Potrace Library API

1 Scope

The Potrace library provides:

- tracing, i.e., conversion of bitmaps to a vector representation (Bezier curves and straight line segments).

It does not provide frontend functionality such as:

- preparation of bitmaps (e.g. reading a bitmap from a file, preparing a bitmap by thresholding/scaling/filtering a greyscale image etc)

And it does not provide backend functionality such as:

- post-processing of the vector representation (e.g. conversion to a file format such as PostScript or SVG, scaling + rotation, quantization etc).

2 Data representation

2.1 Bitmaps

2.1.1 Coordinate system

For Potrace, a bitmap of size $w \times h$ is embedded in a cartesian coordinate system where each pixel takes up the space of one unit square. The pixels are positioned so that the *corners* of pixels (and not their centers) lie at points with integer coordinates, as illustrated in Figure 1. The origin of the coordinate system is at the *lower left* corner of the bitmap. The four corners of the bitmaps have coordinates $(0,0)$, $(0,h)$, (w,h), and $(w,0)$.

Sometimes we need to refer to a specific pixel (as opposed to a point in the plane). When we speak of "pixel $[i,j]$", we mean the pixel whose corners have coordinates (i,j), $(i,j+1)$, $(i+1,j+1)$, $(i+1,j)$ in Potrace's coordinate system. Thus, pixel $[i,j]$ is the pixel whose center is at coordinates $(i+0.5, j+0.5)$. To avoid confusion, we use square brackets to refer to the pixel $[i,j]$, and round brackets to refer to the point (i,j).

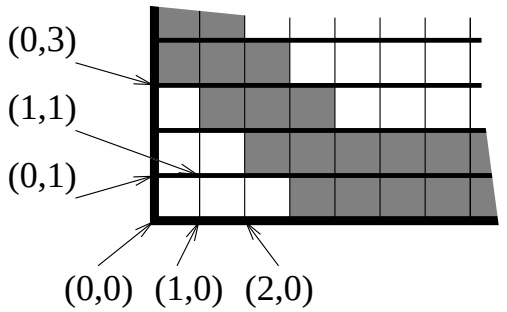


Figure 1: The Potrace coordinate system

2.1.2 Bitmap representation

The Potrace library expects bitmaps in the following format, defined in potracelib.h:

```
struct potrace_bitmap_s {
  int w, h;          /* width and height, in pixels */
  int dy;            /* scanline offset in words */
  potrace_word *map; /* pixel data, dy*h words */
};
typedef struct potrace_bitmap_s potrace_bitmap_t;
```

Here, `potrace_word` is an unsigned integer type defined in potracelib.h. It is usually equal to a native machine word (i.e., 32 bits on a 32-bit architecture). In the following explanation, we assume that the type `potrace_word` holds N bits.

A bitmap of dimensions $w \times h$ is divided, bottom to top, into h horizontal scanlines. Each scanline is divided, left to right, into blocks of N pixels. Each such block of N pixels is stored as a single `potrace_word`, with the leftmost pixels of the block corresponding to the most significant bit of the word, and the rightmost pixel of the block corresponding to the least significant bit of the word.

Pixels that are "on" (or "black" or "foreground") are represented by bit value 1. Pixels that are "off" (of "white" or "background") are represented by bit value 0.

If the number of bits in a scanline is not divisible by N, then the rightmost word of the scanline is padded on the right with zeros.

The data for scanline 0 (the bottom-most scanline) begins at `map[0]`. The data for scanline 1 begins at `map[dy]`. The data for scanline 2 begins at `map[2*dy]`, and so forth. Note that dy can be either positive or negative, depending on how an application wishes to lay out the image data in memory.

In summary, the pixel with coordinates $[i,j]$ can be accessed by the following C formula:

```
pixel(i,j) = ((map + j*dy)[i/N] & (1 << (N-1-i%N)) ? 1 : 0.
```

2.1.3 Example

Figure 2 shows an example bitmap of size 36×12. Shaded pixels are "on" and white pixels are "off". Figure 3 shows a possible representation of this bitmap in the `potrace_bitmap_t` data structure. Note that the data is stored in the `map` array in a bottom-to-top and left-to-right fashion.

