Uniformity Testing

Testing for uniformity in multidimensional data
with the MST-based test of Smith and Jain

Ben Pfaff

The author may be contacted as `blp@gnu.org` on the Internet, or write to:

Ben Pfaff
12167 Airport
DeWitt, MI 48820
UNITED STATES

Table of Contents

1 Introduction

This report presents a library for comparing the distribution of multidimensional data against a uniform distribution, using the minimum spanning tree-based approach described in [Smith 1984]. The complete source code for the library is included, along with discussion of the rationale behind its structure. Finally, its performance is analyzed, and possibilities for further refinements are noted.

The audience for this document is twofold. First, it is intended as a class semester report. For this purpose, the first part of the next chapter (see Chapter 2 [Uniformity Testing], page 5) and the final chapter (see Chapter 6 [Discussion], page 41) are probably the most interesting parts. A secondary purpose for this document is to present the source code for the library as a literate program, as described below. For this reason, some algorithms, such as those for maintaining a binary heap, are described in more depth than would otherwise be necessary.

1.1 How to Read a TexiWEB

The code in this report is written in ANSI/ISO C89 using TexiWEB, a literate programming system. Literate programming is a philosophy that regards software as a form of literature. The ideas behind literate programming have been around for a long time, but the term itself was invented by famous computer scientist Donald Knuth in 1984, who wrote two of his most famous programs (TeX and METAFONT) with a literate programming system of his own design. That system, called `WEB`, inspired the form and much of the syntax of TexiWEB.

A TexiWEB document is a C program that has been cut into sections, rearranged, and annotated, with the goal to make the program as a whole as comprehensible as possible to a reader who starts at the beginning and reads the entire program in order. To this end, each section of a TexiWEB program is assigned both a number and a name. Section numbers are assigned sequentially, starting from 1 with the first section, and they are used for cross-references between sections. Section names are English words or phrases assigned by the TexiWEB program's author to describe the function of the corresponding code.

Here's a sample TexiWEB section:

19. ⟨ Clear hash table entries 19 ⟩ ≡
for ($i = 0$; $i < hash \rightarrow m$; i++)
    $hash \rightarrow entry[i]$ = `NULL`;

You should notice how the first line gives the section's number, followed by its name and again its number within angle brackets. If you know C, you'll also notice that the C operator -> has been replaced by the nicer-looking arrow $\rightarrow$. TexiWEB makes an attempt to "prettify" C in a few ways like this. The table below lists most of these substitutions:

->	becomes	$\rightarrow$
0x12ab	becomes	`0x12ab`
0377	becomes	*0377*
1.2e34	becomes	$1.2{\cdot}10^{34}$

In addition, + and − are written as superscripts when used to indicate sign, as in $^{-}5$ and $^{+}10$.

In TexiWEB, C's reserved words are shown like this: **int**, **struct**, **while**, and so on. Types defined with **typedef** or with **struct**, **union**, and **enum** tags are also shown this way. Identifiers in all capital letters (often names of macros) are shown like this: `BUFSIZ`, `EOF`, `ERANGE`, and so on. Other identifiers are shown like this: *getc*, *argv*, *strlen*, and so on.

Sometimes it is desirable to talk about mathematical expressions, rather than C expressions. When this is done, mathematical operators ($\leq$, $\geq$) instead of C operators (<=, >=) are used. In particular, equality is indicated with $\equiv$ instead of $=$ in order to minimize potential confusion.

Code segments often contain references to other code segments, shown as a section name and number within angle brackets. These act something like macros, in that they stand for the corresponding replacement text. For instance, consider the following segment:

15. ⟨ Initialize hash table 15 ⟩ $\equiv$
hash→*m* = 13;
⟨ Clear hash table entries 19 ⟩

This means that the code for 'Clear hash table entries' should be inserted as part of 'Initialize hash table'. Since the name of a section explains what it does, it's often unnecessary to know anything more. But if you do want more detail, the section number 19 in ⟨ Clear hash table entries 19 ⟩ can easily be used to find the full text and annotations for 'Clear hash table entries'. In addition, at the bottom of section 19 you will find a note reading 'This code is included in segment 15.', making it easy to move back to section 15 that includes it.

It is possible to have multiple sections with the same name. When a name that corresponds to multiple sections is referenced, code from all the sections with that name is substituted, in order of appearance. For instance, the following shows another line of code in 'Initialize hash table':

16. ⟨ Initialize hash table 15 ⟩ $+\equiv$
hash→*n* = 0;

When multiple sections have the same name, those sections end with a note listing the numbers of all other same-named sections. In this case, the note would read 'See also segment 15.'.

