Generating 3D Discrete Morse Functions from Point Data

Henry C. King

This code is a companion to the paper [**KKM**] by King, Knudson, and Mramor which explains the theory behind why it works. It takes a simplicial complex K, which is a subcomplex of a compact 3 dimensional manifold[1] M, and an integer valued function h on the vertices of K and extends it to a discrete Morse function on K in the sense of Forman [**F**]. It also cancels pairs of critical j and $j-1$ simplices if they are joined by a single gradient path and the maximum values of h on the two simplices differ by less than a given persistence p.

We do not actually calculate a discrete Morse function on K, but instead determine the crucial information given by a discrete Morse function. This crucial information is the designation of some simplices as critical, and a pairing of the remaining simplices so that each noncritical simplex is either paired to one of its codimension one faces or is paired to a simplex of which it is a codimension one face.

While [**KKM**] assumes that h is injective, this program does not. The program gets around this assumption by, in effect, perturbing h.

We have tested this program with M a triangulation of $B \times S^1$ where B is a fixed triangulation of the Klein bottle and we vary the triangulation of the circle S^1 and take a product triangulation. Then we let K be M minus a disc and put a random function on the vertices of K. On a 933 MHz PowerPC G4 with 640 MB running Mac OS X with 11,960,000 simplices it runs in about 30 seconds with persistence set at 5000 out of a total range of 65536 for the value of h on the vertices. A average run took 19.44 seconds to generate a discrete Morse function, .57 seconds to cancel 0 and 1 simplices, 1.71 seconds to cancel 2 and 3 simplices, and 2.92 seconds to cancel 1 and 2 simplices. An average run with half the number of simplices, 5,980,000 and the same persistence of 5000 took 9.21 seconds to generate a discrete Morse function, .25 seconds to cancel 0 and 1 simplices, .46 seconds to cancel 2 and 3 simplices, and 1.42 seconds to cancel 1 and 2 simplices. All the above times are with the slower_and_finer option, which takes a bit longer to determine critical simplices around a given edge, but does it in such a way that the critical triangles descend as steeply as possible from the edge. For 11,960,000 simplices without the slower_and_finer option it's about .12 seconds faster to generate a discrete Morse function but canceling 1 and 2 simplices is about .39 seconds slower and canceling 2 and 3 simplices is .09 seconds slower. For what it's worth, in the end about 4% of the vertices were critical, about 2.8% of the edges were critical, about 1.7% of the triangles were critical, and about .7% of the tetrahedra were critical.

In practice, the code looks linear in the number of simplices of M. The major portion of the time is spent generating a discrete Morse function. This is linear in the number of simplices if there is an a priori bound on the number of simplices in the Star of a vertex. Without such a bound I expect the time to be proportional to the number of vertices times the average square of the number of simplices in the Star of a vertex. This square comes from local cancelation of critical simplices in the Star of a vertex, since presumably the number of critical simplices in the Star of a vertex is on average proportional to the number of all simplices in the Star of a vertex and the average path length is likewise proportional. I presume that cancelling critical points would tend to be quadratic in the number of simplices since I would expect the number of cancelled critical points to be proportional to the number of simplices and for each pair of cancelled critical points the gradient path between them presumably has length proportional to the number of simplices. In experiments this is not evident except possibly when cancelling 2 and 3 simplices. I only say "possibly" because there was a weird phenomenon where the time to cancel 2 and 3 simplices for the same 11,960,000 simplex complex was either about 1 second or about 1.8 seconds depending on when it was run. Always the lower figure right after compilation and maybe a few more runs, then the higher figure. Go figure.

Bibliography

[**F**] R. Forman, *A User's Guide to Discrete Morse Theory*.

[**KKM**] H. King, K. Knudson, and N. Mramor, *Generating Discrete Morse Functions from Point Data*, preprint.

[1] Actually M does not have to be a manifold as long as M minus its vertices is a manifold and the link of each vertex is connected. But probably in practice M will be S^3 anyway.

January 4, 2005 at 02:54

1. Extract. Portability issue: This code requires **sizeof**(**int**) = **sizeof**(**void** ∗). This could be fixed by duplicating the ordered list routines so that there is a version using pointer keys rather than integer keys, or else always converting any address keys (which are always pointers to simplexes) to the index of the simplex.

This documentation is produced by cweave which uses the following symbols:

Λ is a NULL pointer
∧ is logical and (&&)
∨ is logical or (||)
¬ is logical not (!)

The program has the following structure:

⟨ Header files to include 18 ⟩
⟨ Subroutine prototypes 19 ⟩
⟨ Global variables 2 ⟩
⟨ List functions 84 ⟩
⟨ Simplex functions 102 ⟩
⟨ Debugging output 115 ⟩
⟨ Subroutines for constructing complexes 117 ⟩
⟨ Subroutines finding gradient paths 51 ⟩
⟨ Subroutines canceling a single pair of critical simplices 47 ⟩
⟨ Subroutines canceling many pairs of critical simplices 21 ⟩
⟨ The main routine *Extract* 3 ⟩

2. ⟨ Global variables 2 ⟩ ≡

```
list *crit[4];      /* lists of critical simplices of dimensions 0,1,2, and 3 */
long num_simp[4];   /* number of simplices */
vertex *vertexlist;
edge *elist;
triangle *flist;
tetrahedron *tlist;
```

This code is used in section 1.

3. And now the main routine *Extract*. The parameter *vfirst* is the first vertex of the ambient manifold M containing K, and p is the persistence. The complex M will have information in it which allows us to determine K and the function h on the vertices of K. We cancel 1 and 2 simplices last because this cancelation takes longer, so it is best to reduce the number of 1 and 2 simplices as much as possible beforehand.

```
⟨ The main routine Extract 3 ⟩ ≡
  void Extract(int p)
  {
    long i;
    long t1, t2, t3, t4, t5, t6;
    t1 = clock( );
    for (i = 0; i < 4; i++) list_initialize(crit + i, sizeof(void *));
    ⟨ Find a nonoptimized discrete Morse function on K 4 ⟩;
    t2 = clock( );
    ExtractCancel1(p);      /* Cancel 0 and 1 simplices with persistence less than p */
    t3 = clock( );
    ExtractCancel3(p);      /* Cancel 2 and 3 simplices with persistence less than p */
    t4 = clock( );
    ExtractCancel2(p);      /* Cancel 1 and 2 simplices with persistence less than p */
    t5 = clock( );
#ifdef verbose
    printf("\nExtract␣Raw␣=␣%d,␣Cancel1␣=␣%d,␣Cancel2␣=␣%d,␣Cancel3␣=␣%d\n", t2 − t1, t3 − t2,
        t5 − t4, t4 − t3);
#endif
  }
```

This code is used in section 1.

4. We called this section of code *ExtractRaw* in [**KKM**]. For each vertex v of K, it finds an optimal discrete Morse function on the lower Star of v. (The lower Star of v is all simplices for which v is the vertex of maximal value.) It then takes one critical edge *beste* in the lower Star of v, makes it noncritical, and pairs it with v. It cancels all pairs of critical simplices it can in the lower Star so that in the end you get as few of them as possible. There are other algorithms which find optimal discrete Morse functions and are presumably faster if the lower Star has a great many simplices, but this is simple and works fine for reasonably sized links. It also chooses critical simplices based on the given function values on the vertices (especially if *slower_and_finer* is set), which other algorithms we know of do not. Note that if the lower Star of v is empty, you make the vertex v critical, since v is a local minimum.

When we say "steepest" below we are assuming all edges have equal length. If we wish to account for edges of unequal length and pick steepest edges, triangles, etc., it would be necessary to modify the algorithm.

```
⟨ Find a nonoptimized discrete Morse function on K 4 ⟩ ≡
  {
    edge *beste;      /* this will be the steepest edge from v */
    vertex *v;
    int is_lower_Star_empty;
    long vid;
    list *lcrit2;      /* list of critical 2 simplices in lower Star of v */
    list *edges_todo;      /* list of unprocessed edges containing v */
    list_initialize(&lcrit2, sizeof(triangle *));
    list_initialize(&edges_todo, sizeof(edge *));
    for (vid = 0; vid < num_simp[0]; vid++) {
      v = id2vertex(vid);
      if (¬is_in_K(v)) continue;
      beste = Λ;
      ⟨ extract the lower Star of v 5 ⟩;
      if (is_lower_Star_empty) make_critical(v);
      else {
        pair01(v, beste);      /* pair v with the steepest edge down */
        LocalCancel(v, lcrit2);      /* cancel critical simplices in the lower Star of v */
      }
    }
    list_abandon(&lcrit2);
    list_abandon(&edges_todo);
  }
```

This code is used in section 3.

5. Now we determine the lower Star of v and find a discrete Morse function on it. We do this by taking each edge e in the lower Star of v and putting a discrete Morse function on its lower Star. The lower Star of e is all simplices σ of K containing e so that the maximal and next–to–maximal vertices of σ are in e.

```
#define mark_edge_done(v,e)  ((e)→type |= (((v) ≡ get_vertex(e,0)) ? #200 : #400))
#define edge_marked_done(v,e)  ((e)→type & (((v) ≡ get_vertex(e,0)) ? #200 : #400))
⟨ extract the lower Star of v 5 ⟩ ≡
  {
    edge *e;
    vertex *w;
    int val, bestval, flag;
    val = value(v);
    list_clear(edges_todo);
    plist_push(edges_todo, v→links[0]);      /* push an edge containing v */
    is_lower_Star_empty = 1;      /* flag to test if lower Star(v) is empty */
    while (¬list_is_empty(edges_todo)) {
      ⟨ pop the next item e from edges_todo which we haven't processed yet 6 ⟩
      ⟨ set flag and reset is_lower_Star_empty if e is in lower Star(v) 7 ⟩
      ⟨ add adjacent edges of e to edges_todo and Extract lower Star(e) if flag is set 8 ⟩
    }
  }
```

This code is cited in section 100.

This code is used in section 4.

6.
```
⟨ pop the next item e from edges_todo which we haven't processed yet 6 ⟩ ≡
  e = plist_pop(edges_todo);
  if (edge_marked_done(v,e)) continue;
  mark_edge_done(v,e);
```

This code is used in section 5.

7.
```
⟨ set flag and reset is_lower_Star_empty if e is in lower Star(v) 7 ⟩ ≡
  flag = 0;      /* default */
  if (is_in_K(e)) {
    w = smallest_vertex_in_edge(e);
    if (w ≠ v)      /* is e in lower Star(v)? */
    {
      set_value(e, val);      /* fill in max value on e */
      is_lower_Star_empty = 0;      /* indicate that lower Star(v) is not empty */
      flag = 1;
    }
  }
```

This code is used in section 5.

8. This section of code puts a discrete Morse function on the lower Star of the edge e. Since this essentially means putting a discrete Morse function on a subcomplex of a circle, we can use a quick algorithm to do this. The drawback is that the choice of critical simplices has little to do with the values of the given function on the vertices, beyond the fact that this is used to determine the lower Star of e. A slightly slower algorithm is used if you define *slower_and_finer*. It chooses critical triangles so that the value at the vertex opposite e is a minimum in its connected component of lower Star(e). It takes a bit longer here, but experiments indicate it tends to speed up canceling 1 and 2 simplices, presumably by making descending discs descend faster.

The *state* variable has the following significance:
0 means a triangle in lower Star(e) has not yet been encountered.
1 means all triangles and tetrahedra so far are in lower Star(e)
2 means the last triangle was not in lower Star(e), but some previous triangle was.
3 means the last triangle was in lower Star(e), but some previous triangles or tetrahedra weren't.

```
#define slower_and_finer 1
⟨ add adjacent edges of e to edges_todo and Extract lower Star(e) if flag is set 8 ⟩ ≡
  {
    triangle *s, *bests, *sp;
    tetrahedron *t;
    edge *ep;
    int state = 0;
    int bestsval;
    int first_in_lower_Star;
    vertex *min_vertex;      /* the vertex of s with minimum value */
#ifdef slower_and_finer
    triangle *bests_in_run, *first_bests_in_run;
    tetrahedron *bestt_in_run, *first_bestt_in_run;
    int bestsval_in_run, first_bestsval;
#else
    vertex *first_min_vertex;      /* the vertex of sp with minimum value */
#endif
    s = sp = e→links[2];      /* a triangle containing e */
    t = coface(s, 0);      /* a tetrahedron containing s */
    do {
      ep = other_edge(v, e, s);
      if (¬edge_marked_done(v, ep)) plist_push(edges_todo, ep);
#ifdef slower_and_finer
      if (flag) ⟨ xextract s and t 14 ⟩
#else
      if (flag) ⟨ extract s and t 9 ⟩
#endif
      t = other_coface(s, t);
      s = other_face(e, s, t);
    } while (s ≠ sp);
#ifdef slower_and_finer
    if (flag) ⟨ xlast extract of s and t 17 ⟩
#else
    if (flag) ⟨ last extract of s and t 12 ⟩
#endif
  }
```

This code is used in section 5.

9. ⟨ extract s and t 9 ⟩ ≡

```
{
  int in_lower_Star_of_e;
  ⟨ see if s is in lower Star(e) and set min_vertex and in_lower_Star_of_e accordingly 10 ⟩;
  switch (state) {
  case 0:
    if (in_lower_Star_of_e) {
      first_in_lower_Star = (s ≡ sp ∧ is_in_K(t));
      if (first_in_lower_Star) {
        state = 1;
        bests = Λ;
        first_min_vertex = min_vertex;
      }
      else {
        state = 3;
        bests = s;
        bestsval = value(min_vertex);
      }
    }
    break;
  case 1: case 3:
    if (¬in_lower_Star_of_e) {
      state = 2;
      break;
    }
    else if (is_in_K(t)) {
      set_value(t, val);      /* max value on t must be at vertex v */
      pair23(s, t);
      break;
    }      /* else pass through to case 2 */
  case 2:
    if (in_lower_Star_of_e) {
      ⟨ make s critical xor make bests critical and replace bests 11 ⟩
      state = 3;
    }
    break;
  }
}
```

This code is used in section 8.