2.1.4 A remark on byte order

It is important to keep in mind that bitmaps are stored as arrays of words, *not* as arrays of bytes. While this distinction makes no difference on big-endian architectures, it makes a significant difference on little-endian architectures such as the Intel-based architecture. For instance, when the integer word 0x1f80fc02 is accessed as a byte-array on a little-endian machine, then the bytes appear in reverse order 0x02, 0xfc, 0x80, 0x1f. Therefore, special care must be taken when converting a bitmap from a byte-based format to Potrace's word-based format.

2.1.5 Coordinate independence

The vector data that is the output of Potrace is taken with respect to the same coordinate system as the input bitmap, i.e., the coordinate system from Figure 1. In principle, it is immaterial whether an application puts the coordinate origin in the bottom-left corner or the top-left corner of an image, as long as it interprets the output coordinates in the same way as the input coordinates.

However, a reversal of the coordinate system will upset the meaning of the words "clockwise" and "counterclockwise" in the specification of vector images below (see Section 2.2.5), and will also affect the meaning of Potrace's turnpolicies (see Section 2.3). We therefore assume, for definiteness, that the coordinate origin is in the *lower* left corner. Applications that wish to follow a different convention have to compensate accordingly.

2.2 Vector format

2.2.1 Points

A point (x, y) in the Euclidean plane is represented in Potrace by a value of type `potrace_dpoint_t`.

```
struct potrace_dpoint_s {
  double x, y;
};
typedef struct potrace_dpoint_s potrace_dpoint_t;
```

2.2.2 Segments

Curves in Potrace are composed of the following two types of segments:

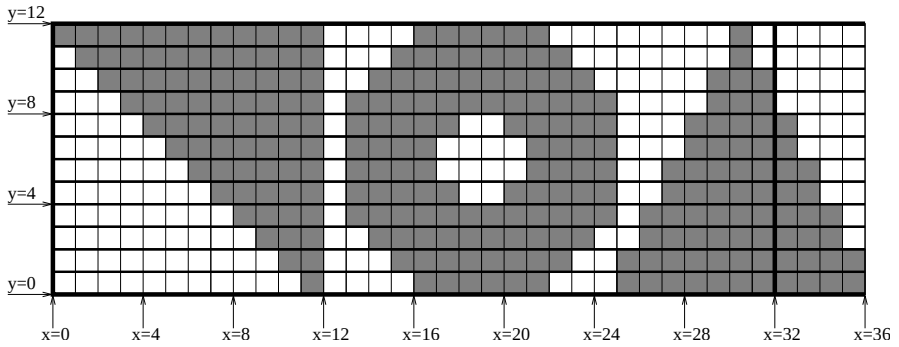


Figure 2: Sample bitmap

```
w = 36;
h = 12;
dy = 2;
map[22] = 0xfff0fc02;    map[23] = 0x00000000;
map[20] = 0x7ff1fe02;    map[21] = 0x00000000;
map[18] = 0x3ff3ff07;    map[19] = 0x00000000;
map[16] = 0x1ff7ff87;    map[17] = 0x00000000;
map[14] = 0x0ff7cf8f;    map[15] = 0x80000000;
map[12] = 0x07f7878f;    map[13] = 0x80000000;
map[10] = 0x03f7879f;    map[11] = 0xc0000000;
map[8]  = 0x01f7cf9f;    map[9]  = 0xc0000000;
map[6]  = 0x00f7ffbf;    map[7]  = 0xe0000000;
map[4]  = 0x0073ff3f;    map[5]  = 0xe0000000;
map[2]  = 0x0031fe7f;    map[3]  = 0xf0000000;
map[0]  = 0x0010fc7f;    map[1]  = 0xf0000000;
```

Figure 3: Sample bitmap representation

Figure 4: (a) A Bezier curve segment. (b) A corner segment

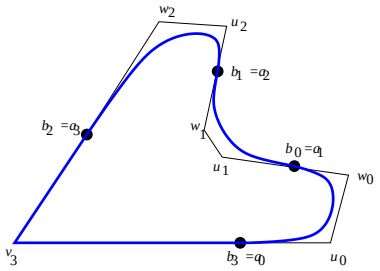


Figure 5: A closed curve consisting of 4 segments

- Bezier curve segments. A Bezier curve segment is given in the usual way by a starting point a, two control points u and w, and an endpoint b, as shown in Figure 4(a).
- Corner segments. A corner segment is given by a starting point a, a vertex v, and an endpoint b. A corner segment is drawn as two straight lines: one from a to v, and one from v to b, as shown in Figure 4(b).