When a TexiWEB program is converted to C, conversion conceptually begins from sections whose names are file names; e.g., ⟨ '`foo.c`' 37 ⟩. Within these sections, all of the section references are expanded, then any references within those sections are expanded, and so on. When expansion is complete, the specified files are written out.

Finally, another useful resource in reading a TexiWEB is the index, which contains an entry for the points of declaration of every section name, function, type, structure, union, global variable, and macro. Declarations within functions are not indexed.

(This section is adapted from "Reading the Code," [Pfaff 2000]. It was inspired by "How to read a `WEB`," [Knuth 1992].)

1.2 Coding Style

The source code here complies to the requirements imposed by ANSI/ISO C89 and ISO/IEC C99. Only features present in C89 are used, but the code is forward-compatible with C99.

Most of the GNU Coding Standards are followed. Indentation style is an exception: in print, to save vertical space, K&R indentation style is used instead of GNU style. The code here also conforms to Ben Pfaff's published personal coding standards.

References: [ISO 1990]; [ISO 1999]; [FSF 2001], section "Writing C"; [Pfaff 2001].

1.3 License

The library code itself is under a liberal BSD-style license:

1. ⟨BSD License 1⟩ ≡

```
/* Copyright (c) 2001 Ben Pfaff <pfaffben@msu.edu>. All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are
   met:

   - Redistributions of source code must retain the above copyright
   notice, this list of conditions, and the following disclaimer.

   - Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions, and the following disclaimer in
   the documentation and/or other materials provided with the
   distribution.

   There is NO WARRANTY, not even for MERCHANTABILITY or FITNESS FOR A
   PARTICULAR PURPOSE. */
```

This code is included in segments 3, 9, 26, 34, 35, 50, 68, 70, and 72.

TexiWEB and some of the library's test code is derived from GNU libavl. This code falls under the GNU General Public License:

2. ⟨GNU GPL 2⟩ ≡

```
/* Copyright (C) 2001 Free Software Foundation, Inc.

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful, but
   WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
   General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
   02111-1307, USA.

   The author may be contacted at <pfaffben@msu.edu> on the Internet,
   or as Ben Pfaff, 12167 Airport Rd, DeWitt MI 48820, USA through
   more mundane means. */
```

2 Uniformity Testing

The main function of the library presented in this report is to compare the distribution of points in a multidimensional space against a uniform distribution. The library takes as input a set p of n points in d dimensions each and returns as output a single real number that signifies how uniformly the points are distributed in d-space.

The algorithm used for uniformity testing is based on a minimum spanning tree. It can be outlined as follows:

1. Generate a set q of n points uniformly distributed over the approximate convex hull of p.
2. Find a minimum spanning tree (MST) for the set of points containing both p and q.
3. Calculate test statistic t, based on the MST results, as well as its expected value e and variance var given that p is uniformly distributed.
4. Compute the normal significance s of the difference between t and e.

The smaller the value of s determined in the final step, the more likely it is that the set p is not representative of a uniform distribution. Typically a small threshold value such as 0.05 or 0.02 is compared to s in order to make the final decision on uniformity.

This chapter presents the external interface to the uniformity testing routines as well as their high-level implementation.

References: [Smith 1984], pp. 76–77.

2.1 External Interface: '`uniformity.h`'

The entire external interface to the uniformity testing routines is in '`uniformity.h`'. All externally visible identifiers in this file and others are prefixed by *unf_* to avoid polluting global namespace. This is the only header file that needs to be included to use this library.

Here is the header file's high-level outline:

3. ⟨ '`uniformity.h`' 3 ⟩ ≡
⟨ BSD License 1 ⟩
#ifndef `UNIFORMITY_H`
#define `UNIFORMITY_H`
#include ⟨ stddef.h ⟩
⟨ Test options 6 ⟩
⟨ High-level uniformity test routine prototype 4 ⟩
⟨ Low-level uniformity test routines prototypes 5 ⟩
#endif /* uniformity.h */

Most of the time, only one function in this file, *unf_test*(), will be of interest. It performs all the steps needed for a MST-based uniformity test, using a straightforward interface. The function takes a set of points p and returns a value representing how uniformly the points are distributed. It has the following parameters:

struct unf_options **options*
: A set of configuration options, which may be set to `NULL` if no special configuration is desired.

const double ∗P
: The set of points to test, a total of $n * d$ **double**s. The first d **double**s make up the first point, the second d **double**s make up the second point, and so on.

int *n*
: The number of points in *p*.

int *d*
: The number of dimensions in the points in *p*.

unf_test() returns a number between 0 and 1, where 0 is very non-uniform and 1 is completely uniform.

4. ⟨ High-level uniformity test routine prototype 4 ⟩ ≡
/∗ High-level routine. ∗/
double *unf_test* (**const struct unf_options** ∗, **const double** ∗, **int**, **int**);

This code is included in segment 3.