10. ⟨ see if s is in lower Star(e) and set min_vertex and in_lower_Star_of_e accordingly 10 ⟩ ≡

```
in_lower_Star_of_e = 0;      /* default */
if (is_in_K(s)) {
  vertex *vlist[3];
  get_triangle_vertices(s, vlist);
  min_vertex = vlist[smallest_vertex(vlist, 3)];
  in_lower_Star_of_e = (vertex_in_edge(min_vertex, e) < 0);
  if (in_lower_Star_of_e) set_value(s, val);
}
```

This code is used in sections 9 and 14.

11. ⟨ make s critical xor make *bests* critical and replace *bests* 11 ⟩ ≡

```
if (bests ≡ Λ) {
  bests = s;
  bestsval = value(min_vertex);
}
else if (value(min_vertex) < bestval) {
  make_critical(bests);
  plist_push(lcrit2, bests);
  bests = s;
  bestsval = value(min_vertex);
}
else {
  make_critical(s);
  plist_push(lcrit2, s);      /* remember all critical triangles for LocalCancel */
}
```

This code is used in sections 9 and 12.

12. ⟨ last extract of s and t 12 ⟩ ≡

```
switch (state) {
case 0: ⟨ make e critical unless it is the best candidate so far to pair with v 13 ⟩
  break;
case 1:
  make_critical(t);
  set_value(t, val);
  pair12(e, s);
  break;
case 3:
  if (first_in_lower_Star) {
    pair23(s, t);
    set_value(t, val);
  }
  pair12(e, bests);
  break;
case 2:
  if (first_in_lower_Star) {
    min_vertex = first_min_vertex;
    ⟨ make s critical xor make bests critical and replace bests 11 ⟩
  }
  pair12(e, bests);
  break;
}
```

This code is used in section 8.

13. ⟨ make e critical unless it is the best candidate so far to pair with v 13 ⟩ ≡

```
if (beste ≡ Λ)      /* is it the first? */
{
  beste = e;
  bestval = value(w);
}
else if (value(w) < bestval) {
  make_critical(beste);
  beste = e;
  bestval = value(w);
}
else make_critical(e);
```

This code is used in sections 12 and 17.

14. ⟨ xextract s and t 14 ⟩ ≡

```
{
  int in_lower_Star_of_e;
  ⟨ see if s is in lower Star(e) and set min_vertex and in_lower_Star_of_e accordingly 10 ⟩;
  switch (state) {
  case 0:
    if (in_lower_Star_of_e) {
      first_in_lower_Star = (s ≡ sp ∧ is_in_K(t));
      if (first_in_lower_Star) state = 1;
      else state = 3;
      bests = Λ;
      bests_in_run = s;
      bestt_in_run = t;
      bestsval_in_run = value(min_vertex);
    }
    break;
  case 1: case 3:
    if (¬in_lower_Star_of_e ∨ ¬is_in_K(t)) {
      ⟨ make bests_in_run or bests critical and update bests 15 ⟩
    }
    if (¬in_lower_Star_of_e) {
      state = 2;
      break;
    }
    else if (is_in_K(t)) {
      set_value(t, val);      /* max value on t must be at vertex v */
      if (value(min_vertex) < bestsval_in_run) ⟨ replace bests_in_run by s 16 ⟩
      else pair23(s, t);
      break;
    }     /* else pass through to case 2 */
  case 2:
    if (in_lower_Star_of_e) {
      bests_in_run = s;
      bestt_in_run = t;
      bestsval_in_run = value(min_vertex);
      state = 3;
    }
    break;
  }
}
```

This code is used in section 8.

15. ⟨ make *bests_in_run* or *bests* critical and update *bests* 15 ⟩ ≡

```
if (bests ≡ Λ) {
  if (state ≡ 1) {
    first_bests_in_run = bests_in_run;
    first_bestt_in_run = bestt_in_run;
    first_bestsval = bestsval_in_run;
  }
  else {
    bests = bests_in_run;
    bestsval = bestsval_in_run;
  }
}
else if (bestsval_in_run < bestval) {
  make_critical(bests);
  plist_push(lcrit2, bests);
  bests = bests_in_run;
  bestsval = bestsval_in_run;
}
else {
  make_critical(bests_in_run);
  plist_push(lcrit2, bests_in_run);      /* remember all critical triangles for LocalCancel */
}
```

This code is used in sections 14 and 17.

16. ⟨ replace *bests_in_run* by *s* 16 ⟩ ≡

```
{
  triangle *ss;
  tetrahedron *tt;
  do {
    tt = other_coface(bests_in_run, bestt_in_run);
    ss = r32(tt);
    pair23(bests_in_run, tt);
    bests_in_run = ss;
    bestt_in_run = tt;
  } while (bestt_in_run ≠ t);
  bests_in_run = s;
  bestsval_in_run = value(min_vertex);
}
```

This code is used in sections 14 and 17.

17. ⟨ xlast extract of s and t 17 ⟩ ≡

```
{
  triangle *sss;
  switch (state) {
  case 0: ⟨ make e critical unless it is the best candidate so far to pair with v 13 ⟩
    break;
  case 1:
    make_critical(t);
    set_value(t, val);
    pair12(e, bests_in_run);
    break;
  case 3:
    if (first_in_lower_Star) {
      set_value(t, val);
      if (first_bestsval > bestsval_in_run) {
        sss = bests_in_run;
        bestt_in_run = other_coface(first_bests_in_run, first_bestt_in_run);
        bests_in_run = first_bests_in_run;
      }
      else sss = first_bests_in_run;
      ⟨ replace bests_in_run by s 16 ⟩
      bests_in_run = sss;
    }
    ⟨ make bests_in_run or bests critical and update bests 15 ⟩
    pair12(e, bests);
    break;
  case 2:
    if (first_in_lower_Star) {
      bests_in_run = first_bests_in_run;
      bestsval_in_run = first_bestsval;
      ⟨ make bests_in_run or bests critical and update bests 15 ⟩
    }
    pair12(e, bests);
    break;
  }
}
```

This code is used in section 8.

18. A few helpful functions.

```
#define min(x, y) (((x) > (y)) ? y : x)
#define max(x, y) (((x) > (y)) ? x : y)
#define in_MorseExtract 1
```

⟨ Header files to include 18 ⟩ ≡

```
#include <stdio.h>
#include "MorseExtract.h"
```

This code is used in section 1.

19. ⟨ Subroutine prototypes 19 ⟩ ≡

```
void abort_message(char *s);
```

This code is used in section 1.

20. Cancel Routines. Sometimes a critical $m-1$ simplex and a critical m simplex can be cancelled. You can do this if there is exactly one gradient path between them. A gradient path is a sequence of simplices $\sigma_i, \sigma_{i+1}, \sigma_{i+2}, \ldots, \sigma_k$ so that for j even, σ_j is an m simplex, and σ_{j-1} is paired with σ_j, and σ_{j+1} is a codimension one face of σ_j. To cancel a pair of critical simplices σ and τ, you take the unique gradient path $\sigma = \sigma_0, \ldots, \sigma_{2k-1} = \tau$, and you pair each σ_{2i} with σ_{2i+1} (rather than with σ_{2i-1} as was done formerly).

21. Local Cancellation. *LocalCancel* cancels simplices in the lower Star of a vertex. After doing it, you can show that you are left with a minimal number of critical simplices. For example, at a local maximum you would end up with only one critical simplex, a 3 simplex. Likewise at PL approximations of traditional smooth Morse saddles of index i you would end up with a single critical simplex, an i simplex.

```
⟨ Subroutines canceling many pairs of critical simplices 21 ⟩ ≡
  void LocalCancel(vertex *v, list *lcrit2)
  {
    triangle *t, *s;
    tetrahedron *te[2];
    edge *e[2], *ep[2];
    int j, i, in;
    while (¬list_is_empty(lcrit2))      /* run through the list of new critical 2 simplices */
    {
      t = plist_pop(lcrit2);
      ⟨ find the edges e[0] and e[1] of t containing v 22 ⟩;
      ⟨ find the ends ep[i] of the gradient paths starting at e[i] 23 ⟩;
      if (ep[0] ≠ ep[1])      /* can we cancel? */
      {
        ⟨ Cancel the t with the best of the two edges 25 ⟩;
      }
      else {      /* We can't cancel with an edge, so see if we can cancel with a 3 simplex */
        for (i = 0; i < 2; i++) {
          ⟨ find the 3 simplex te[i] connected to coface(t, i) by a gradient path 24 ⟩;
        }
        if (te[0] ≠ te[1]) {
          ⟨ Cancel the t with the best of the two tetrahedra 26 ⟩;
        }
      }
    }
  }
```

See also sections 27, 35, and 43.

This code is used in section 1.

```
22.  ⟨ find the edges e[0] and e[1] of t containing v 22 ⟩ ≡
  for (i = 0, j = 0; j < 3; j++) {
    if (vertex_in_edge(v, get_edge(t, j)) ≥ 0) {
      e[i++] = get_edge(t, j);
    }
  }
```

This code is used in section 21.

23. ⟨ find the ends $ep[i]$ of the gradient paths starting at $e[i]$ 23 ⟩ ≡

```
for (i = 0; i < 2; i++) {
  ep[i] = e[i];
  while (¬is_critical(ep[i])) {
    if (is_paired_down(ep[i])) {
      ep[i] = Λ;      /* runs into B_1 so no cancelation possible */
      break;
    }
    s = r12(ep[i]);      /* get paired triangle */
    ep[i] = other_edge(v, ep[i], s);
  }
}
```

This code is used in section 21.

24. ⟨ find the 3 simplex $te[i]$ connected to $coface(t, i)$ by a gradient path 24 ⟩ ≡

```
te[i] = coface(t, i);
s = t;
while (is_in_K(te[i]))      /* while we are in K */
{
  in = is_in_lower_Star(te[i], v);
  if (¬in) break;      /* see if exit lower Star */
  if (is_critical(te[i])) break;      /* see if reached critical */
  s = r32(te[i]);
  te[i] = other_coface(s, te[i]);
}
if (¬is_in_K(te[i]) ∨ ¬in) te[i] = Λ;
```

This code is used in section 21.

25. ⟨ Cancel the t with the best of the two edges 25 ⟩ ≡

```
if (ep[0] ≡ Λ) LocalCancel12(v, ep[1], e[1], t);
else if (ep[1] ≡ Λ) LocalCancel12(v, ep[0], e[0], t);
else if (min_value(ep[0], 1) > min_value(ep[1], 1)) LocalCancel12(v, ep[0], e[0], t);
else LocalCancel12(v, ep[1], e[1], t);
```

This code is used in section 21.

26. ⟨ Cancel the t with the best of the two tetrahedra 26 ⟩ ≡

```
if (te[0] ≡ Λ) Cancel23(t, 1, te[1]);
else if (te[1] ≡ Λ) Cancel23(t, 0, te[0]);
else if (min_value(te[1], 3) > min_value(te[0], 3)) Cancel23(t, 0, te[0]);
else Cancel23(t, 1, te[1]);
```

This code is used in section 21.

27. Canceling vertices and edges. Find all critical edges and vertices connected by a single gradient path. Cancel the pair with smallest difference in value, as long as it's less than p. Keep on doing this until you can't do any more.

```
⟨ Subroutines canceling many pairs of critical simplices 21 ⟩ +≡
  void ExtractCancel1(int p){
    olist *c0;    /* list of vertices to cancel with */
    olist *c1;    /* list of edges which can be cancelled with vertices */
    struct ccrit1 {
      edge *s;
      long c[2];
    } s;
    int i;
    vertex *v[2];
    int thisk;
    olist_initialize(&c0, sizeof(long));
    olist_initialize(&c1, sizeof(struct ccrit1));
    ⟨ Find all cancelable pairs and put them on lists c0 and c1 28 ⟩;
    while (¬olist_is_empty(c1)) {
      thisk = olist_min(c1, &s);    /* put best candidate in s */
      ⟨ let v[i] be the two vertices with which s can cancel 30 ⟩
      i = ⟨ index of best vertex of s.s to cancel 34 ⟩;
      if (is_critical(v[i]) ∧ thisk ≡ value(s.s) − value(v[i])) ⟨ cancel with the vertex 31 ⟩
      else ⟨ see if we can still cancel with some other vertex and replace s on c1 32 ⟩
    }
    olist_abandon(&c0);
    olist_abandon(&c1);
  }
```

28.

```
⟨ Find all cancelable pairs and put them on lists c0 and c1 28 ⟩ ≡
  {
    edge *e;
    list_read_init(crit[1]);
    while ((e = (edge *) plist_read(crit[1])) ≠ Λ)    /* run through all critical edges */
    {
      if (¬is_critical(e)) list_read_delete(crit[1]);
      else {    /* find the two gradient paths descending from e */
        v[0] = FindGrad01(get_vertex(e, 0), value(e) − p);
        v[1] = FindGrad01(get_vertex(e, 1), value(e) − p);
        if (v[0] ≠ v[1]) {
          ⟨ put e on the list c1 and put v[i] on the list c0 29 ⟩;
        }
      }
    }
  }
```

This code is used in section 27.

29. ⟨ put e on the list $c1$ and put $v[i]$ on the list $c0$ 29 ⟩ ≡

```
{
  long link;
  link = -1;
  s.s = e;
  s.c[0] = (v[0] ≡ Λ) ? -1 : olist_find_add(c0, (long) v[0], &link, Λ);
  s.c[1] = (v[1] ≡ Λ) ? -1 : olist_find_add(c0, (long) v[1], &link, Λ);
  i = ⟨ index of best vertex of s.s to cancel 34 ⟩;
  olist_add(c1, value(e) - value(v[i]), &s);
}
```

This code is used in section 28.