2.2.3 Curves

A curve in Potrace is a sequence of segments, such that the endpoint of each segment coincides with the starting point of the next one. All curves in Potrace are closed, and therefore the endpoint of the final segment also coincides with the starting point of the first one. Figure 5 shows an example of a curve consisting of 4 segments: 3 Bezier curve segments and 1 corner segment. For clarity, the start- and endpoints of segments have been marked with dots "•".

Curves are represented as values of type `potrace_curve_t`, which is defined as follows:

```
struct potrace_curve_s {
  int n;                    /* number of segments */
  int *tag;                 /* array of segment types */
  potrace_dpoint_t (*c)[3]; /* array of control points. */
};
typedef struct potrace_curve_s potrace_curve_t;
```

Here $n \geq 1$ is the number of segments in the curve. For $i = 0, \ldots, n-1$, `tag[i]` is the type of the i-th segment, which is `POTRACE_CURVETO` for a Bezier curve segment and `POTRACE_CORNER` for a corner segment. c is an array of size $n \times 3$ that holds the control points of the curve segments in the following manner:

- If the i-th segment is a Bezier curve segment, then `c[i][0]` $= u$ and `c[i][1]` $= w$ are the two control points of that segment, and `c[i][2]` $= b$ is its endpoint.

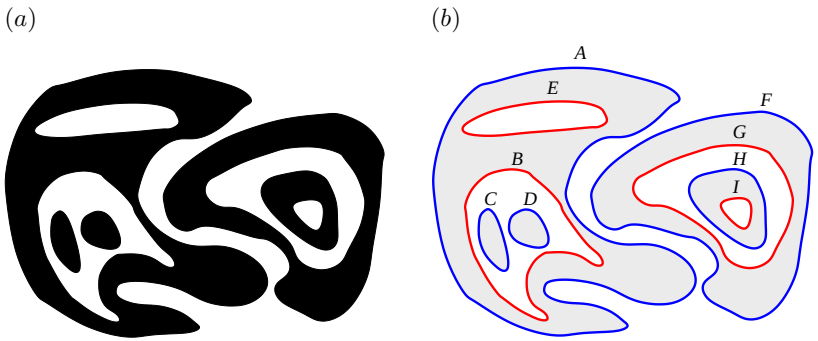


Figure 6: (a) A vector image. (b) Its boundary decomposition.

- If the i-th segment is a corner segment, then `c[i][0]` is unused, `c[i][1]` $= v$ is the vertex of the segment, and `c[i][2]` $= b$ is its endpoint.

Note that, since the starting point a of each segment coincides with the endpoint b of the preceding segment (and the starting point a of the first segment coincides with the endpoint b of the last segment), there is no need to store the starting points a explicitly. Also, note that regardless of the type of segment, the endpoint of the i-th segment is `c[i][2]`, and the starting point of the i-th segment is `c[i ? i-1 : n-1][2]`.

The curve from Figure 5 is therefore represented by the following data:

```
n = 4;
tag[0] = POTRACE_CURVETO;
c[0][0] = u0;     c[0][1] = w0;  c[0][2] = b0 = a1;
tag[1] = POTRACE_CURVETO;
c[1][0] = u1;     c[1][1] = w1;  c[1][2] = b1 = a2;
tag[2] = POTRACE_CURVETO;
c[2][0] = u2;     c[2][1] = w2;  c[2][2] = b2 = a3;
tag[3] = POTRACE_CORNER;
c[3][0] = unused; c[3][1] = v3;  c[3][2] = b3 = a0;
```

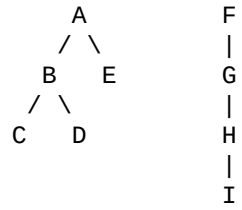
2.2.4 Boundary decomposition of bitonal vector images

In Potrace, a bitonal (i.e. black-and-white) vector image, as in Figure 6(a), is decomposed into a collection of closed boundary curves, shown in blue and red and labeled A–I in Figure 6(b).

We introduce some terminology. A closed curve is *simple* if it does not intersect itself. Each simple closed curve, taken by itself, divides the plane into two regions, called the *inside* and and the *outside* of the curve. If C_1 and C_2 are simple closed curves, we say that C_1 is *contained* in C_2, written $C_1 < C_2$, if C_1 lies entirely within the inside of C_2. For example, in Figure 6(b), the curves B–E are contained in A, whereas F–I are not.