2.1.1 Low-Level Routines

Three low-level routines are also provided for use by external programs:

unf_inside_hull()
: Tests whether a provided point is within the approximate convex hull of a set of points.

unf_run_mst()
: Finds the MST of a set of points and calculates statistics based on the results.

unf_calc_results()
: Calculates the significance of test results.

These functions will be discussed in more detail and implemented later in this chapter. Here are their prototypes:

5. ⟨ Low-level uniformity test routines prototypes 5 ⟩ ≡
/∗ Low-level routines. ∗/
int *unf_inside_hull* (**const double** ∗, **int**, **int**, **const double** ∗, **double** ∗);
int *unf_run_mst* (**struct unf_mem** ∗, **struct unf_mst** ∗, **int** ∗, **int** ∗, **const double** ∗, **int**, **int**);
double *unf_calc_results* (**int**, **int**, **int**);

This code is included in segment 3.

2.1.2 Options

Occasionally, it may be desirable to fine-tune the operation of *unf_test*(). This possibility is supported by the "hooks" provided by **struct unf_options**, defined as follows:

6. ⟨ Test options 6 ⟩ ≡
/∗ Options. ∗/
/∗ A set of options for *unf_test*(). ∗/
struct unf_options {
    struct unf_mem ∗*unf_mem*; /∗ Memory allocator. ∗/
    struct unf_rng ∗*unf_rng*; /∗ Random number generator. ∗/
    struct unf_set ∗*unf_set*; /∗ Point set. ∗/

```
    struct unf_mst *unf_mst; /* Minimum spanning tree. */
};
```

See also segments 7, 8, 25, 67, 69, and 71. This code is included in segment 3.

To use this structure, declare an instance of it, initialize its members to their defaults, then set as desired any members that should not take their default values. This technique provides for the best forward-compatibility should a new version of this library be released with additional options.

Two methods are provided to initialize members of **struct unf_options** to their defaults. At declaration time, macro `UNF_DEFAULTS` may be used as an initializer. Alternatively, a previously declared option structure may be initialized with *unf_init_options*():

7. ⟨ Test options 6 ⟩ +≡

```
/* Default options. */
#define UNF_DEFAULTS {NULL, NULL, NULL, NULL}
void unf_init_options (struct unf_options *);
```

See also segments 6, 8, 25, 67, 69, and 71. This code is included in segment 3.

Each of the members of **struct unf_options** points to an object-oriented "class" structure. These classes are used during uniformity testing for various tasks: memory allocation, random number generation, point selection, and minimum spanning tree calculation. The defaults are generally reasonable, but these hooks are provided to allow for substitution of better alternatives. Later chapters will discuss these classes in detail. For now, it is sufficient to list the implementations provided here:

8. ⟨ Test options 6 ⟩ +≡

```
/* Provided class implementations. */
extern struct unf_mem unf_mem_malloc;
extern struct unf_set unf_set_rectangular;
extern struct unf_rng unf_rng_mt;
extern struct unf_rng unf_rng_system;
extern struct unf_mst unf_mst_prim_binary;
extern struct unf_mst unf_mst_prim_fibonacci;
```

See also segments 6, 7, 25, 67, 69, and 71. This code is included in segment 3.

2.2 Implementation: '`uniformity.c`'

The functions declared in '`uniformity.h`' are implemented in '`uniformity.c`', which begins as follows:

9. ⟨ '`uniformity.c`' 9 ⟩ ≡

```
⟨ BSD License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ stdio.h ⟩
#include ⟨ math.h ⟩
#include ⟨ stdlib.h ⟩
#include "uniformity.h"
```

See also segments 10, 15, 16, 17, 22, and 24.

2.2.1 High-Level Test Function

Here is the outline of the unf_test() function:

```
10. ⟨‘uniformity.c’ 9⟩ +≡
/* Tests for uniformity of p, a set of n points in d dimensions.
   p is an array of n * d doubles, where the first point is stored
   in the first d elements, the second in the next d elements, and
   so on.
   The contents in user_options, if non-NULL, are used to control
   the testing procedure.
   Returns a confidence level between 0 and 1, 1 indicating that the
   set is perfectly uniform and 0 indicating that the set is perfectly
   clustered. Returns ⁻1 in the event of a memory allocation
   error. */
double unf_test (const struct unf_options *user_options, const double *p, int n, int d) {
    struct unf_options options; /* Local copy of options. */
    double *r; /* Stores p plus generated points. */
    assert (p != NULL && n > 0 && d > 0);
    ⟨Set up options for uniformity test 11⟩
    ⟨Allocate memory for uniformity test 12⟩
    ⟨Generate points for uniformity test 13⟩
    ⟨Run MST, calculate results, and return 14⟩
}
```

See also segments 9, 15, 16, 17, 22, and 24.

