75. A counter variable is used as a sample-accurate timer to navigate between sequential points.

⟨ The Line 73 ⟩ +≡
  unsigned int *counter*;

76. The duration of the current point is in stored in the variable *idur*. This unit of this duration is in whole-**i**nteger samples, which is the justification for the "i" in the beginning of the variable.

⟨ The Line 73 ⟩ +≡
  unsigned int *idur*;

77. The line interface can handle memory allocation internally. It does so using callback interfaces for allocating and freeing memory. By default, these functions are wrappers around the standard C *malloc* and *free* functions.

⟨ The Line 73 ⟩ +≡
  ll_cb_malloc *malloc*;
  ll_cb_free free;

78. The struct also has an entry for custom user data, defined as a void pointer *ud*.

⟨ The Line 73 ⟩ +≡
  void ∗*ud*;

79. The variable *end* is a boolean value that is set when the line reaches the end.

⟨ The Line 73 ⟩ +≡
  int *end*;

80. A timescaling variable speeds or slows down the units of time. This can be used to make the units of duration match to beats rather than just seconds.

⟨ The Line 73 ⟩ +≡
  ll_flt *tscale*;

81. When a line produces a sample of audio, it saves a copy of it to a pointer. By default, this pointer points to an internal value. However, it can be overriden later by other applications who wish to read the data directly.

⟨ The Line 73 ⟩ +≡
  ll_flt ∗*val*;
  ll_flt *ival*;
  } ;

82. The size of **ll_line** is implemented as a function.

⟨ The Line 73 ⟩ +≡
  size_t *ll_line_size*(**void**)
  {
    return sizeof(**ll_line**);
  }

83. Line Initalization.

84. After the line is allocated, it must be initialized. A line starts out with zero points. Pointers are set to be Λ (NULL). The memory allocation functions are set to defaults.

⟨ The Line 73 ⟩ +≡

```
void ll_line_init(ll_line *ln, int sr)
{
  ln→root = Λ;
  ln→last = Λ;
  ln→size = 0;
  ln→sr = sr;
  ln→malloc = ll_malloc;
  ln→free = ll_free;
  ln→idur = 0;
  ln→counter = 0;
  ln→curpos = 0;
  ln→end = 0;
  ln→tscale = 1.0;
  ln→val = &ln→ival;
}
```

85. The time scale of a line determines the rate at which line is stepped through. A value of 1 has the line move normally. A value of 0.5, twice the speed. A value of 2 at half-speed.

⟨ The Line 73 ⟩ +≡

```
void ll_line_timescale(ll_line *ln, ll_flt scale)
{
  ln→tscale = scale;
}
```

86. Appending a Point to a Line.

87. Points are added to a line in chronological order because they are appended to the end of a linked list.

A new line with zero points must set the root of the linked list with the added point. For the case when there are already items populated in the linked list, the last pointer entry is used. The "next" entry in this pointer is set to be the appended point. The "B" value of the last point is set to point to the "A" value of the appended point.

After the point is appended, the last point is set to be the appended point. The size of the line is incremented by 1.

⟨ The Line 73 ⟩ +≡

```
void ll_line_append_point(ll_line *ln, ll_point *p)
{
  if (ln→size ≡ 0) {
    ln→root = p;
  }
  else {
    ll_point_set_next_point(ln→last, p);
    ll_point_set_next_value(ln→last, ll_point_get_value(p));
  }
  ln→last = p;
  ln→size++;
}
```

88. The function *ll_line_append_point* assumes that memory is already allocated. This, however, is a very inconvenient burden for the programmer to keep track of. The function *ll_line_append* wraps around *ll_line_append_point* and uses the internal memory functions to allocate memory.

When the point is initialized, the memory functions used in the line are forwarded to the point callback via *ll_point_mem_callback*.