In a decomposition of a vector image as in Figure 6, we say that a curve C_1 is a *child* of C_2 if $C_1 < C_2$ and there exists no other curve C_3 between C_1 and C_2 (i.e., no curve C_3 such that $C_1 < C_3 < C_2$). In this case, we also say that C_1 is a *parent* of C_2. Since boundary curves do not intersect, each curve has at most one parent. Two curves are said to be *siblings* if they are either both parentless, or else they have a parent in common. Note that the "child" relation naturally defines a tree structure on the set of curves (more precisely, it defines a "forest", since there can be more than one root).

For example, in Figure 6(b), the curve A has no parent, and has children B and E. The curve E has no children, and the curve B has children C and D. A and F are siblings, B and E are siblings, and C and D are siblings. The curves from Figure 6(b) form the following forest under the "child" relation:



We can assign each curve a *sign* by calling a curve *positive* if it encloses a "foreground" region, and *negative* if it encloses a "background" region (or "hole"). For example, in Figure 6(b), positive curves are shown in blue and negative curves in red.

Since foreground and background regions alternate, it follows that the sign of curves also alternates, i.e., parentless curves are always positive, and all other curves have the opposite sign of their parent. It follows that, in the tree structure, curves that appear at even levels are positive and those that appear at odd levels are negative. In particular, siblings share a common sign.

2.2.5 Representation of vector images

In Potrace, a vector image is represented as a linked collection of zero or more structures of type `potrace_path_t`, which is defined as follows:

```
struct potrace_path_s {
  int area;                       /* enclosed area */
  int sign;                       /* '+' or '-' */
  potrace_curve_t curve;          /* vector data */

  struct potrace_path_s *next;    /* list structure */

  struct potrace_path_s *childlist; /* tree structure */
  struct potrace_path_s *sibling;   /* tree structure */

  struct potrace_privpath_s *priv;  /* private state */
};
typedef struct potrace_path_s potrace_path_t;
```


Each such structure holds a single curve, and the structures are linked to each other via the `next`, `childlist`, and `sibling` pointers.

- The `sign` field holds the sign of the curve (`'+'` or `'-'` in ASCII).
- The `curve` field contains the curve's vector data as described in Section 2.2.3. Potrace additionally follows the convention that positive curves run counterclockwise and negative curves run clockwise; this facilitates rendering in environments (such as PostScript or PDF) that have a "fill" rule based on winding number.
- The `area` field gives the approximate magnitude of the area enclosed by the curve. (In fact, it is the precise integer area of the original untraced "jaggy" curve). Some clients use this information to improve interactive rendering speeds by ignoring very small areas in a first rendering pass. See also the description of the `turdsize` parameter in Section 2.3 below.
- The `priv` field is used internally by Potrace, and is not accessible to applications.
- The `next` field is used to link all the curves of a given vector image into a linked list. Each member points to the next one via its `next` field, and the last member of the list has `next==NULL`. The order of the elements of this list is unspecified, but is guaranteed to satisfy the following constraints:

  (a) outer curves appear before inner ones, so if $C_1 < C_2$, then C_2 always appears sometime before C_1 in the linked list, and

  (b) each positive curve is immediately followed by all of its children.

  These two constraints make it easy for clients to render the image by simply processing the linked list in sequential order. Constraint (a) makes it possible to fill each curve with solid black or white color, allowing later curves to paint over parts of earlier ones. Constraint (b) further allows a client to fill a positive curve, minus its negative children, in a single paint operation, leaving a "hole" for each of the negative children.
- The `childlist` and `sibling` fields define a forest structure on the set of curves, which can be used independently of the linked list structure. For each curve, `childlist` is a pointer to its first child, or `NULL` if there are no children. Also, `sibling` is a pointer to the next sibling, or `NULL` if there are no further siblings. The relative order of siblings is unspecified. The root node of the tree structure always coincides with the root node of the linked list structure.

An image consisting of zero curves is represented as a `NULL` pointer.

2.2.6 Intersecting curves

While in the above discussion we have assumed a set of non-intersecting curves, in practice it can happen that the curves output by Potrace intersect slightly. Clients should therefore carefully choose their rendering parameters (e.g., the non-zero winding number rule is preferable to the odd winding number rule) to avoid undesirable artifacts.