30. ⟨ let $v[i]$ be the two vertices with which s can cancel 30 ⟩ ≡

```
v[0] = (s.c[0] < 0) ? Λ : (vertex *) olist_get_key(c0, s.c[0]);
v[1] = (s.c[1] < 0) ? Λ : (vertex *) olist_get_key(c0, s.c[1]);
```

This code is used in section 27.

31. ⟨ cancel with the vertex 31 ⟩ ≡

```
{
  Cancel01(v[i], i, s.s);
  *((long *) olist_entry(c0, s.c[i])) = s.c[1 - i];
    /* now any gradient path to v[i] goes to other vertex v[1 - i] */
}
```

This code is used in section 27.

32. ⟨ see if we can still cancel with some other vertex and replace s on $c1$ 32 ⟩ ≡

```
{
  int m;
  for (i = 0; i < 2; i++) {
    if (s.c[i] < 0) continue;
    ⟨ update s.c[i] to point to a critical vertex 33 ⟩
  }
  if (s.c[1] ≠ s.c[0]) {
    i = ⟨ index of best vertex of s.s to cancel 34 ⟩;
    m = value(s.s) - value(v[i]);
    if (m < p) olist_add(c1, m, &s);
  }
}
```

This code is used in section 27.

33. ⟨ update $s.c[i]$ to point to a critical vertex 33 ⟩ ≡

```
{
  long a;
  for (a = s.c[i]; a ≥ 0; ) {
    v[i] = (vertex *) olist_get_key(c0, a);
    if (is_critical(v[i])) break;
    a = *((long *) olist_entry(c0, a));
  }
  *((long *) olist_entry(c0, s.c[i])) = a;
  if (value(v[i]) ≤ value(s.s) − p) {
    v[i] = Λ;
    s.c[i] = −1;
  }
  else s.c[i] = a;
}
```

This code is used in section 32.

34. ⟨ index of best vertex of $s.s$ to cancel 34 ⟩ ≡

```
(v[0] ≡ Λ) ? 1 :
(v[1] ≡ Λ) ? 0 :
(value(v[0]) > value(v[1])) ? 0 :
(value(v[0]) < value(v[1])) ? 1 :
is_critical(v[0]) ? 1 : 0
```

This code is used in sections 27, 29, and 32.

35. Canceling triangles and tetrahedra. Find all critical triangles and tetrahedra connected by a single gradient path. Cancel the pair with smallest difference in value, as long as it's less than p. Keep on doing this until you can't do any more. The code is essentially the same as that used to cancel edges and vertices.

```
⟨ Subroutines canceling many pairs of critical simplices 21 ⟩ +≡
  void ExtractCancel3(int p){
      olist *c3;      /* list of tetrahedra to cancel */
      olist *c2;      /* list of triangles which cancel with tetrahedra */
      struct ccrit2 {
        triangle *s;
        long c[2];
      } r;
      int i;
      tetrahedron *t[2];
      int thisk;
      olist_initialize(&c2, sizeof(struct ccrit2));
      olist_initialize(&c3, sizeof(long));
      ⟨ Find all cancelable pairs and put them on lists c2 and c3 36 ⟩;
      while (¬olist_is_empty(c2)) {
        thisk = olist_min(c2, &r);
        ⟨ let t[i] be the two tetrahedra with which we can cancel 38 ⟩
        i = ⟨ index of best coface of r.s to cancel 42 ⟩;
        if (is_critical(t[i]) ∧ thisk ≡ value(t[i]) − value(r.s)) ⟨ cancel with the tetrahedron 39 ⟩
        else ⟨ see if we can still cancel with some other tetrahedron and replace on c2 40 ⟩
      }
      olist_abandon(&c2);
      olist_abandon(&c3);
    }
```

36. ⟨ Find all cancelable pairs and put them on lists $c2$ and $c3$ 36 ⟩ ≡

```
  {
      triangle *s;
      list_read_init(crit[2]);
      while ((s = (triangle *) plist_read(crit[2])) ≠ Λ)      /* go through all critical triangles */
      {
        if (¬is_critical(s)) list_read_delete(crit[2]);
        else {      /* find the beginnings of the two gradient paths ending at s */
          t[0] = FindGrad23(coface(s, 0), value(s) + p);
          t[1] = FindGrad23(coface(s, 1), value(s) + p);
          if (t[0] ≠ t[1]) ⟨ put s on c2 and t[0] and t[1] on c3 37 ⟩
        }
      }
  }
```

This code is used in section 35.

37. ⟨ put s on $c2$ and $t[0]$ and $t[1]$ on $c3$ 37 ⟩ ≡

```
{
  long link;
  link = −1;
  r.s = s;
  r.c[0] = (t[0] ≡ Λ) ? −1 : olist_find_add(c3, (long) t[0], &link, Λ);
  r.c[1] = (t[1] ≡ Λ) ? −1 : olist_find_add(c3, (long) t[1], &link, Λ);
  i = ⟨ index of best coface of r.s to cancel 42 ⟩;
  olist_add(c2, value(t[i]) − value(s), &r);
}
```

This code is used in section 36.

38. ⟨ let $t[i]$ be the two tetrahedra with which we can cancel 38 ⟩ ≡

```
t[0] = (r.c[0] < 0) ? Λ : (tetrahedron *) olist_get_key(c3, r.c[0]);
t[1] = (r.c[1] < 0) ? Λ : (tetrahedron *) olist_get_key(c3, r.c[1]);
```

This code is used in section 35.

39. ⟨ cancel with the tetrahedron 39 ⟩ ≡

```
{
  Cancel23(r.s, i, t[i]);
  *((long *) olist_entry(c3, r.c[i])) = r.c[1 − i];
}
```

This code is used in section 35.

40. ⟨ see if we can still cancel with some other tetrahedron and replace on $c2$ 40 ⟩ ≡

```
{
  int m;
  for (i = 0; i < 2; i++) {
    if (r.c[i] < 0) continue;
    ⟨ update r.c[i] to point to a critical tetrahedron 41 ⟩
  }
  if (r.c[1] ≠ r.c[0]) {
    i = ⟨ index of best coface of r.s to cancel 42 ⟩;
    m = value(t[i]) − value(r.s);
    if (m < p) olist_add(c2, m, &r);
  }
}
```

This code is used in section 35.

41. ⟨ update $r.c[i]$ to point to a critical tetrahedron 41 ⟩ ≡

```
{
  long a;
  for (a = r.c[i]; a ≥ 0; ) {
    t[i] = (tetrahedron *) olist_get_key(c3, a);
    if (is_critical(t[i])) break;
    a = *((long *) olist_entry(c3, a));
  }
  *((long *) olist_entry(c3, r.c[i])) = a;
  if (value(t[i]) ≥ value(r.s) + p) {
    t[i] = Λ;
    r.c[i] = −1;
  }
  else r.c[i] = a;
}
```

This code is used in section 40.

42. ⟨ index of best coface of $r.s$ to cancel 42 ⟩ ≡

```
(t[0] ≡ Λ) ? 1 :
(t[1] ≡ Λ) ? 0 :
(value(t[0]) > value(t[1])) ? 1 :
(value(t[0]) < value(t[1])) ? 0 :
is_critical(t[0]) ? 1 : 0
```

This code is used in sections 35, 37, and 40.

43. Canceling edges and triangles. Find and cancel pairs of 1 and 2 simplices whose values differ by less than p. This version speeds things up by not always canceling the pair with least persistence. In particular, after one cancelation it may become possible for some other pair of critical simplices to cancel which could not have been cancelled before. If this happens, this routine could be unaware until it has cancelled all the pairs it knows about already. Checking for this situation would slow down this routine dramatically. In fact such occurrences appear to be quite rare. For random functions on K with millions of simplices and persistence 1/13 of the range of the values of vertices, it tends to happen at most once or twice.

```
⟨ Subroutines canceling many pairs of critical simplices 21 ⟩ +≡
  void ExtractCancel2(int p)
  {
    list *changed;      /* triangles whose pairing was changed by Cancel12 */
    list *grad_path;    /* edges in gradient path of a canceling pair */
    olist *goodpairs;   /* cancelable critical triangles */
    int lastp;
    list_initialize(&changed, sizeof(triangle *));
    list_initialize(&grad_path, sizeof(edge *));
    olist_initialize(&goodpairs, sizeof(triangle *));
    lastp = 0;
    do {
      ⟨ fill list goodpairs with possible cancels 44 ⟩
      if (olist_is_empty(goodpairs)) break;
      ⟨ cancel all pairs on goodpairs 45 ⟩
    } while (1);
    list_abandon(&changed);
    list_abandon(&grad_path);
    olist_abandon(&goodpairs);
  }
```

44.

```
⟨ fill list goodpairs with possible cancels 44 ⟩ ≡
  {
    triangle *t;
    edge *e;
    list_read_init(crit[2]);
    while ((t = (triangle *) plist_read(crit[2])) ≠ Λ)      /* go through all critical triangles */
    {
      if (¬is_critical(t)) list_read_delete(crit[2]);
      else {
        e = FindGradPaths12(t, p, Λ, 0);
        if (e ≠ Λ)      /* if there is a gradient path from t to e */
          olist_add(goodpairs, value(t) − value(e), &t);
      }
    }
  }
```

This code is used in section 43.

45. ⟨ cancel all pairs on *goodpairs* 45 ⟩ ≡

```
{
  triangle *t;
  edge *e;
  int bp, thisp;
  while (¬olist_is_empty(goodpairs)) {
    bp = olist_min(goodpairs, &t);
    e = FindGradPaths12(t, p, grad_path, 0);
    if (e ≡ Λ) continue;      /* no gradient paths from t found */
    thisp = value(t) − value(e);
    if (thisp ≤ bp) {
      list_clear(changed);
      Cancel12(e, t, grad_path, changed);      /* cancel e and t */
      ⟨ fix up split-rejoin paths 46 ⟩;
#ifdef verbose
      if (thisp < lastp) printf("\n persistence out of order: %d > %d\n", lastp, thisp);
#endif
      lastp = thisp;
    }
    else olist_add(goodpairs, thisp, &t);
  }
}
```

This code is used in section 43.

46. After canceling a pair of one and two simplices, there may be new pairs of gradient paths which differ only on the boundary of a single tetrahedron. This code modifies the simplex pairings to eliminate this happening. It is not clear whether it is worth doing this. Experiments show that this tends to allow cancellation of a few more pairs of critical one and two simplices, but not many. Experiments also show that very little time is spent in this code. So I have left it in. We have not yet attempted to prove that this fixing up process always terminates, so to be careful I have put a limit of 10000 iterations which has so far never been exceeded.

```
⟨ fix up split-rejoin paths 46 ⟩ ≡
  {
    int i, j, k;
    tetrahedron *te;
    k = 0;
    while (¬list_is_empty(changed) ∧ k < 10000) {
      t = plist_pop(changed);
      if (t ≡ Λ) continue;
      for (i = 0; i < 2; i++) {
        te = coface(t, i);
        if (is_in_K(te) ∧ ¬is_critical(te)) {
          j = splitrejoin(te);
          if (j ≥ 0) {
            plist_push(changed, unsplitrejoin(te, j));
            k++;
          }
        }
      }
    }
    if (¬list_is_empty(changed)) printf("May be infinite loop in split-rejoin");
  }
```

This code is used in section 45.

47. Canceling pairs of Critical Simplices. The following routine cancels a critical 1 simplex σ containing v with a critical 2 simplex τ containing v. It speeds up the ordinary Cancel12 by assuming that the unique gradient path from τ to σ has all of its simplices containing v. The gradient path from τ to σ goes through the face *sigmap* of τ.

```
⟨ Subroutines canceling a single pair of critical simplices 47 ⟩ ≡
  void LocalCancel12(vertex *v, edge *sigma, edge *sigmap, triangle *tau)
  {
    edge *u;
    triangle *w, *wp;
    u = sigmap;
    w = tau;
    unmake_critical(tau);
    while (¬is_critical(u))      /* while u is noncritical */
    {
      wp = r12(u);
      pair12(u, w);
      w = wp;
      u = other_edge(v, u, w);
    }
    unmake_critical(sigma);
    pair12(sigma, w);
  }
```

See also sections 48, 49, and 50.

This code is used in section 1.

48. The following routine cancels a critical 2 simplex σ with a critical 3 simplex τ. The n is such that the gradient path from τ to σ goes through *coface*(*sigma*, n). It does not delete σ and τ from *crit*[2] and *crit*[3].

```
⟨ Subroutines canceling a single pair of critical simplices 47 ⟩ +≡
  void Cancel23(triangle *sigma, int n, tetrahedron *tau)
  {
    triangle *s, *sp;
    tetrahedron *t;
    s = sigma;
    t = coface(s, n);
    unmake_critical(sigma);
    while (¬is_critical(t))      /* while t is not critical */
    {
      sp = r32(t);
      pair23(s, t);
      s = sp;
      t = other_coface(s, t);
    }
    unmake_critical(tau);
    pair23(s, tau);
  }
```

49. The following routine cancels a critical vertex v with a critical edge κ. It does not delete v and κ from $crit[0]$ and $crit[1]$ however. The n is such that the gradient path from v to κ goes through $get\_vertex(kappa, n)$.

```
⟨ Subroutines canceling a single pair of critical simplices 47 ⟩ +≡
  void Cancel01(vertex *v, int n, edge *kappa)
  {
    vertex *u;
    edge *e, *ep;
    u = get_vertex(kappa, n);
    e = kappa;
    unmake_critical(kappa);
    while (¬is_critical(u))      /* while non critical */
    {
      ep = r01(u);
      pair01(u, e);
      e = ep;
      u = other_vertex_in_edge(u, e);
    }
    unmake_critical(v);
    pair01(v, e);
  }
```

50. Cancel a critical 1 simplex σ with a critical 2 simplex κ, using the list of edges in the gradient path $grad\_path$. Push any 2 simplices with changed pairings to the stack $changed$. Note σ and κ are not deleted from $crit[1]$ and $crit[2]$.

```
⟨ Subroutines canceling a single pair of critical simplices 47 ⟩ +≡
  void Cancel12(edge *sigma, triangle *kappa, list *grad_path, list *changed)
  {
    triangle *t, *nextt;
    edge *e;
    unmake_critical(kappa);
    unmake_critical(sigma);
    t = kappa;
    do {
      e = plist_pop(grad_path);
      if (e ≡ sigma) break;
      nextt = r12(e);
      pair12(e, t);
      plist_push(changed, t);
      t = nextt;
    } while (1);
    pair12(e, t);
    plist_push(changed, t);
  }
```

51. Finding Gradient Paths. Find the end of a gradient path starting at $u \in K_0$. Return the critical vertex v at the end of the path, or return Λ if $h(v) \leq m$.

```
⟨ Subroutines finding gradient paths 51 ⟩ ≡
  vertex *FindGrad01(vertex *u, int m)
  {
    vertex *v;
    edge *e;
    v = u;
    while (¬is_critical(v) ∧ value(v) > m) {
      e = r01(v);
      v = other_vertex_in_edge(v, e);
    }
    if (value(v) ≤ m) return Λ;
    return v;
  }
```

See also sections 52, 53, 59, 60, 62, and 72.