⟨ The Line 73 ⟩ +≡

```
ll_point *ll_line_append(ll_line *ln, ll_flt val, ll_flt dur)
{
  ll_point *pt;

  pt = ln→malloc(ln→ud, ll_point_size());
  ll_point_init(pt);
  ll_point_value(pt, val);
  ll_point_dur(pt, dur);
  ll_point_mem_callback(pt, ln→malloc, ln→free);
  ll_line_append_point(ln, pt);
  return pt;
}
```

89. Once points are doing being added to a line, it must be rewound and reset to the beginning.

⟨ The Line 73 ⟩ +≡

```
void ll_line_done(ll_line *ln)
{
  ln→curpos = 0;
  ln→last = ln→root;
  ln→idur = ll_point_get_dur(ln→root) * ln→sr * ln→tscale;
  ln→counter = ln→idur;
  ln→end = 0;
}
```

90. The function *ll_line_done* also can be called at any point to rewind the line to the beginning.

⟨ The Line 73 ⟩ +≡

```
void ll_line_reset(ll_line *ln)
{
  ll_line_done(ln);
}
```

91. Freeing Line Memory.

92. All things that must be allocated internally must then be freed using the function *ll_line_free*. This function essentially walks through the linked list and frees all the points.

⟨ The Line 73 ⟩ +≡

```
void ll_line_free(ll_line *ln)
{
  ll_point *pt;
  ll_point *next;
  unsigned int i;

  pt = ln→root;
  for (i = 0; i < ln→size; i++) {
    next = ll_point_get_next_point(pt);
    ll_point_destroy(pt);
    ln→free(ln→ud, pt);
    pt = next;
  }
}
```

93. Line Step Function.

94. *ll_line_step* is the top-level function that computes the line. This is done through both ticking down the timer and walking through the linked list.

⟨ The Line 73 ⟩ +≡
```
ll_flt ll_line_step(ll_line *ln){ UINT dur;
    UINT pos;
```

95. If the line has ended, the step value is simply the "A" value of the point. The function returns early with this value. If the line has not ended, the routine moves forward.

⟨ The Line 73 ⟩ +≡
```
if (ln→end) {
  return ll_point_A(ln→last);
}
```

96. There is a check to see if the counter has ticked to zero.

⟨ The Line 73 ⟩ +≡
```
if (ln→counter ≡ 0) {
```

97. If the counter is zero, there is a check to see if there are any points left in the list. This is done by comparing the current point position with the size of the of the list. Note that since the current point position is zero indexed, the size is subtracted by 1.

⟨ The Line 73 ⟩ +≡
```
if (ln→curpos < (ln→size − 1)) {
```

98. If the line is not at the end, then it will step to the next point in the linked list. The duration in samples is computed, the counter is reset, and the position is incremented by one.

⟨ The Line 73 ⟩ +≡
```
ln→last = ll_point_get_next_point(ln→last);
ln→idur = ll_point_get_dur(ln→last) ∗ ln→sr ∗ ln→tscale;
ln→counter = ln→idur;
ln→curpos++;
```

99. If there are no points left in the list, the line has ended, and the *end* variable is turned on. This concludes both nested conditionals.

⟨ The Line 73 ⟩ +≡
```
}
else {
  ln→end = 1;
}
}
```

100. The step function inside the point is called. The current point position is a value derived from the counter. Since the counter moves backwards, it is subtracted for the total line duration. The counter is then decremented right before the point step function is called.

⟨ The Line 73 ⟩ +≡
```
dur = ln→idur;
pos = dur − ln→counter;
ln→counter−−;
∗ln→val = ll_point_step(ln→last, pos, dur);
return ∗ln→val; }
```

101. Other Functions. The following section is for functions that couldn't quite fit anywhere else.

102. Sometimes it can be useful to print points in a line. *ll_line_print* does just that, walking through the list and printing the values.

⟨ The Line 73 ⟩ +≡
```
void ll_line_print(ll_line *ln)
{
  ll_point *pt;
  ll_point *next;
  unsigned int i;
  ll_flt *val;
  pt = ln→root;
  printf("there␣are␣%d␣points...\n", ln→size);
  for (i = 0; i < ln→size; i++) {
    next = ll_point_get_next_point(pt);
    val = ll_point_get_value(pt);
    printf("point␣%d:␣dur␣%g,␣val␣%g\n", i, ll_point_get_dur(pt), *val);
    pt = next;
  }
}
```

103. In Sporth, it is a better arrangement to have a Sporth float be injected into libline, rather than a libline float injected into Sporth. This function binds a float pointer to a line.