The first step is to set up the *user_options* argument. If it is NULL, we need to replace it entirely by the defaults. Otherwise, we make a copy and check member by member for nulls:

```
11. ⟨Set up options for uniformity test 11⟩ ≡
{
    static const struct unf_options default_options = {
        &unf_mem_malloc, &unf_rng_system, &unf_set_rectangular, &unf_mst_prim_binary,
    };
    if (user_options == NULL) options = default_options;
    else {
        options = *user_options;
        if (options.unf_mem == NULL) options.unf_mem = default_options.unf_mem;
        if (options.unf_rng == NULL) options.unf_rng = default_options.unf_rng;
        if (options.unf_set == NULL) options.unf_set = default_options.unf_set;
        if (options.unf_mst == NULL) options.unf_mst = default_options.unf_mst;
    }
}
```

This code is included in segment 10.

The second step is to allocate memory, using the memory allocator in *options.unf_mem*. We need storage for $(2 * n)$ points of d dimensions for storing the points from p and an equal number of generated points, plus space for 2 more such points for temporary storage.

In order to save on calls for memory allocation, we consolidate all this temporary storage into a single block and allocate it all at once.

12. ⟨ Allocate memory for uniformity test 12 ⟩ ≡

```
r = options.unf_mem→unf_alloc (sizeof *r * ((2 + n * 2) * d));
if (r == NULL)
    return ⁻1.0;
```

This code is included in segment 10.

The third step is to generate the points in r on which to perform the MST. The first n of these points are copied from p. It would be faster here to avoid the copy, passing instead a pair of arrays to the MST calculation routine, but it would both complicate and slow down the MST routine itself, so such a change would probably not be worthwhile.

The remaining n of the points are generated randomly within the approximate convex hull of p. This is done using a rejection procedure: points are picked randomly from a conveniently shaped volume that contains the convex hull (for instance, a hyper-rectangle or hypersphere), then points that do not fall within the approximate convex hull are discarded.

The routine for testing whether a point falls within the approximate convex hull of p, *unf_inside_hull*(), uses the storage allocated earlier for an additional pair of points. Notice that this is one place where being able to use C99 features would be handy. Variable Length Arrays (VLAs) from C99 would allow these additional points to be declared as local variables with no need to pre-allocate and pass them down, improving modularity.

13. ⟨ Generate points for uniformity test 13 ⟩ ≡

```
{
    double *const tmp = r + (n * 2) * d;
    struct unf_a_set *s;
    int i;

    s = options.unf_set→unf_create (options.unf_mem, p, n, d);
    if (s == NULL) {
        options.unf_mem→unf_free (r);
        return ⁻1.0;
    }
    memcpy (r, p, sizeof *r * n * d);
    for (i = 0; i < n; ) {
        double *y = r + (n + i) * d;
        options.unf_set→unf_random (s, options.unf_rng, y);
        if (unf_inside_hull (p, n, d, y, tmp))
            i++;
    }
    options.unf_set→unf_discard (s);
}
```

This code is included in segment 10.

Finally, we run the MST and calculate and return the results:

14. ⟨ Run MST, calculate results, and return 14 ⟩ ≡

```
{
    int c, t;
    int okay;
    okay = unf_run_mst (options.unf_mem, options.unf_mst, &c, &t, r, n, d);
    options.unf_mem→unf_free (r);
    return okay ? unf_calc_results (c, t, n) : −1;
}
```

This code is included in segment 10.

2.2.2 Convex Hull Insideness Testing Function

Function *unf_inside_hull*() determines whether a given point y is within the approximate convex hull of a set of points p, using the algorithm specified. This algorithm first calculates the vector

$$\widehat{n}^* = \frac{1}{n}\sum_{i=1}^{n} \frac{p[i] - y}{(\|p[i] - y\|^2)^{d+1}} \ ,$$

where, as usual, n is the number of points and d the number of dimensions. This vector is an estimate of n^*, a vector for which the dot product $n^* \cdot (p[i] - y) > 0$ is true for all $i = 1, 2, \ldots, n$, which is true if and only if n^* is within the convex hull of p. The test calculates this dot product for all values of i and reports that y is within p's convex hull only if the result is always positive.

The first part of the implementation is a helper function to calculate $\widehat{n}^*$. This function, *calc_n_star_est*(), iterates over the values of i. For each i, it measures the magnitude of the vector difference $p[i] - y$ and calculates a scalar *factor* representing the denominator in the equation above. The vector difference, times *factor*, is added to the running sum for $\widehat{n}^*$. Finally, the $1/n$ factor is applied, and the function returns.