This code is used in section 1.

52. Find the start of a gradient path passing through a tetrahedron τ. Return the critical 3 simplex t at the start of the path, or return Λ if the value of t is $\geq m$ or if the path starts at a boundary 2 simplex. Mark simplices deadend if we know the only gradient paths going to them start at a boundary 2 simplex. This is a lazy way of shortening future gradient path searches.

```
⟨ Subroutines finding gradient paths 51 ⟩ +≡
  tetrahedron *FindGrad23(tetrahedron *tau, int m)
  {
    tetrahedron *t;
    triangle *s;
    if (tau ≡ Λ) return Λ;      /* obsolete? */
    t = tau;
    while (is_in_K(t) ∧ value(t) < m) {
      if (is_critical(t)) return t;      /* reached critical t */
      s = r32(t);
      if (is_deadend(s)) {
        make_deadend(t);
        return Λ;
      }
      t = other_coface(s, t);
      if (is_deadend(t)) {
        make_deadend(s);
        return Λ;
      }
    }
    if (¬is_in_K(t) ∧ t ≠ tau) make_deadend(s);
    return Λ;
  }
```

53. Find all of the gradient paths starting at σ which descend less than p, and return the best one. If *grad_path* is not Λ return a list of the edges in the best gradient path. The *flags* parameter, if odd, is experimental code which changes the behavior by returning if possible a critical edge which is connected to σ by at least one gradient path and could possibly generate persistent homology.

For performance reasons, the lists used are declared static so they do not have to be initialized each time the routine is called. I found that without doing this this routine spent much of its time calling *malloc*. As a consequence this routine is not reentrant. If this is ever a problem, just delete the words **static** below.

Note that here and elsewhere we are essentially just searching for paths in a graph, in this case for nodes connected by exactly one path. No doubt there are sophisticated and well-known algorithms for doing this which would improve performance. As an excercise the reader can improve on these naive implementations.

```
⟨ Subroutines finding gradient paths 51 ⟩ +≡
  edge *FindGradPaths12(triangle *sigma, int p, list *grad_path, int flags)
  {
    static olist *graph;
    static list *to_do;
    static list *crits;
    static int first = 0;
    struct edge_graph {
      long up;      /* first uplink */
      long count;      /* number of uplinks */
    } r, *q;
    edge *e;
    triangle *t;
    long m;
    ⟨ initialize lists graph, crits, and to_do 54 ⟩
    m = -1;      /* indicate Λ uplink */
    e = Λ;
    t = sigma;
    ⟨ put eligible edges of t on graph, crits, and to_do 55 ⟩
    while (¬list_is_empty(to_do)) {
      list_pop(to_do, &m);
      e = (edge *) olist_get_key(graph, m);
      t = r12(e);      /* get the t which is paired with e */
      ⟨ put eligible edges of t on graph, crits, and to_do 55 ⟩
    }
    if (flags & 1) ⟨ find a persistent critical edge e in crits 82 ⟩
    else ⟨ find the critical edge e with only one grad path so value(e) is maximized 56 ⟩
    return e;
  }
```

54. ⟨ initialize lists *graph*, *crits*, and *to_do* 54 ⟩ ≡

```
if (first ≡ 0) {
  olist_initialize(&graph, sizeof(struct edge_graph));
  list_initialize(&crits, sizeof(long));
  list_initialize(&to_do, sizeof(long));
  first = 1;
}
else {
  olist_clear(graph);
  list_clear(crits);
  list_clear(to_do);
}
```

This code is used in section 53.

55. ⟨ put eligible edges of t on *graph*, *crits*, and *to_do* 55 ⟩ ≡

```
{
  int i;
  long n;
  int flag;
  edge *ep;
  int livecount;
  livecount = 0;
  for (i = 0; i < 3; i++) {
    ep = get_edge(t, i);
    if (e ≠ ep ∧ ((is_paired_up(ep) ∧ ¬is_deadend(ep)) ∨ is_critical(ep))) {
      livecount++;
      if (value(ep) ≤ value(sigma) − p) continue;
      r.up = m;
      r.count = 0;
      n = olist_find_add(graph, (long) ep, &r, &flag);
      q = (struct edge_graph *) olist_entry(graph, n);
      q→count++;
      if (¬flag) {
        if (is_critical(ep)) list_push(crits, &n);
        else list_push(to_do, &n);
      }
    }
  }
  if (livecount ≡ 0 ∧ e ≠ Λ) make_deadend(e);
}
```

This code is used in section 53.

56. ⟨ find the critical edge e with only one grad path so $value(e)$ is maximized 56 ⟩ ≡

```
{
  int best_value, val, bestm;
  bestm = -1;
  while (¬list_is_empty(crits)) {
    list_pop(crits, &m);
    q = (struct edge_graph *) olist_entry(graph, m);
    val = value((edge *) olist_get_key(graph, m));
    if (q→count ≡ 1 ∧ (bestm < 0 ∨ val > best_value))
      ⟨ replace bestm by m if there's just one path to it 57 ⟩
  }
  if (bestm ≥ 0) e = (edge *) olist_get_key(graph, bestm);
  else e = Λ;
  if (e ≠ Λ ∧ grad_path ≠ Λ) ⟨ put gradient path to e on grad_path 58 ⟩
}
```

This code is used in section 53.

57. ⟨ replace $bestm$ by m if there's just one path to it 57 ⟩ ≡

```
{
  while (q→up ≥ 0) {
    q = (struct edge_graph *) olist_entry(graph, q→up);
    if (q→count > 1) break;
  }
  if (q→count ≡ 1) {
    best_value = val;
    bestm = m;
  }
}
```

This code is used in section 56.

58. ⟨ put gradient path to e on $grad\_path$ 58 ⟩ ≡

```
{
  list_clear(grad_path);
  m = bestm;
  while (m ≥ 0) {
    plist_push(grad_path, (edge *) olist_get_key(graph, m));
    m = ((struct edge_graph *) olist_entry(graph, m))→up;
  }
}
```

This code is used in section 56.

59. Check if t is a bad tetrahedron where a gradient path could split and rejoin. It returns -1 if not, otherwise it returns i so that *get_face*(t, i) is bad.

```
⟨ Subroutines finding gradient paths 51 ⟩ +≡
  int splitrejoin(tetrahedron *t)
  {
    int i, j;
    edge *e, *ep;
    triangle *s;
    for (i = 0; i < 4; i++) {
      s = get_face(t, i);
      if (is_paired_down(s)) {
        e = r21(s);
        for (j = 0; j < 3; j++) {
          ep = get_edge(s, j);
          if (ep ≠ e ∧ (¬is_paired_up(ep) ∨ triangle_in_tetrahedron(r12(ep), t) < 0)) break;
        }
        if (j ≡ 3) return i;
      }
    }
    return −1;
  }
```

60. Fix up a bad split rejoin tetrahedron t with bad face *get_face*(t, n). Return the new triangle to which the bad edge is newly paired.

```
⟨ Subroutines finding gradient paths 51 ⟩ +≡
  triangle *unsplitrejoin(tetrahedron *t, int n)
  {
    triangle *s, *sp;
    edge *e;
    s = get_face(t, n);
    e = r21(s);
    sp = other_face(e, s, t);
    if (¬is_in_K(t)) return Λ;      /* can't fix if not in K */
    if (is_critical(sp)) return Λ;      /* can't fix if critical */
    if (is_paired_down(sp)) abort_message("tetrahedron␣surrounded␣by␣2->1␣triangles");
        /* I think this is impossible */
    pair23(s, t);
    pair12(e, sp);
    return sp;
  }
```

61. Find all gradient paths from *sigma* and return them encoded in *edges*. If *options* & 3 ≡ 0, then no pruning is done, if *options* & 3 ≡ 1, then lazy pruning is done, if *options* & 3 ≡ 2, then full pruning is done. Pruning deletes edges which are not connected to a critical edge by some gradient path. If *options* & 4 is set, then the count field in the edges data structure will be filled in. The ordered list *edges* is a list of structures *grad12_struct* below, one for each edge connected to *sigma*, with key the list index of the edge. The fields *links* have the following meaning for an edge e: First suppose that e is paired with a triangle t. Let $i = 0, 1, 2$ be the index of e in t. Then *links*[i] is the list index of an edge *ep* so that e is contained in the triangle paired with *ep*. For $j \neq i$, *links*[j] is the list index of another edge *epp* so that the j-th edge of t is contained in the triangle paired with *epp*. In case e is critical, then *links*[0] is the list index of an edge *ep* so that e is in the triangle paired with *ep* and *links*[1] is the list index of another critical edge. In other words we have a directed graph G without cycles so that the vertices of G are edges of our complex which are paired to triangles, or are critical. There is a directed path in G from e' to e if e is in the triangle paired with e' and $e \neq e'$. Then one of the links of e is an e' pointing to e, and the other two are e'' pointing to an edge also pointed to by e. It still sounds confusing but it works. *flags* & 3 is the index of the edge in its paired triangle, or 3 if critical; If complete pruning is requested (*options* & 3 ≡ 2) and *flags* & 4 is set, then the edge is connected to a critical edge by a gradient path, i.e., it cannot be pruned. The return value is the list index of a critical edge, so that all critical edges are obtained by following the *links*[1]. The *count* field gives the signed number of gradient paths going through the edge (counting orientation). If *flags* &8 is set, then the count field in the edges data structure is filled in. If *flags* & 16 is set, then the count field in the edges data structure cannot yet be filled in.

⟨ `MorseExtract.h` 61 ⟩ ≡

```
struct grad12_struct {
  long links[3];
  long count;
  int flags;
};
```

See also sections 83, 88, 92, 101, 107, 111, 116, and 128.

62. ⟨ Subroutines finding gradient paths 51 ⟩ +≡

```
long find_all_grad12_paths(triangle *sigma, olist *edges, int options)
{
  int kk = -1;
  edge *ep;
  long todo = -2;
  long this;
  long crit = -1;
  struct grad12_struct *p;
  triangle *f;
  olist_clear(edges);
  do {
    this = todo;
    if (this ≥ 0) {
      p = (struct grad12_struct *) olist_entry(edges, this);
      ep = id2edge(olist_get_key(edges, this));
      kk = p→flags & 3;
      if (kk) {
        todo = p→links[0];
        p→links[0] = -1;
      }
      else {
        todo = p→links[1];
        p→links[1] = -1;
      }
      f = r12(ep);
    }
    else f = sigma;
    ⟨ find-add edges of f to edges if needed 63 ⟩
  } while (todo ≥ 0);
  if ((options & 3) ≡ 2) ⟨ prune edges 65 ⟩
  if (options & 4) ⟨ fill in count field 67 ⟩
  return crit;
}
```

63. ⟨ find-add edges of f to *edges* if needed 63 ⟩ ≡

```
{
  int k;
  edge *e;
  int livecount = 0;
  for (k = 0; k < 3; k++) {
    if (k ≡ kk) continue;
    e = get_edge(f, k);
    ⟨ find-add e to edges if needed 64 ⟩
  }
  if (livecount ≡ 0 ∧ (options & 3) ∧ this ≥ 0) {
    make_deadend(f);
    make_deadend(ep);
  }
}
```

This code is used in section 62.

64. ⟨ find-add e to *edges* if needed 64 ⟩ ≡

```
{
  struct grad12_struct r, *q;
  triangle *t;
  int flag;
  int j;
  long id;
  if (is_paired_up(e) ∧ (¬is_deadend(e) ∨ (options & 3) ≡ 0)) {
    livecount++;
    id = olist_find_add(edges, edge_id(e), &r, &flag);
    q = (struct grad12_struct *) olist_entry(edges, id);
    if (flag)      /* already on list */
    {
      p→links[k] = q→links[q→flags & 3];
      q→links[q→flags & 3] = this;
    }
    else      /* newly added to list */
    {
      q→links[1] = q→links[2] = -1;
      t = r12(e);
      q→flags = edge_in_triangle(e, t);
      q→links[q→flags] = this;
      if (q→flags) q→links[0] = todo;
      else q→links[1] = todo;
      todo = id;
    }
  }
  else if (is_critical(e)) {
    livecount++;
    id = olist_find_add(edges, edge_id(e), &r, &flag);
    q = (struct grad12_struct *) olist_entry(edges, id);
    if (flag) {
      p→links[k] = q→links[0];
      q→links[0] = this;
    }
    else {
      q→links[0] = this;
      q→links[1] = crit;
      q→flags = 3;
      crit = id;
    }
  }
}
```

This code is used in section 63.