⟨ The Line 73 ⟩ +≡
```
void ll_line_bind_float (ll_line *ln, ll_flt * line ) { ln→val = line ; }
```

104. To access all points in a line, one only needs the top point. Since points are entries in a linked list, one can step through the line using *ll_get_next_point*.

⟨ The Line 73 ⟩ +≡
```
ll_point *ll_line_top_point(ll_line *ln)
{
  return ln→root;
}
```

105. The function *ll_line_npoints* returns the number of points in a line.

⟨ The Line 73 ⟩ +≡
```
int ll_line_npoints(ll_line *ln)
{
  return ln→size;
}
```

106. The Lines. When more than one line is required, you need *lines*. **ll_lines** is the next abstraction up from **ll_line**.

⟨ Top 45 ⟩ +≡
  ⟨ Lines 108 ⟩

107. The ll_lines Declaration.

108. The *ll_line_entry* data struct wraps **ll_line** into a linked list entry.

⟨ Lines 108 ⟩ ≡

```
typedef struct ll_line_entry {
  ll_line *ln;    /* main ll_line entry */
  ll_flt val;    /* store output step value */
  struct ll_line_entry *next;    /* next ll_line_entry value */
} ll_line_entry;
```

See also sections 109, 111, 112, 114, 115, 117, 118, 120, and 121.

This code is used in section 106.

109. The **ll_lines** data struct is linked list of **ll_line_entry**.

⟨ Lines 108 ⟩ +≡

```
struct ll_lines {
  ll_line_entry *root;
  ll_line_entry *last;
  unsigned int size;
  int sr;    /* samplerate */
  ll_cb_malloc malloc;
  ll_cb_freefree;
  void *ud;
  ll_line *ln;
  ll_point *pt;
  ll_flt tscale;
};
```

110. Lines Initialization.

111. *ll_lines_size* returns the size of the ll_lines data struct.

⟨ Lines 108 ⟩ +≡
```
size_t ll_lines_size()
{
  return sizeof(ll_lines);
}
```

112. *ll_lines_init* initializes all the data of an allocated **ll_lines** struct.

⟨ Lines 108 ⟩ +≡
```
void ll_lines_init(ll_lines *l, int sr)
{
  l→root = Λ;
  l→last = Λ;
  l→size = 0;
  l→malloc = ll_malloc;
  l→free = ll_free;
  l→sr = sr;
  l→tscale = 1.0;
}
```

113. Lines Memory Handling.

114. Alternative memory allocation functions can be set for **ll_lines** via *ll_lines_mem_callback*.

⟨ Lines 108 ⟩ +≡

```
void ll_lines_mem_callback(ll_lines *l, void *ud, ll_cb_malloc m, ll_cb_free f)
{
  l→malloc = m;
  l→free = f;
  l→ud = ud;
}
```

115. Write some words here.

⟨ Lines 108 ⟩ +≡

```
void ll_lines_free(ll_lines *l)
{
  unsigned int i;
  ll_line_entry *entry;
  ll_line_entry *next;
  entry = l→root;
  for (i = 0; i < l→size; i++) {
    next = entry→next;
    ll_line_free(entry→ln);
    l→free(l→ud, entry→ln);
    l→free(l→ud, entry);
    entry = next;
  }
}
```

116. Appending a Line to Lines.

117. This creates and appends a new **ll_line** to the **ll_lines** linked list. The address of this new **ll_line** is saved to the variable *pline*. The output memory address of the **ll_line** is saved to the variable *val*.

```
⟨ Lines 108 ⟩ +≡
  void ll_lines_append(ll_lines *l, ll_line **pline, ll_flt **val)
  {
    ll_line_entry *entry;
    entry = l→malloc(l→ud, sizeof(ll_line_entry));
    entry→val = 0.F;
    entry→ln = l→malloc(l→ud, ll_line_size());
    ll_line_init(entry→ln, l→sr);
    ll_line_timescale(entry→ln, l→tscale);
    if (pline ≠ Λ) *pline = entry→ln;
    if (val ≠ Λ) *val = &entry→val;
    if (l→size ≡ 0) {
      l→root = entry;
    }
    else {
      l→last→next = entry;
    }
    l→size++;
    l→last = entry;
    l→ln = entry→ln;
  }
```