15. ⟨'uniformity.c' 9⟩ +≡

```
/* Calculates vector n_star_est at point y within points p.
   tmp is used for temporary storage. Its contents are destroyed.
   There are d elements in each of n_star, tmp, and y, and n * d elements in p. */
static void calc_n_star_est (double *n_star_est, double *tmp,
                             const double *p, int n, int d, const double *y) {
    int i;
    assert (n_star_est != NULL && tmp != NULL && p != NULL
            && n > 0 && d > 0 && y != NULL);
    for (i = 0; i < d; i++)
        n_star_est[i] = 0.0;
    for (i = 0; i < n; i++) {
        double length = 0.0;
        double factor;
        int j;
        for (j = 0; j < d; j++) {
            double diff = tmp[j] = p[j] − y[j];
            length += diff * diff;
        }
        factor = 1.0 / pow (length, d + 1.0);
        for (j = 0; j < d; j++)
```

```
                $n\_star\_est[j]$ += $tmp[j] * factor$;
            $p$ += $d$;
        }
        for ($i$ = 0; $i < d$; $i$++)
            $n\_star\_est[i]$ /= $n$;
}
```

See also segments 9, 10, 16, 17, 22, and 24.

The *unf_inside_hull*() function itself is simple. It calls *calc_n_star_est*() to calculate *n_star_est*, then calculates the dot product *dot* of *n_star_est* with the vector difference $p[i] - y$ for each possible value of i. If *dot* is positive every time, then y is within the approximate convex hull of p; otherwise, it is not.

16. ⟨'uniformity.c' 9⟩ +≡

```
/* Returns nonzero only if point y is within the (approximate) convex
   hull of the n points in array p.
   Each point has d dimensions, and tmp must point to a modifiable
   scratch array with room for 2 * d doubles. */
int unf_inside_hull (const double *p, int n, int d, const double *y, double *tmp) {
    double *n_star_est;
    int i;
    assert (p != NULL && n > 0 && d > 0 && y != NULL && tmp != NULL);
    n_star_est = tmp + d;
    calc_n_star_est (n_star_est, tmp, p, n, d, y);
    for (i = 0; i < n; i++) {
        double dot;
        int j;
        dot = 0.0;
        for (j = 0; j < d; j++)
            dot += (*p++ - y[j]) * n_star_est[j];
        if (dot <= 0.0)
            return 1;
    }
    return 0;
}
```

See also segments 9, 10, 15, 17, 22, and 24.

References: [Smith 1984], pp. 75–76.

2.2.3 MST Test Run Function

The *unf_run_mst*() function actually runs the MST of the pooled original and generated points, then generates the test statistics c and t based on the results. It works like this:

17. ⟨'uniformity.c' 9⟩ +≡

```
/* Runs the MST-based test for uniformity on the 2 * n points in r, of d dimensions each.
   The first n of the points should be the points to test, the last
   n the randomly generated points within the approximate convex
   hull of those points.
   *c and *t receive on output the C and T statistics.
   Returns nonzero if successful, zero if a memory allocation error occurred. */
int unf_run_mst (struct unf_mem *mem_class, struct unf_mst *mst_class,
                 int *c, int *t, const double *r, int n, int d) {
   unf_mst_result *mst;
   int i;

   assert (mem_class != NULL && mst_class != NULL && c != NULL && t != NULL
         && r != NULL && n > 0 && d > 0);

   ⟨ Find MST of pooled points 18 ⟩
   ⟨ Calculate T statistic 19 ⟩
   ⟨ Calculate C statistic 20 ⟩
   ⟨ Free memory and return 21 ⟩
}
```

See also segments 9, 10, 15, 16, 22, and 24.

We delegate the computation of the MST to a hook function. The function takes a memory allocator class and the set of points as its arguments. It returns a pointer to $2 * n - 1$ pairs of **int**s, where each pair designates the indexes of two elements of r connected within the MST.

18. ⟨ Find MST of pooled points 18 ⟩ ≡

```
mst = mst_class→unf_run (mem_class, r, 2 * n, d);
if (mst == NULL)
   return 0;
```

This code is included in segment 17.

The T statistic is the number of edges in the MST that join a point in the original set of points p with a point in the set of randomly generated points. The original points are the first n in r, the generated ones are the remainder, so this is simple.

19. ⟨ Calculate T statistic 19 ⟩ ≡

```
*t = 0;
for (i = 0; i < 2 * n − 1; i++)
   if ((mst[i][0] < n) != (mst[i][1] < n))
      (*t)++;
```

This code is included in segment 17.

The C statistic is the number of pairs of edges in the MST that share a node in common.