65. ⟨ prune *edges* 65 ⟩ ≡

```
{
  long j;
  struct grad12_struct *p;
  edge *e;
  list *todo_list;
  list_initialize(&todo_list, sizeof(long));
  todo = crit;
  while (todo ≥ 0) {
    ⟨ mark all edges on critical paths to this critical edge 66 ⟩
    p = (struct grad12_struct *) olist_entry(edges, todo);
    todo = p→links[1];
  }
  list_abandon(&todo_list);
  for (j = 0; j < olist_count(edges); j++) {
    p = (struct grad12_struct *) olist_entry(edges, j);
    if (p→flags & 4) continue;
    e = id2edge(olist_get_key(edges, j));
    make_deadend(e);
    make_deadend(r12(e));
  }
}
```

This code is used in section 62.

66. ⟨ mark all edges on critical paths to this critical edge 66 ⟩ ≡

```
{
  long next, id;
  struct grad12_struct *q;
  int i;
  list_clear(todo_list);
  list_push(todo_list, &todo);
  while (¬list_is_empty(todo_list)) {
    list_pop(todo_list, &next);
    p = (struct grad12_struct *) olist_entry(edges, next);
    if (p→flags & 4) continue;
    p→flags |= 4;
    i = p→flags & 3;
    if (i ≡ 3) id = p→links[0];
    else id = p→links[i];
    e = id2edge(olist_get_key(edges, next));
    while (id ≥ 0) {
      q = (struct grad12_struct *) olist_entry(edges, id);
      ep = id2edge(olist_get_key(edges, id));
      if ((q→flags & 4) ≡ 0) {
        list_push(todo_list, &id);
      }
      i = edge_in_triangle(e, r12(ep));
      if (i < 0) abort_message("error");
      id = q→links[i];
    }
  }
}
```

This code is used in section 65.

67. ⟨ fill in count field 67 ⟩ ≡

```
{
  list *todo_list;
  long id = -1, *idp;
  struct grad12_struct *q;
  int error_check;
  list_initialize(&todo_list, sizeof(long));
  list_push(todo_list, &id);
  while (¬list_is_empty(todo_list)) {
    list_read_init(todo_list);
    error_check = 1;
    while ((idp = (long *) list_read(todo_list)) ≠ Λ) {
      if (*idp < 0) {
        list_read_delete(todo_list);
        kk = -1;
        f = sigma;
      }
      else {
        p = (struct grad12_struct *) olist_entry(edges, *idp);
        if (p->flags & 24) continue;
        ⟨ see if all incoming faces are counted 68 ⟩
      }
      if (f ≠ Λ) {
        error_check = 0;
        ⟨ add edges of f to todo_list 69 ⟩
      }
    }
    if (error_check ∧ ¬list_is_empty(todo_list)) abort_message("count␣error");
  }
  list_abandon(&todo_list);
}
```

This code is used in section 62.

68. ⟨ see if all incoming faces are counted 68 ⟩ ≡

```
{
  edge *e;
  int i;
  ep = id2edge(olist_get_key(edges, *idp));
  kk = p→flags & 3;
  if (kk ≡ 3) id = p→links[0];
  else id = p→links[kk];
  while (id ≥ 0) {
    q = (struct grad12_struct *) olist_entry(edges, id);
    if ((q→flags & 8) ≡ 0) break;
    e = id2edge(olist_get_key(edges, id));
    i = edge_in_triangle(ep, r12(e));
    id = q→links[i];
  }
  if (id < 0) ⟨ count incoming faces to ep 71 ⟩
  else p→flags |= 16;
  if (id ≥ 0 ∨ is_critical(ep)) f = Λ;
  else f = r12(ep);
}
```

This code is used in section 67.

69. ⟨ add edges of f to *todo_list* 69 ⟩ ≡

```
{
  int k;
  edge *e;
  for (k = 0; k < 3; k++) {
    if (k ≡ kk) continue;
    e = get_edge(f, k);
    ⟨ add e to todo_list 70 ⟩
  }
}
```

This code is used in section 67.

70. ⟨ add e to *todo_list* 70 ⟩ ≡

```
{
  int flag;
  long qid;
  if (is_critical(e) ∨ (is_paired_up(e) ∧ (¬is_deadend(e) ∨ (options & 3) ≡ 0))) {
    qid = olist_find_add(edges, edge_id(e), Λ, &flag);
    if (¬flag) abort_message("count␣error␣2");
    q = (struct grad12_struct *) olist_entry(edges, qid);
    if ((q→flags & 32) ≡ 0) {
      list_read_insert(todo_list, &qid);
      q→flags |= 32;
    }
    else q→flags &= #fffffef;
  }
}
```

This code is used in section 69.

71. ⟨ count incoming faces to *ep* 71 ⟩ ≡

```
{
  edge *e;
  int i;
  long ct = 0;
  long orient;
  triangle *t;
  p→flags &= ^fffffffef;
  p→flags |= 8;
  list_read_delete(todo_list);
  kk = p→flags & 3;
  if (kk ≡ 3) id = p→links[0];
  else id = p→links[kk];
  while (id ≥ 0) {
    q = (struct grad12_struct *) olist_entry(edges, id);
    e = id2edge(olist_get_key(edges, id));
    t = r12(e);
    i = edge_in_triangle(ep, t);
    id = q→links[i];
    ct += edge_orient(t, i) * q→count;
  }
  if (id ≡ −2) {
    ct += edge_orient(sigma, edge_in_triangle(ep, sigma));
  }
  if (is_critical(ep)) p→count = ct;
  else p→count = −ct * edge_orient(r12(ep), p→flags & 3);
}
```

This code is used in section 68.

72. Find all gradient paths ending at tau. Only now the key is a triangle id, rather than an edge id. The list *crits* is a list of indices of items in *triangles* which are critical triangles.

```
#define is_xdeadend(s) ((s)→type & #1000)
#define make_xdeadend(s) ((s)→type |= #1000)
⟨ Subroutines finding gradient paths 51 ⟩ +≡
  long find_all_backward_grad12_paths(edge *tau, olist *triangles, list *crits, int options)
  {
    int kk = −1;
    triangle *f;
    edge *ep;
    long todo = −1;
    long this;
    struct grad12_struct *p;
    long start = −1;
    list *todo_list;
    olist_clear(triangles);
    if (crits ≠ Λ) list_clear(crits);
    if ((options & 3) ≡ 2) list_initialize(&todo_list, sizeof(long));
    do {
      this = todo;
      if (this ≥ 0) {
        p = (struct grad12_struct *) olist_entry(triangles, this);
        f = id2triangle(olist_get_key(triangles, this));
        ep = r21(f);
        kk = p→flags & 3;
        todo = p→links[kk];
        p→links[kk] = −1;
      }
      else ep = tau;
      ⟨ find-add cofaces of ep to triangles if needed 73 ⟩
    } while (todo ≥ 0);
    if ((options & 3) ≡ 2) ⟨ prune triangles 75 ⟩
    return start;
  }
```

73. ⟨ find-add cofaces of *ep* to *triangles* if needed 73 ⟩ ≡

```
{
  tetrahedron *t;
  int link_count = 0;
  f = ep→links[2];
  t = coface(f, 0);
  do {
    if (is_in_K(f) ∧ (is_critical(f) ∨ ((¬is_xdeadend(f) ∨ (options & 3) ≡ 0) ∧ is_paired_down(f) ∧ ep ≠
          r21(f)))) {
      link_count++;
      ⟨ find-add f to triangles 74 ⟩
    }
    t = other_coface(f, t);
    f = other_face(ep, f, t);
  } while (f ≠ ep→links[2]);
  if (link_count ≡ 0 ∧ (options & 3) ∧ this ≥ 0) {
    make_xdeadend(ep);
    make_xdeadend(r12(ep));
  }
}
```

This code is used in section 72.

74. ⟨ find-add f to *triangles* 74 ⟩ ≡

```
{
  struct grad12_struct r, *q;
  long id;
  int flag;
  int k;
  id = olist_find_add(triangles, triangle_id(f), &r, &flag);
  q = (struct grad12_struct *) olist_entry(triangles, id);
  k = edge_in_triangle(ep, f);
  if (this ≥ 0) p = (struct grad12_struct *) olist_entry(triangles, this);
  if (flag)      /* already on list */
  {
    if (q→links[k] ≥ 0) abort_message("backgrad␣error␣1");
    if (this < 0) abort_message("backgrad␣error␣2");
    q→links[k] = p→links[kk];
    p→links[kk] = id;
  }
  else      /* newly added to list */
  {
    q→links[0] = q→links[1] = q→links[2] = -2;
    if (this ≥ 0) {
      q→links[k] = p→links[kk];
      p→links[kk] = id;
    }
    else {
      q→links[k] = start;
      start = id;
    }
    if (is_critical(f)) {
      q→flags = 3;
      if (crits ≠ Λ) list_push(crits, &id);
      if ((options & 3) ≡ 2) list_push(todo_list, &id);
    }
    else {
      q→flags = edge_in_triangle(r21(f), f);
      q→links[q→flags] = todo;
      todo = id;
    }
  }
}
```

This code is used in section 73.

75. ⟨ prune *triangles* 75 ⟩ ≡

```
{
  long id;
  int k;
  edge *e;
  struct grad12_struct r, *q;
  triangle *f;
  int flag;
  while (¬list_is_empty(todo_list)) {
    list_pop(todo_list, &id);
    p = (struct grad12_struct *) olist_entry(triangles, id);
    if (p→flags & 4) continue;
    p→flags |= 4;
    kk = p→flags & 3;
    f = id2triangle(olist_get_key(triangles, id));
    for (k = 0; k < 3; k++) {
      if (k ≡ kk ∨ p→links[k] < −1) continue;
      e = get_edge(f, k);
      if (is_paired_up(e) ∧ ¬is_deadend(e)) {
        if (is_xdeadend(e)) abort_message("back␣grad␣error␣3");
        id = olist_find_add(triangles, triangle_id(r12(e)), &r, &flag);
        q = (struct grad12_struct *) olist_entry(triangles, id);
        if (flag ≡ 0) {
          q→flags = edge_in_triangle(e, r12(e));
          printf("back␣grad␣puzzle\n");
        }
        if ((q→flags & 4) ≡ 0) list_push(todo_list, &id);
      }
    }
  }
  list_abandon(&todo_list);
  for (id = 0; id < olist_count(triangles); id++) {
    p = (struct grad12_struct *) olist_entry(triangles, id);
    if (p→flags & 4) continue;
    f = id2triangle(olist_get_key(triangles, id));
    make_deadend(f);
    make_deadend(r21(f));
  }
}
```

This code is used in section 72.

76. Persistent critical simplices. This is experimental code, trying to identify critical simplices which might generate persistent homology. So far in random examples all but a few critical simplices end up being persistent.

```
⟨ find persistent critical simplices 76 ⟩ ≡
  ⟨ make all critical vertices persistent 77 ⟩
  ⟨ find persistent critical edges 78 ⟩
  ⟨ find persistent critical triangles 81 ⟩
  ⟨ find persistent critical tetrahedra 79 ⟩
```

77.
```
⟨ make all critical vertices persistent 77 ⟩ ≡
  {
    vertex *v;
    list_read_init(crit[0]);
    while ((v = (vertex *) plist_read(crit[0])) ≠ Λ) {
      if (¬is_critical(v)) list_read_delete(crit[0]);
      else make_persistent(v);
    }
  }
```
This code is used in section 76.

78. This is called after canceling is done, so if v is not Λ then we know that the two gradient paths from e end at the same vertex v and $value(v) \geq value(e) - p$.

```
⟨ find persistent critical edges 78 ⟩ ≡
  {
    edge *e;
    vertex *v;
    list_read_init(crit[1]);
    while ((e = (edge *) plist_read(crit[1])) ≠ Λ) {
      if (¬is_critical(e)) list_read_delete(crit[1]);
      else {
        v = FindGrad01(get_vertex(e, 0), value(e) − p);
        if (v ≡ Λ) make_persistent(e);
      }
    }
  }
```
This code is used in section 76.

79.
```
⟨ find persistent critical tetrahedra 79 ⟩ ≡
  ⟨ make all critical tetrahedra persistent 80 ⟩
  {
    triangle *t;
    tetrahedron *te;
    list_read_init(crit[2]);
    while ((t = (triangle *) plist_read(crit[2])) ≠ Λ) {
      if (is_persistent(t)) {
        te = FindGrad23(coface(t, 0), value(t) + p);
        if (te ≠ Λ) unmake_persistent(te);
      }
    }
  }
```
This code is used in section 76.

80. ⟨ make all critical tetrahedra persistent 80 ⟩ ≡

```
{
  tetrahedron *t;
  list_read_init(crit[3]);
  while ((t = (tetrahedron *) plist_read(crit[3])) ≠ Λ) {
    if (¬is_critical(t)) list_read_delete(crit[3]);
    else make_persistent(t);
  }
}
```

This code is used in section 79.

81. ⟨ find persistent critical triangles 81 ⟩ ≡

```
{
  triangle *t;
  void *q;
  list_read_init(crit[2]);
  while ((t = (triangle *) plist_read(crit[2])) ≠ Λ) {
    if (¬is_critical(t)) list_read_delete(crit[2]);
    else {
      q = FindGradPaths12(t, p, Λ, 1);
      if (q ≡ Λ) make_persistent(t);
    }
  }
}
```

This code is used in section 76.

82. ⟨ find a persistent critical edge e in *crits* 82 ⟩ ≡

```
{
  edge *ee;
  e = Λ;
  while (¬list_is_empty(crits)) {
    list_pop(crits, &m);
    ee = (edge *) olist_get_key(graph, m);
    if (is_persistent(ee)) {
      e = ee;
      break;
    }
  }
}
```

This code is used in section 53.