2.2.7 Example

The image from Figure 6 can be represented by the pointer `plist`, where `A`–`I` are structures of type `potrace_path_t`, as follows. We do not show the `area` and `curve` fields.

```
potrace_path_t *plist = &A;

A.sign = '+';       B.sign = '-';       E.sign = '-';
A.next = &B;        B.next = &E;        E.next = &C;
A.childlist = &B;   B.childlist = &C;   E.childlist = NULL;
A.sibling = &F;     B.sibling = &E;     E.sibling = NULL;

C.sign = '+';       D.sign = '+';       F.sign = '+';
C.next = &D;        D.next = &F;        F.next = &G;
C.childlist = NULL; D.childlist = NULL; F.childlist = &G;
C.sibling = &D;     D.sibling = NULL;   F.sibling = NULL;

G.sign = '-';       H.sign = '+';       I.sign = '-';
G.next = &H;        H.next = &I;        I.next = NULL;
G.childlist = &H;   H.childlist = &I;   I.childlist = NULL;
G.sibling = NULL;   H.sibling = NULL;   I.sibling = NULL;
```

2.3 Tracing parameters

The tracing operation of Potrace is controlled by a small number of parameters. The parameter structure is defined in potracelib.h as:

```
struct potrace_param_s {
  int turdsize;
  int turnpolicy;
  double alphamax;
  int opticurve;
  double opttolerance;
  potrace_progress_t progress;
};
typedef struct potrace_param_s potrace_param_t;
```

For most practical purposes, the default parameters give excellent results. The function `potrace_param_default()` (see Section 3.3) returns the set of default parameters. Applications must always start from these default parameters before changing any parameters. This will increase backward compatibility in case additional parameters are added in the future.

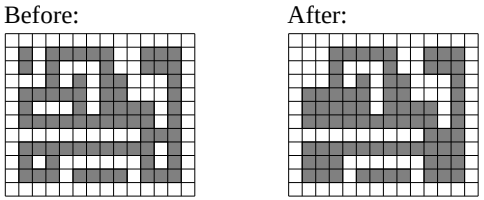


Figure 7: Despeckling with turdsize=3.

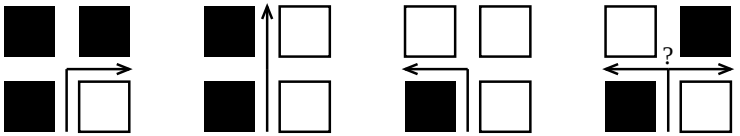


Figure 8: Path decomposition

2.3.1 Turdsize

The `turdsize` parameter can be used to "despeckle" the bitmap to be traced, by removing all curves whose enclosed area is below the given threshold. Figure 7 shows the result of applying turdsize=3 to a bitmap. The current default for the turdsize parameter is 2; its useful range is from 0 to infinity.

2.3.2 Turnpolicy

The turnpolicy parameter determines how to resolve ambiguities during decomposition of bitmaps into paths. The ambiguity arises in the last situation shown in Figure 8. The possible choices for the `turnpolicy` parameter are:

- `POTRACE_TURNPOLICY_BLACK`: prefers to connect black (foreground) components.
- `POTRACE_TURNPOLICY_WHITE`: prefers to connect white (background) components.
- `POTRACE_TURNPOLICY_LEFT`: always take a left turn.
- `POTRACE_TURNPOLICY_RIGHT`: always take a right turn.

$\alpha_{max} = 0.0$ $\alpha_{max} = 0.6$ $\alpha_{max} = 1.0$ $\alpha_{max} = 1.2$ $\alpha_{max} = 1.3$

Figure 9: The alphamax parameter

- `POTRACE_TURNPOLICY_MINORITY`: prefers to connect the color (black or white) that occurs least frequently in a local neighborhood of the current position.
- `POTRACE_TURNPOLICY_MAJORITY`: prefers to connect the color (black or white) that occurs most frequently in a local neighborhood of the current position.
- `POTRACE_TURNPOLICY_RANDOM`: choose pseudo-randomly.

The current default policy is `POTRACE_TURNPOLICY_MINORITY`, which tends to keep visual lines connected.

2.3.3 Alphamax

The `alphamax` parameter is a threshold for the detection of corners. It controls the smoothness of the traced curve, as shown in Figure 9. The current default is 1.0. The useful range of this parameter is from 0.0 (polygon) to 1.3333 (no corners).

2.3.4 Opticurve and opttolerance

The `opticurve` parameter is a boolean flag that controls whether Potrace will attempt to "simplify" the final curve by reducing the number of Bezier curve segments. Opticurve=1 turns on optimization, and opticurve=0 turns it off. The current default is on.