118. The current line being created can be returned using a wrapper function called *ll_lines_get_current*. This function is needed in order to get line data bound to data in Sporth.

```
⟨ Lines 108 ⟩ +≡
  ll_line *ll_lines_current_line(ll_lines *l)
  {
    return l→ln;
  }
```

119. Lines Step Function.

120. The step function for **ll_lines** will walk through the linked list and call the step function for each **ll_line** inside each **ll_line_entry**.

⟨ Lines 108 ⟩ +≡

```
void ll_lines_step(ll_lines *l)
{
  unsigned int i;
  ll_line_entry *entry;

  entry = l→root;
  for (i = 0; i < l→size; i++) {
    entry→val = ll_line_step(entry→ln);
    entry = entry→next;
  }
}
```

121. Wrappers for adding points. The Line API provides a set of high-level functions for populating lines with points. These use functions abstract away some of the C structs needed, making it easier to export to higher-level languages like Lua.

⟨ Lines 108 ⟩ +≡

```
void ll_add_linpoint(ll_lines *l, ll_flt val, ll_flt dur)
{
  ll_point *pt;

  pt = ll_line_append(l→ln, val, dur);
  ll_linpoint(pt);
}
void ll_add_exppoint(ll_lines *l, ll_flt val, ll_flt dur, ll_flt curve)
{
  ll_point *pt;

  pt = ll_line_append(l→ln, val, dur);
  ll_exppoint(pt, curve);
}
void ll_add_step(ll_lines *l, ll_flt val, ll_flt dur)
{
  ll_line_append(l→ln, val, dur);
}
void ll_add_tick(ll_lines *l, ll_flt dur)
{
  ll_point *pt;

  pt = ll_line_append(l→ln, 0.0, dur);
  ll_tick(pt);
}
void ll_end(ll_lines *l)
{
  ll_line_done(l→ln);
}
void ll_timescale(ll_lines *l, ll_flt scale)
{
  l→tscale = scale;
}
void ll_timescale_bpm(ll_lines *l, ll_flt bpm)
{
  l→tscale = 60.0/bpm;
}
```

122. Linear Points. Linear points create a straight line from point A to point B.

⟨ Top 45 ⟩ +≡
  ⟨ Linear Point 123 ⟩

123. The main data structure for a linear point contains an incrementor value *inc* and an accumulator value *acc*.

⟨ Linear Point 123 ⟩ ≡

```
typedef struct {
  ll_flt inc;
  ll_flt acc;
} linpoint;
```

See also section 124.

This code is used in section 122.

124. The setup function for linpoint allocates the memory needed for the **linpoint** struct, then binds it and the step callback to the point.

⟨ Linear Point 123 ⟩ +≡
  ⟨ Private Functions for Linear Point 125 ⟩

```
void ll_linpoint(ll_point *pt)
{
  linpoint *lp;
  lp = ll_point_malloc(pt, sizeof(linpoint));
  ll_point_cb_step(pt, linpoint_step);
  ll_point_data(pt, lp);
  ll_point_cb_destroy(pt, ll_linpoint_destroy);
}
```

125. The linear step function is reasonably straightforward. When the line position is zero, the incrementor and acculumaltor values are implemented. Next, the current value of the acculumator is returned and then incremented.

⟨ Private Functions for Linear Point 125 ⟩ ≡

```
static ll_flt linpoint_step(ll_point *pt, void *ud, UINT pos, UINT dur)
{
  linpoint *lp;
  ll_flt val;
  lp = ud;
  if (pos ≡ 0) {
    lp→acc = ll_point_A(pt);
    lp→inc = (ll_point_B(pt) − ll_point_A(pt))/dur;
  }
  val = lp→acc;
  lp→acc += lp→inc;
  return val;
}
```

See also section 126.