83. Unordered Lists. Unordered lists work as LIFO stacks. We can push to and pop from a **list**. We can find the n-th entry on a **list**. We can read the entries of a **list** one by one, deleting those we no longer want.

format *list* *int*

```
⟨ MorseExtract.h 61 ⟩ +≡
  typedef struct {
    long *body;
    unsigned int size;      /* size of each entry, not including padding */
    unsigned int padded_size;      /* size of each entry, in long words */
    unsigned long body_length;      /* capacity of body */
    unsigned long length;      /* number of entries on list */
    unsigned long read_index;
    unsigned long read_delete_index;
  } list;
#define list_count(l) ((l)→length)
#define list_is_empty(l) ((l)→length ≡ 0)
#define list_clear(l) ((l)→length = 0)
#define list_entry(l, n) ((l)→body + (n) * ((l)→padded_size))
```

84. Initialize a list, setting up storage

```
⟨ List functions 84 ⟩ ≡
  void list_initialize(list **l, unsigned int sz)
  {
    *l = (list *) malloc(sizeof(list));
    if ((*l) ≡ Λ) abort_message("Out of memory");
    (*l)→size = sz;
    (*l)→padded_size = (sz + sizeof(long) − 1)/sizeof(long);
    (*l)→length = 0;
    (*l)→body = Λ;
    (*l)→body_length = 0;
  }
```

See also sections 85, 86, 87, 89, 90, 93, 94, 95, 96, 97, and 98.

This code is used in section 1.

85. Abandon a list, freeing storage used.

```
⟨ List functions 84 ⟩ +≡
  void list_abandon(list **l)
  {
    if ((*l)→body_length > 0) free((*l)→body);
    free(*l);
    *l = Λ;
  }
```

86. *list_push* pushes a new entry pointed to by q to the top of the list. The variant *plist_push* pushes the pointer q directly to the list (as opposed to the contents pointed to by q).

```
⟨ List functions 84 ⟩ +≡
  void *list_push(list *l, void *q)
  {
    long *p;
    if (l→body_length ≤ l→length) {
      l→body_length = l→length + 1000;
      l→body = (long *) realloc(l→body, l→body_length * l→padded_size * sizeof(long));
      if (l→body ≡ Λ) abort_message("out of memory");
    }
    p = list_entry(l, l→length);
    memcpy(p, q, l→size);
    l→length++;
    return p;
  }
  void plist_push(list *l, void *q)
  {
    void *p = q;
    list_push(l, &p);
  }
```

87. *list_pop* pops the top entry into the region pointed to by q, and deletes it from the list. It copies the top entry to q. The variant *plist_pop* returns the entry from a pointer list directly.

```
⟨ List functions 84 ⟩ +≡
  void list_pop(list *l, void *q)
  {
    long *p;
    if (l→length ≡ 0) abort_message("pop from empty list");
    l→length--;
    p = list_entry(l, l→length);
    memcpy(q, p, l→size);
  }
  void *plist_pop(list *l)
  {
    void *p;
    list_pop(l, &p);
    return p;
  }
```

88. Routine to read the entries of a list one by one, and deleting those you don't wish to retain on the list. To use it, first call *list_read_init*. Then repeated calls to *list_read* will return the entries on the list, until a Λ is returned signifying the end. If you wish to delete the entry you last read, just call *list_read_delete*. If your list is a list of pointers, then the companion pointer list routine *plist_read* returns the pointer, rather than its address. Since the end of the list is signaled by returning a Λ pointer it will only work well if the list of pointers contains no Λ pointers, otherwise you might stop too early. So in this case you would need some alternate method to recognize the end of the list.

Warnings: if you use *list_read_delete* then you must read through to the end of the list.

```
⟨ MorseExtract.h 61 ⟩ +≡
#define list_read_init(l) ((l)→read_index = (l)→read_delete_index = 0)
#define list_read_delete(l) ((l)→read_delete_index −−)
```

89.
```
⟨ List functions 84 ⟩ +≡
  void *list_read(list *l)
  {
    void *p, *pp;
    if (l→read_index ≥ l→length) {      /* we are at the end of the list */
      l→length = l→read_index = l→read_delete_index;
      return (Λ);
    }
    pp = list_entry(l, l→read_delete_index);
    if (l→read_index ≠ l→read_delete_index) {
      p = list_entry(l, l→read_index);
      memcpy(pp, p, l→size);
    }
    l→read_delete_index++;
    l→read_index++;
    return pp;
  }
  void *plist_read(list *l)
  {
    void **r;
    r = list_read(l);
    if (r ≠ Λ) return *r;
    else return Λ;
  }
```

90. The following *list_read_insert* will insert an item in a list you are reading, placing it just before the most recently read item if there is room, or if not, placing it at the end.

```
⟨ List functions 84 ⟩ +≡
  void *list_read_insert(list *l, void *p)
  {
    if (l→read_index ≡ l→read_delete_index) list_push(l, p);
    else {
      memcpy(list_entry(l, l→read_delete_index), p, l→size);
      l→read_delete_index++;
    }
  }
```

91. Ordered Lists. Ordered lists are implemented naively and could no doubt be made more efficient. Items on an **olist** are ordered by an integer key.

Portability issue: these are sometimes used with a simplex pointer as the key, so that requires **sizeof**(**long**) = **sizeof** (*simplex* ∗). When this is done, of course we don't really care about the resulting order, we are just using ordered lists to find entries quickly. One could also implement such lists by adding auxiliary fields to each simplex, but then memory usage is increased.

format *olist* *int*

92. ⟨ `MorseExtract.h` 61 ⟩ +≡

```
struct olist_key {
  long high;
  long low;
  long eq;
  long k;      /* the list is ordered by this key k */
};
typedef struct {
  list *keys;
  list *entries;
  long top;
  long free;
  int sz;
} olist;
#define olist_is_empty(l) ((l)->top < 0)
#define olist_entry(l,n) list_entry ((l)->entries,n)
#define olist_get_key(l,n) (((struct olist_key *) list_entry((l)->keys,n))->k)
#define olist_count(l) list_count ((l)->entries)
```

93. ⟨ List functions 84 ⟩ +≡

```
void olist_initialize(olist **l,int sz)
{
  *l = (olist *) malloc(sizeof(olist));
  if ((*l) ≡ Λ) abort_message("Out of memory");
  list_initialize(&((*l)->keys),sizeof(struct olist_key));
  list_initialize(&((*l)->entries),sz);
  (*l)->top = -1;
  (*l)->free = -1;
  (*l)->sz = sz;
}
```

94. ⟨ List functions 84 ⟩ +≡

```
void olist_abandon(olist **l)
{
  list_abandon(&((*l)->keys));
  list_abandon(&((*l)->entries));
  free(*l);
  *l = Λ;
}
```

95. ⟨ List functions 84 ⟩ +≡

```
void olist_clear(olist *l)
{
  list_clear(l→keys);
  list_clear(l→entries);
  l→top = −1;
  l→free = −1;
}
```

96. Pop the smallest entry from the list, return its key, and put the list entry in p.

⟨ List functions 84 ⟩ +≡

```
long olist_min(olist *l, void *p)
{
  struct olist_key *q, *r;
  long min_index;
  min_index = l→top;
  if (min_index < 0) abort_message("min␣of␣empty␣list");
  q = (struct olist_key *) list_entry(l→keys, min_index);
  if (q→low ≥ 0) {
    while (q→low ≥ 0) {
      r = q;
      min_index = q→low;
      q = (struct olist_key *) list_entry(l→keys, min_index);
    }
    if (q→eq ≥ 0) {
      r = q;
      min_index = q→eq;
      q = (struct olist_key *) list_entry(l→keys, min_index);
      r→eq = q→eq;
    }
    else r→low = q→high;
  }
  else if (q→eq ≥ 0) {
    r = q;
    min_index = q→eq;
    q = (struct olist_key *) list_entry(l→keys, min_index);
    r→eq = q→eq;
  }
  else l→top = q→high;
  memcpy(p, list_entry(l→entries, min_index), l→sz);
  q→low = l→free;
  l→free = min_index;
  return q→k;
}
```

97. Add an entry p with key m to the ordered list. Return the index of the entry.

```
⟨ List functions 84 ⟩ +≡
  long olist_add(olist *l, long m, void *p)
  {
    struct olist_key *q, *r;
    long b, *last;
    b = olist_next_free(l);
    ⟨ point r the b-th entry and copy p to it 99 ⟩
    last = &(l→top);
    while ((*last) ≥ 0) {
      q = (struct olist_key *) list_entry(l→keys, *last);
      if (m < q→k) last = &(q→low);
      else if (m > q→k) last = &(q→high);
      else {
        r→eq = *last;
        r→low = q→low;
        r→high = q→high;
        q→low = −1;
        q→high = −1;
        break;
      }
    }
    *last = b;
    return b;
  }
```

98. This searches the ordered list l for an entry with key m. If it finds one, it returns its index and sets *flag*. If it doesn't find one, and p is not NULL, it creates an entry with key m and moves $*p$ into that entry and resets *flag*.

```
#define olist_next_free(l) ((l)→free ≥ 0) ? (l)→free : list_count((l)→entries)
⟨ List functions 84 ⟩ +≡
  long olist_find_add(olist *l, long m, void *p, int *flag)
  {
    struct olist_key *q, *r;
    long b, *last;
    last = &(l→top);
    while ((*last) ≥ 0) {
      q = (struct olist_key *) list_entry(l→keys, *last);
      if (m < q→k) last = &(q→low);
      else if (m > q→k) last = &(q→high);
      else      /* if duplicate key, return entry */
      {
        if (flag ≠ Λ) *flag = 1;
        return *last;
      }
    }
    if (p ≠ Λ) {
      b = *last = olist_next_free(l);
      ⟨ point r the b-th entry and copy p to it 99 ⟩
    }
    else b = −1;
    if (flag ≠ Λ) *flag = 0;
    return b;
  }
```

99.
```
⟨ point r the b-th entry and copy p to it 99 ⟩ ≡
  {
    struct olist_key rr;
    rr.high = rr.low = rr.eq = −1;
    rr.k = m;
    if (l→free ≥ 0) {
      memcpy(list_entry(l→entries, b), p, l→sz);
      r = (struct olist_key *) list_entry(l→keys, b);
      l→free = r→low;
      memcpy(r, &rr, sizeof(struct olist_key));
    }
    else {
      list_push(l→entries, p);
      r = (struct olist_key *) list_push(l→keys, &rr);
    }
  }
```

This code is used in sections 97 and 98.

100. Navigating Simplicial Complexes. Simplices of all dimensions have a common initial part, there is a word with all sorts of bit flags. Yes I know I should implement this with all these bit fields as different entries in the struct. Maybe later. The meaning of the bits in the type field is:

- Bits 0 and 1 give the dimension.
- Bit 2 is set if the simplex is in K.
- Bit 3 is set if the simplex is critical.
- Bit 4 is set if the simplex is paired with a codimension one face.
- Bits 5 and 6 give the index of that face if bit 4 is set. If bit 3 is set then bit 5 is set the simplex is persistent critical. If bits 3 and 4 are not set and the simplex is a triangle then bit 5 gives the index of the coface to which the simplex is paired.
- Bit 7 is set if the h field is valid.
- Bit 8 is set in an edge if the edge is paired up and it is known that all gradient paths from the edge do not lead to critical edges.
- Bit 9 is set in an edge e when that edge is processed by ⟨ extract the lower Star of v 5 ⟩ with $v = get\_vertex(e, 0)$.
- Bit 10 is set in an edge e when that edge is processed by ⟨ extract the lower Star of v 5 ⟩ with $v = get\_vertex(e, 1)$.
- Bits 11-15 are available for other uses. For example, if this program were adapted to use the CGAL representation of a complex, edges and triangles are not represented explicitly but the unused bits could indicate, say, which face of a tetrahedron the triangle is, or which of its two vertices an edge is.

The field h is the maximum of the values of the vertices of the simplex, if bits 2 and 7 of *type* are set. Not essential, but there was this unused space, so what the heck. I expect 16 bit resolution of the function is good enough.

The remaining field *links* is a variable sized array of pointers to other simplices and is used to navigate around the complex. The entries in *links* have the following meaning:

For a vertex:

- $links[0]$ is an edge containing the vertex, or Λ if there is none.

For an edge:

- $links[0-1]$ are the vertices in the edge.
- $links[2]$ is a triangle containing the edge.

For a triangle:

- $links[0-2]$ are the edges contained in the triangle.
- $links[3-4]$ are the two tetrahedra which contain the triangle.