The `opttolerance` parameter defines the amount of error allowed in this simplification. The current default is 0.2. Larger values tend to decrease the number of segments, at the expense of less accuracy. The useful range is from 0 to infinity, although in practice one would hardly choose values greater than 1 or so. For most purposes, the default value is a good tradeoff between space and accuracy.

2.3.5 Progress reporting

Since tracing a large bitmap can be time consuming, Potrace has the option of reporting progress to the calling application. This is typically used in interactive applications to implement a progress bar. Progress reporting is controlled by the `progress` parameter, which is a structure of type `potrace_progress_t`, defined as follows:


```
struct potrace_progress_s {
  void (*callback)(double progress, void *privdata);
  void *data;
  double min, max;
  double epsilon;
};
typedef struct potrace_progress_s potrace_progress_t;
```

If `callback` is not NULL, then progress reporting is enabled. In this case, `callback` is the address of a function to be called for progress reports, and `data` is a pointer to that function's private data. Progress reports take the form of a function call `callback(d, data)`, where `d` is a number representing the amount of relative progress in the range `min`...`max`.

The parameter `epsilon` is a hint that tells Potrace what amount of progress the application considers "too small to report". Whenever convenient, Potrace will feel free to suppress progress reports if the increment since the previous report has been less than `epsilon`. As a special case, if `epsilon` $= 0$, then the maximal number of progress reports are sent. In any case, the application should handle progress reports very efficiently, as there may be a large number of reports.

The defaults are `callback` $=$ NULL, `data` $=$ NULL, `min` $= 0.0$, `max` $= 1.0$, and `epsilon` $= 0$.

2.4 Potrace state

A Potrace state holds the result of a tracing operation. It is defined as follows:

```
struct potrace_state_s {
  int status;
  potrace_path_t *plist;           /* vector data */

  struct potrace_privstate_s *priv; /* private state */
};
typedef struct potrace_state_s potrace_state_t;
```

The fields are as follows:

- The `status` field is either `POTRACE_STATUS_OK`, to indicate that the tracing operation was successful, or `POTRACE_STATUS_INCOMPLETE`, to indicate that it was unsuccessful.
- In the event of success, `plist` points to the representation of the bitonal traced vector image as described in Section 2.2.5. In the event of failure, `plist` points to a data structure whose properties are undefined, except that the Potrace state can still be freed with `potrace_state_free()`.
- The `priv` field is used internally by Potrace, and is not accessible by applications.

3 API functions

There is no global or static state in potracelib; all API functions are reentrant and thread-safe.

3.1 potrace_trace

```
potrace_state_t *potrace_trace(const potrace_param_t *param,
                               const potrace_bitmap_t *bm);
```

Inputs:

- `bm`: a bitmap (see Section 2.1).
- `param`: a set of tracing parameters (see Section 2.3).

Output:

- a Potrace state (see Section 2.4).

This function attempts to trace the given bitmap using the given tracing parameters. In the event of success, it returns a valid Potrace state with the `status` field set to `POTRACE_STATUS_OK`. In the event of failure, it sets `errno` to an error number, and either returns `NULL`, or else it returns an incomplete Potrace state, which by definition has the status field set to `POTRACE_STATUS_INCOMPLETE`. Any Potrace state returned by `potrace_trace()` (whether it is valid or invalid) can be freed using the `potrace_state_free()` function below.

3.2 potrace_state_free

```
void potrace_state_free(potrace_state_t *st);
```

Input:

- `st`: a Potrace state previously returned by `potrace_trace()`.

This function frees the memory and other resources (if any) associated with the Potrace state.

3.3 potrace_param_default

```
potrace_param_t *potrace_param_default();
```

Output:

- a set of tracing parameters (see Section 2.3).

This function returns a fresh set of tracing parameters, initialized to defaults. Applications must always use this function to create an object of type `potrace_param_t`, and they must always start from the default parameters before modifying any parameters. This will help increase backward compatibility when additional parameters are added in the future. The parameter set returned by this function can later be freed by `potrace_param_free()`.

3.4 potrace_param_free()

```
void potrace_param_free(potrace_param_t *p);
```

Input:

- tracing parameters previously returned by `potrace_param_default()`.

This function frees the memory occupied by a set of tracing parameters as returned by `potrace_param_default()`. Only the fields initialized by Potrace are freed, not any fields set by the application itself (such as `progress.data`).

3.5 potrace_version()

This function returns a static human-readable text string identifying this version of potracelib.