20. ⟨ Calculate C statistic 20 ⟩ ≡

```
*c = 0;
for (i = 0; i < 2 * n − 1; i++) {
   int j;

   for (j = i + 1; j < 2 * n − 1; j++)
      if (mst[i][0] == mst[j][0] || mst[i][0] == mst[j][1]
          || mst[i][1] == mst[j][0] || mst[i][1] == mst[j][1])
         (*c)++;
}
```

This code is included in segment 17.

Finally, we finish up by freeing the memory allocated for the MST and returning a successful value.

21. ⟨ Free memory and return 21 ⟩ ≡
mem_class→*unf_free* (*mst*);
return 1;

This code is included in segment 17.

2.2.4 Results Calculation Function

The final step in uniformity testing is to analyze the results based on the MST from the previous step, using c and t computed there. Qualitatively, if the test data are uniformly distributed, then we expect there to be about as many edges joining points within the two groups of points (the test set and the generated set) as edges between the two groups. But if the test data are clustered, then we expect most edges to join points within a group, with a few edges joining the two groups.

This notion is quantifiable. The expected or mean value $E[t]$ of t, given that the data are uniformly distributed, is n, the number of test points, with variance

$$\mathrm{Var}[t|c] = \frac{n}{2n-1}\left(n - 1 - \frac{c-2n+2}{2n-3}\right),$$

and the distribution of t conditioned on c is asymptotically normal in n. We treat t as a normal variate and convert it to a z-score using the formula

$$z = \frac{t - E[t]}{\sqrt{\mathrm{Var}[t|c]}}.$$

The significance s is then found using the cumulative distribution function $P(z)$ of the standard normal distribution $Z(z)$,

$$s = P(z) = \int_{-\infty}^{z} Z(x)\,dx, \quad Z(z) = \frac{1}{\sqrt{2\pi}}\,e^{-x^2/2}.$$

The significance is returned to the caller, which can then apply its own criteria to make the final decision about uniformity. Typically, the caller classifies as uniform samples with $s > s_0$ and as non-uniform samples with $s \le s_0$, where s_0 is a small threshold value such as 0.02 or 0.05.

The implementation is straightforward. To find the results, calculate the mean and variance of t conditioned on c, then report the significance of the resulting z-score:

22. ⟨ `'uniformity.c'` 9 ⟩ +≡
⟨ Significance of a normal variate 23 ⟩

```
/* Returns uniformity significance level between 0 and 1.
   c and t are the values returned by unf_run_mst(), n the size of the test set. */
double unf_calc_results (int c, int t, int n) {
    double mean = n;
    double var = ((n / (2.0 * n − 1.0))
            * (n − 1.0 − (c − 2.0 * n + 2.0) / (2.0 * n − 3.0)));
    double z = (t − mean) / sqrt (var);
    return normal_sig (z);
}
```

See also segments 9, 10, 15, 16, 17, and 24.

The code above makes use of a routine *normal_sig*() to compute the significance. The significance P(z) of a standard normal variate $z \geq 0$ can be calculated using the formula

$$P(z) = 1 - Z(z)[a_1 t + a_2 t^2 + a_3 t^3] + \epsilon(x), \quad t = 1/(1 + px)$$

where a_1, a_2, and a_3 are constants and $\epsilon(x) < 10^{-5}$ is an error term. For $z < 0$, we can use the identity $P(x) = 1 - P(-x)$. The implementation in C is as follows:

23. ⟨ Significance of a normal variate 23 ⟩ ≡

```
/* Returns the significance of normal variate x. */
static double normal_sig (double x) {
    const double a1 = 0.4361836;
    const double a2 = ⁻0.1201676;
    const double a3 = 0.9372980;
    const double p = 0.33267;
    const double pi = 3.14159265358979323846;
    double y = fabs (x);
    double t = 1.0 / (1.0 + p * y);
    double t2 = t * t;
    double t3 = t2 * t;
    double z = 1.0 / sqrt (2.0 * pi) / exp (0.5 * y * y);
    double s = z * (a1 * t + a2 * t2 + a3 * t3);
    return x >= 0 ? 1.0 − s : s;
}
```

This code is included in segment 22.

References: [Smith 1984], pp. 76; [Abramowitz 1972], pp. 931–932; [Zwillinger 1996], pp. 583–584.

2.2.5 Options Initialization Function

24. ⟨ 'uniformity.c' 9 ⟩ +≡

```
/* Initializes options to the defaults, as an alternative to using
   unf_defaults as an initializer. */
void unf_init_options (struct unf_options *options) {
    assert (options != NULL);
    options→unf_mem = NULL;
    options→unf_rng = NULL;
    options→unf_set = NULL;
    options→unf_mst = NULL;
}
```

See also segments 9, 10, 15, 16, 17, and 22.