For a tetrahedron:

- $links[0-3]$ are the triangles contained in the tetrahedron.

format *vertex* *int*
format *edge* *int*
format *triangle* *int*
format *tetrahedron* *int*

101. ⟨ MorseExtract.h 61 ⟩ +≡

```
  typedef struct vertexstruct {
    short type;
    short h;
    void *links[1];
  } vertex;
  typedef struct edgestruct {
    short type;
    short h;
    void *links[3];
  } edge;
  typedef struct trianglestruct {
    short type;
    short h;
    void *links[5];
  } triangle;
  typedef struct tetrahedronstruct {
    short type;
    short h;
    void *links[4];
  } tetrahedron;
#define get_vertex(e,i)(vertex *) ((e)->links[i])
#define get_edge_vertices(e, vl) ((vl)[0] = get_vertex(e,0), (vl)[1] = get_vertex(e,1))
#define dimension(s) ((s)->type & 3)
#define is_critical(t) ((t)->type & 8)
#define other_vertex_in_edge(u,e) ((get_vertex(e,0) == u) ? get_vertex(e,1) : get_vertex(e,0))
#define coface(t,i)(tetrahedron *) ((t)->links[3+i])
#define get_edge(t,i)(edge *) ((t)->links[i])
#define get_face(t,i)(triangle *) ((t)->links[i])
#define is_in_K(v) ((v)->type & 4)
#define value(t)(int) ((((t)->type & #80) != 0) ? (t)->h : max_value(t, dimension(t)))
#define other_coface(s,t)(tetrahedron *) (((t) == (s)->links[3]) ? (s)->links[4] : (s)->links[3])
#define is_deadend(s) ((s)->type & #100)
#define make_deadend(s) ((s)->type |= #100)
```

102. Now some functions to manipulate and navigate these simplices

```
#define set_value(t, f) ((t)→type |= #80, (t)→h = f)
⟨ Simplex functions 102 ⟩ ≡
  void get_triangle_vertices(triangle *t, vertex *vlist[3])  /* return a list of the vertices of t */
  {
    vertex *v;
    edge *e;
    e = get_edge(t, 0);
    get_edge_vertices(e, vlist);
    e = get_edge(t, 1);
    v = get_vertex(e, 1);
    if (v ≠ vlist[0] ∧ v ≠ vlist[1]) vlist[2] = v;
    else vlist[2] = get_vertex(e, 0);
  }
  void get_tetrahedron_vertices(tetrahedron *t, vertex *vlist[4])  /* return a list of the vertices of t */
  {
    vertex *vlistp[3];
    int i;
    get_triangle_vertices(get_face(t, 0), vlist);
    get_triangle_vertices(get_face(t, 1), vlistp);
    for (i = 0; i < 3; i++) {
      if (vlistp[i] ≠ vlist[0] ∧ vlistp[i] ≠ vlist[1] ∧ vlistp[i] ≠ vlist[2]) {
        vlist[3] = vlistp[i];
        break;
      }
    }
  }
```

See also sections 103, 104, 105, 106, 108, 109, 110, 112, 113, and 114.

This code is used in section 1.

103. Return ±1 depending on the orientation of the i-th edge of t.

```
⟨ Simplex functions 102 ⟩ +≡
  int edge_orient(triangle *t, int i)
  {
    vertex *vl[2], *vlp[2];
    if (i ≡ 0) return 1;
    get_edge_vertices(get_edge(t, 0), vl);
    get_edge_vertices(get_edge(t, i), vlp);
    if (vlp[0] ≡ vl[1] ∨ vlp[1] ≡ vl[0]) return 1;
    return −1;
  }
```

104. a test for when the value of one vertex is greater than another. The tiebreaker is just the address of the vertex.

```
⟨ Simplex functions 102 ⟩ +≡
  long vertex_compare(vertex *v, vertex *w)
  {
    if (value(v) < value(w)) return −1;
    else if (value(v) > value(w)) return 1;
    else return (long) v − (long) w;
  }
```

105. Find the smallest or largest valued vertex in a list of vertices of length n

```
#define smallest_vertex_in_edge(e)
        (vertex_compare(get_vertex(e, 0), get_vertex(e, 1)) < 0 ? get_vertex(e, 0) : get_vertex(e, 1))
#define largest_vertex_in_edge(e)
        (vertex_compare(get_vertex(e, 0), get_vertex(e, 1)) < 0 ? get_vertex(e, 1) : get_vertex(e, 0))
⟨ Simplex functions 102 ⟩ +≡
  int smallest_vertex(vertex *v[2], int n)
  {
    int i, j;
    for (i = 0, j = 1; j < n; j++) {
      if (vertex_compare(v[j], v[i]) < 0) i = j;
    }
    return i;
  }
  int largest_vertex(vertex *v[2], int n)
  {
    int i, j;
    for (i = 0, j = 1; j < n; j++) {
      if (vertex_compare(v[j], v[i]) > 0) i = j;
    }
    return i;
  }
```

106. This routine tests whether the simplex t is in the lower Star of the vertex v. It does tiebreaking if two vertices in t have the same value.

```
⟨ Simplex functions 102 ⟩ +≡
  int is_in_lower_Star(tetrahedron *t, vertex *v)
  {
    vertex *vlist[4];
    get_tetrahedron_vertices(t, vlist);
    return v ≡ vlist[largest_vertex(vlist, 4)];
  }
```

107. These routines test whether a given simplex is a codimension one face of another, and if it is, returns the index of that face. If it is not, then -1 is returned.

```
⟨ MorseExtract.h 61 ⟩ +≡
#define vertex_in_edge(v, e) (((v) ≡ get_vertex(e, 0)) ? 0 : (((v) ≡ get_vertex(e, 1)) ? 1 : −1))
#define edge_in_triangle(e, t)
  (((e) ≡ get_edge(t, 0)) ? 0 : (((e) ≡ get_edge(t, 1)) ? 1 : (((e) ≡ get_edge(t, 2)) ? 2 : −1)))
```

108. ⟨ Simplex functions 102 ⟩ +≡

```
int triangle_in_tetrahedron(triangle *s, tetrahedron *t)
{
  int i;
  for (i = 0; i < 4; i++) {
    if (s ≡ get_face(t, i)) return i;
  }
  return −1;
}
```

109. These routines take a codimension one face of a simplex and find the unique other codimension one face which also contains the given codimension 2 face.

⟨ Simplex functions 102 ⟩ +≡

```
edge *other_edge(vertex *v, edge *f, triangle *t)
  /* return the edge of t which contains v but is not f */
{
  int i;
  edge *e;
  for (i = 0; i < 3; i++) {
    e = get_edge(t, i);
    if (e ≡ f) continue;
    if (get_vertex(e, 0) ≡ v ∨ get_vertex(e, 1) ≡ v) return e;
  }
  return Λ;      /* error, v not in t */
}
triangle *other_face(edge *e, triangle *f, tetrahedron *t)
  /* Find the face of t which contains e but is not f */
{
  int i;
  triangle *s;
  for (i = 0; i < 4; i++) {
    s = get_face(t, i);
    if (f ≠ s ∧ edge_in_triangle(e, s) ≥ 0) return s;
  }
  return Λ;      /* error, e not in f or f not in t */
}
```

110. Here are various routines for critical simplices. In *unmake_critical*, the formerly critical simplex is not actually removed from the list of critical simplices. So when running through *crit* you may need to check if the entries are in fact critical. The routine *clean_crit* will delete all the noncritical entries from *crit*,

```
#define unmake_critical(s) (s)→type &= #ff87
#define make_critical(s) ((s)→type &= #ff87, (s)→type |= 8, plist_push(crit[dimension(s)], s))
#define is_persistent(t) ((t)→type & #20)
#define make_persistent(t) ((t)→type |= #20)
#define unmake_persistent(t) ((t)→type &= #ffdf)
⟨ Simplex functions 102 ⟩ +≡
  void clean_crit(void)
  {
    tetrahedron *t;
    triangle *f;
    edge *e;
    vertex *v;
    list *l;
    l = crit[0];
    list_read_init(l);
    while ((v = (vertex *) plist_read(l)) ≠ Λ) {
      if (¬is_critical(v)) list_read_delete(l);
    }
    l = crit[1];
    list_read_init(l);
    while ((e = (edge *) plist_read(l)) ≠ Λ) {
      if (¬is_critical(e)) list_read_delete(l);
    }
    l = crit[2];
    list_read_init(l);
    while ((f = (triangle *) plist_read(l)) ≠ Λ) {
      if (¬is_critical(f)) list_read_delete(l);
    }
    l = crit[3];
    list_read_init(l);
    while ((t = (tetrahedron *) plist_read(l)) ≠ Λ) {
      if (¬is_critical(t)) list_read_delete(l);
    }
  }
```

111. Routines to pair simplices. The routine *is_paired_up* finds whether the simplex t is paired with a simplex of one higher dimension. Likewise *is_paired_down* finds whether the simplex t is paired with a simplex of one lower dimension. The routines *rij* return the dimension j simplex paired with the dimension i simplex t. The routines *pairij* pair up the i simplex *s0* and the j simplex *s1*.

```
⟨ MorseExtract.h 61 ⟩ +≡
#define is_paired_up(t) (((t)→type & #1C) ≡ 4)
#define is_paired_down(t) (((t)→type & #1C) ≡ #14)
#define r32(t)get_face (t, ((t)→type & #60) ≫ 5)
#define r21(t)get_edge (t, ((t)→type & #60) ≫ 5)
#define r10(t)get_vertex (t, ((t)→type & #20) ≫ 5)
#define r23(t)coface (t, ((t)→type & #20) ≫ 5)
#define r12(t)(triangle *) ((t)→links[2])
#define r01(t)(edge *) ((t)→links[0])
```

112. ⟨ Simplex functions 102 ⟩ +≡

```
void pair01(vertex *s0, edge *s1)
{
  int n;
  s0→type &= #FF87;
  s1→type &= #FF87;
  s0→links[0] = s1;
  n = vertex_in_edge(s0, s1);
  s1→type |= (16 + (n ≪ 5));
}
void pair12(edge *s0, triangle *s1)
{
  int n;
  s0→type &= #FF87;
  s1→type &= #FF87;
  s0→links[2] = s1;
  n = edge_in_triangle(s0, s1);
  s1→type |= (16 + (n ≪ 5));
}
void pair23(triangle *s0, tetrahedron *s1)
{
  int n;
  s0→type &= #FF87;
  s1→type &= #FF87;
  if (s0→links[4] ≡ s1) s0→type |= #20;
  n = triangle_in_tetrahedron(s0, s1);
  s1→type |= (16 + (n ≪ 5));
}
```

113. find the maximum value of the vertices of the simplex s

```
⟨ Simplex functions 102 ⟩ +≡
  int max_value(void *s, int d)
  {
    tetrahedron *te;
    triangle *t;
    edge *e;
    int m0, m1;
    if (s ≡ Λ) return -#8000;      /* smallest value */
    switch (d) {
    case 3: te = (tetrahedron *) s;
      m0 = value(get_face(te, 0));
      m1 = value(get_face(te, 1));
      break;
    case 2: t = (triangle *) s;
      m0 = value(get_edge(t, 0));
      m1 = value(get_edge(t, 1));
      break;
    case 1: e = (edge *) s;
      m0 = value(get_vertex(e, 0));
      m1 = value(get_vertex(e, 1));
      break;
    default: abort_message("max␣value␣of␣simplex␣of␣unknown␣dimension");
      break;
    }
    return max(m0, m1);
  }
```

114. find the minimum value of the vertices of the simplex s

```
⟨ Simplex functions 102 ⟩ +≡
  int min_value(void *s, int d)
  {
    tetrahedron *te;
    triangle *t;
    edge *e;
    int m0, m1;
    if (s ≡ Λ) return #7fff;      /* largest value */
    switch (d) {
    case 3: te = (tetrahedron *) s;
      m0 = min_value(get_face(te, 0), 2);
      m1 = min_value(get_face(te, 1), 2);
      break;
    case 2: t = (triangle *) s;
      m0 = min_value(get_edge(t, 0), 1);
      m1 = min_value(get_edge(t, 1), 1);
      break;
    case 1: e = (edge *) s;
      m0 = value(get_vertex(e, 0));
      m1 = value(get_vertex(e, 1));
      break;
    default: abort_message("min␣value␣of␣simplex␣of␣unknown␣dimension");
      break;
    }
    return min(m0, m1);
  }
```

115. Debugging routines.

⟨ Debugging output 115 ⟩ ≡

```
void abort_message(char *s)
{
  printf("Error: %s", s);
  exit(1);
}
```

This code is used in section 1.

116. Constructing Complexes. The following routines allocate a new simplex. The last parameter *flag* is > 0 if the simplex is in K, is $= 0$ if the simplex is not in K, and is < 0 if you want it to be in K if and only if all its faces are in K.

```
⟨ MorseExtract.h 61 ⟩ +≡
#define vertex_id(v) ((v) − vertexlist)
#define edge_id(e) ((e) − elist)
#define triangle_id(f) ((f) − flist)
#define tetrahedron_id(t) ((t) − tlist)
#define id2vertex(id) (vertexlist + (id))
#define id2edge(id) (elist + (id))
#define id2triangle(id) (flist + (id))
#define id2tetrahedron(id) (tlist + (id))
```

117. ⟨ Subroutines for constructing complexes 117 ⟩ ≡

```
vertex *new_vertex(long id, short val, short flag)
{
  vertex *v;
  v = id2vertex(id);
  v→type = (flag) ? #84 : #80;      /* indicate in K or not */
  v→links[0] = Λ;
  v→h = val;
  return v;
}
```

See also sections 118, 119, 120, and 123.

This code is used in section 1.