This code is used in section 124.

126. The destroy function for linpoint destroys the memory previously allocated.

⟨ Private Functions for Linear Point 125 ⟩ +≡

```
static void ll_linpoint_destroy(void *ud, void *ptr)
{
  ll_point *pt;
  pt = ud;
  ll_point_free(pt, ptr);
}
```

127. Exponential Points.

⟨ Top 45 ⟩ +≡
⟨ Exponential Point 128 ⟩

128. Exponential Point Data.

⟨ Exponential Point 128 ⟩ ≡
```
typedef struct {
  SPFLOAT curve;
} exppoint;
```
See also section 129.
This code is used in section 127.

129. This function sets up the exponential point data struct **exppoint**.

⟨ Exponential Point 128 ⟩ +≡
```
⟨ Private Functions For Exponential Points 130 ⟩ void ll_exppoint(ll_point *pt, ll_flt curve)
    {
      exppoint *ep;
      ep = ll_point_malloc(pt, sizeof(exppoint));
      ep→curve = curve;
      ll_point_cb_step(pt, exppoint_step);
      ll_point_data(pt, ep);
      ll_point_cb_destroy(pt, exppoint_destroy);
    }
```

130. The exponential step function uses the following equation:

$$y = A + (B - A) \cdot (1 - e^{tc/(N-1)})/(1 - e^c)$$

Where:
y is the output value.
A is the start value.
B is the end value.
t is the current sample position.
N is the duration of the line segment, in samples.
c determines the slope of the curve. When $c = 0$, a linear line is produced. When $c > 0$, the curve slowly rises (concave) and decays (convex). When $c < 0$, the curve quickly rises (convex) and decays (concave).

⟨ Private Functions For Exponential Points 130 ⟩ ≡
```
static ll_flt exppoint_step(ll_point *pt, void *ud, UINT pos, UINT dur)
{
  exppoint *ep;
  ll_flt val;
  ep = ud;
  val = ll_point_A(pt) + (ll_point_B(pt) − ll_point_A(pt)) * (1 − exp(pos * ep→curve/(dur − 1)))/(1 −
      exp(ep→curve));
  return val;
}
```
See also section 131.
This code is used in section 129.

131.

⟨ Private Functions For Exponential Points 130 ⟩ +≡

```
static void exppoint_destroy(void *ud, void *ptr)
{
  ll_point *pt;
  pt = ud;
  ll_point_free(pt, ptr);
}
```

132. Bezier Points. Bezier points are used to create quadratic bezier curves.

⟨ Top 45 ⟩ +≡
  ⟨ Bezier Point 133 ⟩

133. Bezier Point Data. More to write here.

⟨ Bezier Point 133 ⟩ ≡

```
typedef struct {
  ll_flt cx;
  ll_flt cy;
} bezierpt;
```

See also section 134.

This code is used in section 132.

134. Bezier Setup. The setup function for producing a bezier curve is *ll_bezier*. It takes in two arguments for a control point. The X value is normalized between 0 and 1.

⟨ Bezier Point 133 ⟩ +≡
⟨ Static Functions for Bezier Point 135 ⟩

```
void ll_bezier(ll_point *pt, ll_flt cx, ll_flt cy)
{
  bezierpt *b;

  b = ll_point_malloc(pt, sizeof(bezierpt));
  b→cx = cx;
  b→cy = cy;
  ll_point_cb_step(pt, bezier_step);
  ll_point_data(pt, b);
  ll_point_cb_destroy(pt, bezier_destroy);
}
```

135. This is the bezier curve step function which computes a quadratic bezier line. The quadratic equation for a Bezier curve is the following:

$$B(t) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2$$

Where t is a normalized time value between 0 and 1, and P_n refers to a 2-dimensional point with a (x, y) coordinate.

The issue with the classic equation above is that the value is derived from t, allowing x to be fractional. This is problematic because the system implemented here is discrete, restricted to whole-integer values of x.