3 Minimum Spanning Tree Computation

Computing the MST of the set of pooled points is the most interesting part of the implementation of the uniformity test. It is also the part that offers the most possibilities. Several methods for computing a minimum spanning tree are suitable. The following table lists the ones considered, along with their asymptotic time requirements for a complete graph with n vertices:

Method	Best/Avg. Time	Worst Time
Kruskal's algorithm	$O(n^2 \log n)$	$O(n^2 \log n)$
Prim's algorithm with binary heap	$O(n^2 \log n)$	$O(n^2 \log n)$
Prim's algorithm with Fibonacci heap	$O(n^2)$	$O(n^2)$
Bentley's single-fragment algorithm	$O(n \log n)$	$O(n^2 \log n)$
Bentley's multifragment algorithm	$O(n \log n)$	(unknown)

Each of these algorithms has its own advantages and disadvantages in this context. Kruskal's algorithm is simple to implement, but slow. Prim's algorithm requires a priority queue. If a binary heap is used for the priority queue, then it has the same asymptotic speed as Kruskal's algorithm. Substituting a Fibonacci heap speeds up processing considerably. Bentley's single-fragment algorithm is faster than any other algorithm in the average case, but in the worst case it degrades to worse than Prim's using a Fibonacci heap. Unfortunately, this worst case occurs when the points are clustered, one of the situations of interest to us. Bentley's multifragment algorithm may remedy this problem, but its worst-case time bound is in fact unknown.

Prim's algorithm was chosen for implementation because it has the best worst-case behavior of any of the algorithms. Both binary heap and Fibonacci heap variants are implemented. The object-oriented architecture used allows different algorithms, such as Bentley's, to be substituted later or even chosen dynamically at runtime.

References: [Fredman 1987]; [Cormen 1990], chapters 21 and 24; [Bentley 1978].

3.1 Class Structure

The "class" structure that allows alternative MST implementations to be used is **struct unf_mst**. Its only member is pointer to function *unf_run*. This function takes as arguments a memory allocator class, a pointer to a set of points, and counts of the points and the number of dimensions that they contain. It returns a pointer to pairs of vertex indexes indicating the edges in the MST.

25. ⟨ Test options 6 ⟩ +≡

```
typedef int unf_mst_result[2];

struct unf_mst {
    unf_mst_result *(*unf_run) (struct unf_mem *, const double *, int, int);
};
```

See also segments 6, 7, 8, 67, 69, and 71. This code is included in segment 3.

The Prim's algorithm MST implemented here uses an additional class structure **struct pq_class**, which represents a particular type of priority queue. This structure will be presented later, along with the priority queue implementations themselves.

3.2 Prim's Algorithm

Prim's algorithm for computing a minimum spanning tree is implemented by '`mst-prim.c`'. It begins as follows:

26. ⟨ '`mst-prim.c`' 26 ⟩ ≡
```
⟨ BSD License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ float.h ⟩
#include ⟨ stdio.h ⟩
#include "pq.h"
#include "uniformity.h"
```
See also segments 27, 28, and 33.

The first piece of code is a helper function to calculate the Euclidean distance between two points:

27. ⟨ '`mst-prim.c`' 26 ⟩ +≡
```
/* Calculates and returns the distance between points a and b, each in d dimensions. */
static double calc_distance (const double *a, const double *b, int d) {
    double sum = 0.0;
    assert (a != NULL && b != NULL && d >= 0);
    while (d--) {
        double diff = *a++ - *b++;
        sum += diff * diff;
    }
    return sum;
}
```
See also segments 26, 28, and 33.

The MST function itself is next. Its outline looks like this:

28. ⟨ '`mst-prim.c`' 26 ⟩ +≡
```
/* Calculates the MST of the n d-dimensional points at p.
   Uses mem for memory allocation
   and an instance of pq_class as a priority queue. */
static unf_mst_result *calc_mst (struct pq_class *pq_class,
                                 struct unf_mem *mem, const double *p, int n, int d) {
    struct pq *pq;
    unf_mst_result *mst;
    int *edges;
    assert (pq_class != NULL && mem != NULL && p != NULL && n > 0 && d > 0);
    ⟨ Memory allocation for Prim MST 29 ⟩
    ⟨ Execute Prim MST 30 ⟩
    ⟨ Record results of Prim MST 31 ⟩
    ⟨Clean up and return from Prim MST 32⟩
}
```

See also segments 26, 27, and 33.