118. Construct a new edge

```
⟨ Subroutines for constructing complexes 117 ⟩ +≡
edge *new_edge(long id, long vids[2], short flag)
{
  edge *e;
  short flagp;
  vertex *v0, *v1;
  e = id2edge(id);
  v0 = id2vertex(vids[0]);
  v1 = id2vertex(vids[1]);
  e→links[0] = v0;
  e→links[1] = v1;
  e→links[2] = Λ;
  e→h = max(v0→h, v1→h);
  flagp = (flag ∧ is_in_K(v0) ∧ is_in_K(v1));
  e→type = (flagp) ? 5 : 1;
  if (flag > 0 ∧ flagp ≡ 0) abort_message("vertices␣of␣edge␣in␣K␣must␣be␣in␣K");
  if (v0→links[0] ≡ Λ) v0→links[0] = e;
  if (v1→links[0] ≡ Λ) v1→links[0] = e;
  return e;
}
```

119. ⟨ Subroutines for constructing complexes 117 ⟩ +≡

```
triangle *new_triangle(long id, long eids[3], short flag)
{
  triangle *t;
  vertex *vl[3], *v[2];
  edge *e0, *e1, *e2;
  short flagp;
  e0 = id2edge(eids[0]);
  e1 = id2edge(eids[1]);
  e2 = id2edge(eids[2]);
  get_edge_vertices(e0, vl);
  get_edge_vertices(e1, v);
  if (vertex_in_edge(v[0], e0) < 0) vl[2] = v[0];
  else if (vertex_in_edge(v[1], e0) < 0) vl[2] = v[1];
  else abort_message("new␣triangle␣has␣more␣than␣three␣vertices");
  get_edge_vertices(e2, v);
  if (v[0] ≠ vl[0] ∧ v[0] ≠ vl[1] ∧ v[0] ≠ vl[2])
    abort_message("new␣triangle␣has␣more␣than␣three␣vertices-1");
  if (v[1] ≠ vl[0] ∧ v[1] ≠ vl[1] ∧ v[1] ≠ vl[2])
    abort_message("new␣triangle␣has␣more␣than␣three␣vertices-2");
  t = id2triangle(id);
  t→links[0] = e0;
  t→links[1] = e1;
  t→links[2] = e2;
  t→h = max(e0→h, e1→h);
  flagp = (flag ∧ is_in_K(e0) ∧ is_in_K(e1) ∧ is_in_K(e2));
  t→type = (flagp) ? 6 : 2;
  t→links[3] = Λ;
  t→links[4] = Λ;
  if (e0→links[2] ≡ Λ) e0→links[2] = t;
  if (e1→links[2] ≡ Λ) e1→links[2] = t;
  if (e2→links[2] ≡ Λ) e2→links[2] = t;
  if (flag > 0 ∧ flagp ≡ 0) abort_message("edges␣of␣triangle␣in␣K␣must␣be␣in␣K");
  return t;
}
```

120. ⟨ Subroutines for constructing complexes 117 ⟩ +≡

```
tetrahedron *new_tetrahedron(long id, long fids[4], short flag)
{
  tetrahedron *t;
  vertex *vl[4], *vlp[3];
  int i;
  short flagp;
  triangle *f0, *f1, *f2, *f3;
  f0 = id2triangle(fids[0]);
  f1 = id2triangle(fids[1]);
  f2 = id2triangle(fids[2]);
  f3 = id2triangle(fids[3]);
  get_triangle_vertices(f0, vl);
  get_triangle_vertices(f1, vlp);
  for (i = 0; i < 3; i++) {
    if (vlp[i] ≠ vl[0] ∧ vlp[i] ≠ vl[1] ∧ vlp[i] ≠ vl[2]) {
      vl[3] = vlp[i];
      break;
    }
  }
  ⟨ make sure vlp entries are all in vl 121 ⟩
  get_triangle_vertices(f2, vlp);
  ⟨ make sure vlp entries are all in vl 121 ⟩
  get_triangle_vertices(f3, vlp);
  ⟨ make sure vlp entries are all in vl 121 ⟩
  t = id2tetrahedron(id);
  t→links[0] = f0;
  t→links[1] = f1;
  t→links[2] = f2;
  t→links[3] = f3;
  t→h = max(f0→h, f1→h);
  flagp = (flag ∧ is_in_K(f0) ∧ is_in_K(f1) ∧ is_in_K(f2) ∧ is_in_K(f3));
  t→type = (flagp) ? 7 : 3;
  if (f0→links[3] ≡ Λ) f0→links[3] = t;
  else if (f0→links[4] ≡ Λ) f0→links[4] = t;
  else abort_message("a␣triangle␣can␣be␣in␣only␣two␣tetrahedra");
  if (f1→links[3] ≡ Λ) f1→links[3] = t;
  else if (f1→links[4] ≡ Λ) f1→links[4] = t;
  else abort_message("a␣triangle␣can␣be␣in␣only␣two␣tetrahedra-1");
  if (f2→links[3] ≡ Λ) f2→links[3] = t;
  else if (f2→links[4] ≡ Λ) f2→links[4] = t;
  else abort_message("a␣triangle␣can␣be␣in␣only␣two␣tetrahedra-2");
  if (f3→links[3] ≡ Λ) f3→links[3] = t;
  else if (f3→links[4] ≡ Λ) f3→links[4] = t;
  else abort_message("a␣triangle␣can␣be␣in␣only␣two␣tetrahedra-3");
  if (flag > 0 ∧ flagp ≡ 0) abort_message("faces␣of␣tetrahedron␣in␣K␣must␣be␣in␣K");
  return t;
}
```

121. ⟨ make sure *vlp* entries are all in *vl* 121 ⟩ ≡

```
for (i = 0; i < 3; i++) {
  if (vlp[i] ≠ vl[0] ∧ vlp[i] ≠ vl[1] ∧ vlp[i] ≠ vl[2] ∧ vlp[i] ≠ vl[3])
    abort_message("new␣tetrahedron␣has␣more␣than␣four␣vertices");
}
```

This code is used in section 120.

122. ⟨ output simplex pairings 122 ⟩ ≡

```
{
  FILE *df;
  vertex *v;
  edge *e;
  triangle *f;
  tetrahedron *t;
  df = fopen(argv[2], "w");
  if (df ≡ Λ) abort_message("Cannot␣open␣output␣file");
  for (i = 0; i < 4; i++)      /* output number if i simplices and critical i simplices */
  {
    fprintf(df, "%d␣%d\n", num_simp[i], n[i]);
  }
  for (i = 0, v = vertexlist; i < num_simp[0]; i++, v++) {
    if (¬is_in_K(v)) fprintf(df, "x%d\n", i);
  }
  fprintf(df, "z\n");
  for (i = 0, e = elist; i < num_simp[1]; i++, e++) {
    if (¬is_in_K(e)) fprintf(df, "x\n");
    else if (is_critical(e)) fprintf(df, "c\n");
    else if (is_paired_up(e)) fprintf(df, "a%d\n", triangle_id(r12(e)));
    else fprintf(df, "b%d\n", vertex_id(r10(e)));
  }
  fprintf(df, "z\n");
  for (i = 0, f = flist; i < num_simp[2]; i++, f++) {
    if (¬is_in_K(f)) {
      if (is_in_K(get_edge(f, 0)) ∧ is_in_K(get_edge(f, 1)) ∧ is_in_K(get_edge(f, 2)))
        fprintf(df, "x%d\n", i);
    }
  }
  fprintf(df, "z\n");
  for (i = 0, t = tlist; i < num_simp[3]; i++, t++) {
    if (¬is_in_K(t)) fprintf(df, "x\n");
    else if (is_critical(t)) fprintf(df, "c\n");
    else fprintf(df, "b%d\n", triangle_id(r32(t)));
  }
  fclose(df);
}
```

123. ⟨ Subroutines for constructing complexes 117 ⟩ +≡

```
void read_in_complex(FILE *df)
{
  long i, n[4];
  int res;
  ⟨ read in vertices 124 ⟩
  ⟨ read in edges 125 ⟩
  ⟨ read in triangles 126 ⟩
  ⟨ read in tetrahedra 127 ⟩
}
```

124. ⟨ read in vertices 124 ⟩ ≡

```
{
  int val;
  int vertex_in_K, c;
  res = fscanf(df, "%ld\n", n);
  if (res ≠ 1) abort_message("bad number of vertices");
  num_simp[0] = n[0];
  vertexlist = (vertex *) malloc(n[0] * sizeof(vertex));
  if (vertexlist ≡ Λ) abort_message("Out of memory");
  for (i = 0; i < n[0]; i++) {
    c = getc(df);
    if (c ≡ 'x') vertex_in_K = 0;
    else {
      vertex_in_K = 1;
      ungetc(c, df);
    }
    res = fscanf(df, "%ld\n", &val);
    if (res ≠ 1) abort_message("bad vertex");
    new_vertex(i, val, vertex_in_K);
  }
}
```

This code is used in section 123.

125. ⟨ read in edges 125 ⟩ ≡

```
{
  long ids[2];
  int edge_in_K, c;
  res = fscanf(df, "%ld\n", n + 1);
  if (res ≠ 1) abort_message("bad number of edges");
  num_simp[1] = n[1];
  elist = (edge *) malloc(n[1] * sizeof(edge));
  if (elist ≡ Λ) abort_message("Out of memory");
  for (i = 0; i < n[1]; i++) {
    c = getc(df);
    if (c ≡ 'x') edge_in_K = 0;
    else {
      edge_in_K = −1;
      ungetc(c, df);
    }
    res = fscanf(df, "%ld %ld\n", ids, ids + 1);
    if (res ≠ 2) abort_message("bad edge");
    new_edge(i, ids, edge_in_K);
  }
}
```

This code is used in section 123.

126. ⟨ read in triangles 126 ⟩ ≡

```
{
  long ids[3];
  int face_in_K, c;
  res = fscanf(df, "%ld\n", n + 2);
  if (res ≠ 1) abort_message("bad number of triangles");
  num_simp[2] = n[2];
  flist = (triangle *) malloc(n[2] * sizeof(triangle));
  if (flist ≡ Λ) abort_message("Out of memory");
  for (i = 0; i < n[2]; i++) {
    c = getc(df);
    if (c ≡ 'x') face_in_K = 0;
    else {
      face_in_K = −1;
      ungetc(c, df);
    }
    res = fscanf(df, "%ld %ld %ld\n", ids, ids + 1, ids + 2);
    if (res ≠ 3) abort_message("bad triangle");
    new_triangle(i, ids, face_in_K);
  }
}
```

This code is used in section 123.

127. ⟨ read in tetrahedra 127 ⟩ ≡

```
{
  long ids[4];
  int tetrahedron_in_K, c;
  res = fscanf(df, "%ld\n", n + 3);
  if (res ≠ 1) abort_message("bad number of tetrahedra");
  num_simp[3] = n[3];
  tlist = (tetrahedron *) malloc(n[3] * sizeof(tetrahedron));
  if (tlist ≡ Λ) abort_message("Out of memory");
  for (i = 0; i < n[3]; i++) {
    c = getc(df);
    if (c ≡ 'x') tetrahedron_in_K = 0;
    else {
      tetrahedron_in_K = −1;
      ungetc(c, df);
    }
    res = fscanf(df, "%ld %ld %ld %ld\n", ids, ids + 1, ids + 2, ids + 3);
    if (res ≠ 4) abort_message("bad tetrahedron");
    new_tetrahedron(i, ids, tetrahedron_in_K);
  }
}
```

This code is used in section 123.

128. ⟨ MorseExtract.h 61 ⟩ +≡

```
#ifndef in_MorseExtract
  extern list *crit[4];
  extern long num_simp[4];      /* number of simplices */
  extern vertex *vertexlist;
  extern edge *elist;
  extern triangle *flist;
  extern tetrahedron *tlist;
#endif
  void Extract(int p);
  void ExtractCancel1(int p);
  void ExtractCancel2(int p);
  void ExtractCancel3(int p);
  void read_in_complex(FILE *df);
  void *plist_read(list *l);
  void get_triangle_vertices(triangle *t, vertex *vlist[3]);
  void get_tetrahedron_vertices(tetrahedron *t, vertex *vlist[4]);
  vertex *FindGrad01(vertex *u, int m);
  tetrahedron *FindGrad23(tetrahedron *tau, int m);
  long find_all_grad12_paths(triangle *sigma, olist *edges, int flags);
  void list_initialize(list **l, unsigned int sz);
  void list_abandon(list **l);
  void *list_push(list *l, void *q);
  void list_pop(list *l, void *q);
  void *list_read(list *l);
  void plist_push(list *l, void *q);
  void *plist_pop(list *l);
  void *plist_read(list *l);
  void olist_initialize(olist **l, int sz);
  void olist_abandon(olist **l);
  void olist_clear(olist *l);
  long olist_min(olist *l, void *p);
  long olist_add(olist *l, long m, void *p);
  long olist_find_add(olist *l, long m, void *p, int *flag);
  void abort_message(char *s);
```

⟨ make *bests_in_run* or *bests* critical and update *bests* 15 ⟩ Used in sections 14 and 17.
⟨ make e critical unless it is the best candidate so far to pair with v 13 ⟩ Used in sections 12 and 17.
⟨ mark all edges on critical paths to this critical edge 66 ⟩ Used in section 65.
⟨ output simplex pairings 122 ⟩
⟨ point r the b-th entry and copy p to it 99 ⟩ Used in sections 97 and 98.
⟨ pop the next item e from *edges_todo* which we haven't processed yet 6 ⟩ Used in section 5.
⟨ prune *edges* 65 ⟩ Used in section 62.
⟨ prune *triangles* 75 ⟩ Used in section 72.
⟨ put eligible edges of t on *graph*, *crits*, and *to_do* 55 ⟩ Used in section 53.
⟨ put gradient path to e on *grad_path* 58 ⟩ Used in section 56.
⟨ put e on the list *c1* and put $v[i]$ on the list *c0* 29 ⟩ Used in section 28.
⟨ put s on *c2* and $t[0]$ and $t[1]$ on *c3* 37 ⟩ Used in section 36.
⟨ read in edges 125 ⟩ Used in section 123.
⟨ read in tetrahedra 127 ⟩ Used in section 123.
⟨ read in triangles 126 ⟩ Used in section 123.
⟨ read in vertices 124 ⟩ Used in section 123.
⟨ replace *bestm* by m if there's just one path to it 57 ⟩ Used in section 56.
⟨ replace *bests_in_run* by s 16 ⟩ Used in sections 14 and 17.
⟨ see if all incoming faces are counted 68 ⟩ Used in section 67.
⟨ see if we can still cancel with some other tetrahedron and replace on *c2* 40 ⟩ Used in section 35.
⟨ see if we can still cancel with some other vertex and replace s on *c1* 32 ⟩ Used in section 27.
⟨ see if s is in lower Star(e) and set *min_vertex* and *in_lower_Star_of_e* accordingly 10 ⟩ Used in sections 9 and 14.
⟨ set *flag* and reset *is_lower_Star_empty* if e is in lower Star(v) 7 ⟩ Used in section 5.
⟨ update $r.c[i]$ to point to a critical tetrahedron 41 ⟩ Used in section 40.
⟨ update $s.c[i]$ to point to a critical vertex 33 ⟩ Used in section 32.
⟨ xextract s and t 14 ⟩ Used in section 8.
⟨ xlast extract of s and t 17 ⟩ Used in section 8.

MORSEEXTRACT