The solution to this problem is to rework the equation above to solve for t in terms of the current sample position x_n. Once t is found, it can be used to compute the result, which is the y component of the bezier curve in terms of t. The reworked bezier equation is touched upon in greater detail in the ⟨ Quadratic Equation Solver 136 ⟩ section.

⟨ Static Functions for Bezier Point 135 ⟩ ≡
⟨ Quadratic Equation Solver 136 ⟩

```
static ll_flt bezier_step(ll_point *pt, void *ud, UINT pos, UINT dur)
{
  bezierpt *bez;
  ll_flt x[3];
  ll_flt y[3];
  ll_flt t;
  ll_flt val;

  bez = ud;
  x[0] = 0.F;   /* always zero */
  x[1] = bez→cx * dur;
  x[2] = dur;
  y[0] = ll_point_A(pt);
  y[1] = bez→cy;
  y[2] = ll_point_B(pt);
  t = find_t(x[0], x[1], x[2], pos);
  val = (1.0 - t) * (1.0 - t) * y[0] + 2.0 * (1.0 - t) * t * y[1] + t * t * y[2];
  return val;
}
```

See also section 138.

This code is used in section 134.

136. The function below implements a quadratic equation solver for all real values. Imaginary values return a value of 0.

⟨ Quadratic Equation Solver 136 ⟩ ≡

```
static ll_flt quadratic_equation(ll_flt a, ll_flt b, ll_flt c)
{
  ll_flt det;    /* determinant */

  det = b * b - 4 * a * c;
  if (det ≥ 0) {
    return ((-b + sqrt(det))/(2.0 * a));
  }
  else {
    return 0;
  }
}
```

See also section 137.

This code is cited in section 135.

This code is used in section 135.

137. The Bezier x component B_x can be rearranged to be a quadratic equation for t, given the x bezier control points x_0, x_1, and x_2, as well as the current sample position x_n.

$$
\begin{aligned}
x_n &= (1-t)^2 x_0 + 2(1-t)t x_1 + t^2 x_2 \\
&= (1 - 2t + t^2)x_0 + (2t - 2t^2)x_1 + t^2 x_2 \\
&= x_0 - 2t x_0 + t^2 x_0 + 2t x_1 - 2t^2 x_1 + t^2 x_2 \\
&= (x_0 - 2x_1 + x_2)t^2 + 2(-x_0 + x_1)t + x_0 \\
0 &= (x_0 - 2x_1 + x_2)t^2 + 2(-x_0 + x_1)t + x_0 - x_n
\end{aligned}
$$

This yields the following a, b, and c quadratic equation coefficients:

$$
\begin{aligned}
a &= x_0 - 2x_1 + x2, \\
b &= 2(-x_0 + x_1) \\
c &= x_0 - x_n
\end{aligned}
$$

Using those variables, the value of t can be found if it is a real value.

⟨ Quadratic Equation Solver 136 ⟩ +≡

```
static ll_flt find_t(ll_flt x0, ll_flt x1, ll_flt x2, int x)
{
  ll_flt a;
  ll_flt b;
  ll_flt c;

  a = (x0 - 2.0 * x1 + x2);
  b = 2.0 * (-x0 + x1);
  c = x0 - x;
  if (a) {
    return quadratic_equation(a, b, c);
  }
  else {
    return (x - x0)/b;
  }
}
```

138. The bezier destroy function frees the data allocated by the bezier.

⟨ Static Functions for Bezier Point 135 ⟩ +≡

```
static void bezier_destroy(void *ud, void *ptr)
{
  ll_point *pt;
  pt = ud;
  ll_point_free(pt, ptr);
}
```

139. Tick. In Sporth, a tick is a single non-zero sample used as a trigger signal for trigger-based unit generators. Ticks can be used as a kind of line to produce these kind of control signals.

⟨ Top 45 ⟩ +≡
  ⟨ Tick 140 ⟩

140. The tick step function will only produce a non-zero value if the position is zero.

⟨ Tick 140 ⟩ ≡
```
static ll_flt tick_step(ll_point *pt, void *ud, UINT pos, UINT dur)
{
  if (pos ≡ 0) {
    return 1.0;
  }
  else {
    return 0.0;
  }
}
```
See also section 141.

This code is used in section 139.

141. The tick initialization function binds *tick_step* to the step function.

⟨ Tick 140 ⟩ +≡
```
void ll_tick(ll_point *pt)
{
  ll_point_cb_step(pt, tick_step);
}
```

142. Memory. Several aspects of this program require memory to be allocated. In order to be maximally flexible, the default system memory handling functions have been wrapped inside helper functions with a void pointer for user data. This way, these functions can be swapped out for custom ones for situations where a different memory handling system is used, such as garbage collection.

⟨ Top 45 ⟩ +≡

```
void *ll_malloc(void *ud, size_t size)
{
  return malloc(size);
}
void ll_free(void *ud, void *ptr)
{
  free(ptr);
}
void ll_free_nothing(void *ud, void *ptr)
{ }
```

143. Lines in Sporth. An optional feature of this line library is to have hooks into the Sporth programming language via the Sporth API.

⟨ Top 45 ⟩ +≡
```
#ifdef LL_SPORTH
  ⟨ Sporth 144 ⟩
#endif
```

144. In order to use Lines in Sporth, it needs to be registered as a Sporth unit generator. This unit generator will handle initialization and tear-down of **ll_lines**, as well as step through all the lines at every sample. This unit generator should be instantiated exactly once at the top of the Sporth patch.

⟨ Sporth 144 ⟩ ≡
```
  ⟨ The Sporth Unit Generators 146 ⟩
      void ll_sporth_ugen(ll_lines *l, plumber_data * pd, const char *ugen)
      {
        plumber_ftmap_add_function(pd, ugen, sporth_ll, l);
      }
```
See also sections 145 and 148.

This code is used in section 143.

145. *ll_sporth_line* registers a line as a named variable.

⟨ Sporth 144 ⟩ +≡
```
  ll_line *ll_sporth_line(ll_lines *l, plumber_data * pd, const char *name)
  {
    ll_line *ln;
    SPFLOAT * val;
    int rc;
    ll_lines_append(l, &ln, Λ);
    rc = plumber_ftmap_search_userdata(pd, name, (void **) &val);
    if (rc ≡ PLUMBER_NOTOK) {
      plumber_create_var(pd, name, &val);
    }
    ll_line_bind_float(ln, val);
    return ln;
  }
```

146. The following is the Sporth unit generator callback used inside of Sporth.

⟨ The Sporth Unit Generators 146 ⟩ ≡
```
  static int sporth_ll(plumber_data * pd, sporth_stack * stack, void **ud)
  {
    ll_lines *l;
    l = *ud;
    if (pd→mode ≡ PLUMBER_COMPUTE) ll_lines_step(l);
    return PLUMBER_OK;
  }
```
See also section 147.

This code is used in section 144.

147. Triggers in Sporth can be leveraged to schedule lines. After creating a new line for Sporth to read via *ll_sporth_line*, a ugen can be created to reset that line with a trigger via *ll_line_reset*.

⟨ The Sporth Unit Generators 146 ⟩ +≡

```
static int sporth_ll_reset(plumber_data *pd, sporth_stack *stack, void **ud)
{
  ll_line *ln;
  SPFLOAT tick;
  ln = *ud;
  switch (pd→mode) {
  case PLUMBER_COMPUTE: tick = sporth_stack_pop_float(stack);
    if (tick) {
      ll_line_reset(ln);
    }
    break;
  case PLUMBER_CREATE: case PLUMBER_INIT: sporth_stack_pop_float(stack);
    break;
  }
  return PLUMBER_OK;
}
```

148. The ugen function must be bound to a named ftable in Sporth, where the user data is the line.

⟨ Sporth 144 ⟩ +≡

```
void ll_sporth_reset_ugen(ll_lines *l, plumber_data *pd, const char *ugen)
{
  ll_line *ln;
  ln = ll_lines_current_line(l);
  plumber_ftmap_add_function(pd, ugen, sporth_ll_reset, ln);
}
```

LIBLINE