The first step is memory allocation. We need to create a priority queue, temporary space used for storing edges, and space in which to return the result to the user:

29. ⟨Memory allocation for Prim MST 29⟩ ≡

```
pq = pq_class→create (mem, n);
mst = mem→unf_alloc (sizeof *mst * (n − 1));
edges = mem→unf_alloc (sizeof *edges * n);
if (pq == NULL || mst == NULL || edges == NULL) {
    if (pq != NULL)
        pq_class→discard (pq);
    mem→unf_free (mst);
    mem→unf_free (edges);
    return NULL;
}
```

This code is included in segment 28.

The actual computation of the MST is the second step. When this step is complete, array *edges*[] implicitly stores the minimum spanning tree: for each i in 1. . .n, there is an edge between vertices i and *edges*[i]. The first element, *edges*[0], is not used.

30. ⟨Execute Prim MST 30⟩ ≡

```
pq_class→decrease_key (pq, 0, 0);
while (pq_class→count (pq) > 0) {
    int u, v;
    u = pq_class→extract_min (pq);
    for (v = 0; v < n; v++)
        if (u != v) {
            double key = pq_class→key (pq, v);
            if (key != −DBL_MAX) {
                double cost = calc_distance (p + d * u, p + d * v, d);
                if (cost < key) {
                    edges[v] = u;
                    pq_class→decrease_key (pq, v, cost);
                }
            }
        }
}
```

This code is included in segment 28.

After executing the MST, we translate the *edges*[] array into the form expected by the caller.

31. ⟨Record results of Prim MST 31⟩ ≡

```
{
    int i;
    for (i = 1; i < n; i++) {
        mst[i − 1][0] = i;
        mst[i − 1][1] = edges[i];
    }
}
```

This code is included in segment 28.

Finally, we free temporary data and return to the caller.

32. ⟨ Clean up and return from Prim MST 32 ⟩ ≡

```
pq_class→discard (pq);
mem→unf_free (edges);
return mst;
```

This code is included in segment 28.

References: [Cormen 1990], section 24.2.

3.3 Class Implementations

The final task is to declare class structures for the two MSTs defined here. This requires a wrapper function for each MST, because *calc_mst*() requires one more argument than does **struct unf_mst**'s *unf_run* member. In all, we have the following:

33. ⟨ `mst-prim.c` 26 ⟩ +≡

```
/* Prim's Algorithm MST with binary heap priority queue. */
static unf_mst_result *calc_mst_prim_bin_heap (struct unf_mem *mem, const double *p, int n, int d)
{
    extern struct pq_class unf_pq_bin_heap;
    return calc_mst (&unf_pq_bin_heap, mem, p, n, d);
}
struct unf_mst unf_mst_prim_binary = { calc_mst_prim_bin_heap, };
/* Prim's Algorithm MST with Fibonacci heap priority queue. */
static unf_mst_result *calc_mst_prim_fib_heap (struct unf_mem *mem, const double *p, int n, int d)
{
    extern struct pq_class unf_pq_fib_heap;
    return calc_mst (&unf_pq_fib_heap, mem, p, n, d);
}
struct unf_mst unf_mst_prim_fibonacci = { calc_mst_prim_fib_heap, };
```

See also segments 26, 27, and 28.

4 Priority Queues

Prim's algorithm for computing a minimum spanning tree requires the use of a priority queue to determine the order of insertion of edges into the tree. This chapter presents two different priority queues for this purpose. The first is based on simple binary heaps, the second on more complicated and asymptotically faster Fibonacci heaps.

4.1 Class Structure

In order to make it easy to switch among implementatations, the priority queue is defined in ⟨ pq.h 34 ⟩ as a "class" structure of function pointers. Notice that because ⟨ pq.h 34 ⟩ is not intended to be included by programs using the uniformity-testing library, only by ⟨ mst-prim.c 26 ⟩, it does not restrict itself to the *unf_* namespace:

34. ⟨ 'pq.h' 34 ⟩ ≡

```
⟨ BSD License 1 ⟩
#ifndef PQ_H
#define PQ_H 1
struct unf_mem;
struct pq_class {
    struct pq *(*create) (struct unf_mem *mem, int n);
    void (*discard) (struct pq *pq);
    int (*extract_min) (struct pq *pq);
    void (*decrease_key) (struct pq *pq, int vertex, double key);
    int (*count) (const struct pq *pq);
    double (*key) (const struct pq *pq, int vertex);
    void (*verify) (const struct pq *pq);
    void (*dump) (const struct pq *pq);
};
#endif /* pq.h */
```

The abstract model behind **struct pq_class** is a first-in-smallest-out priority queue. The items in the queue have two parts: an integer "vertex" number between 0 and $n - 1$, where n is the maximum capacity of the queue, and a "key" stored as a floating-point number. The items are ordered based on the key field. An instance of **struct** *pq_class* must implement these abstract operations:

struct pq **create* (**struct unf_mem** **mem*, **int** *n*);

Creates and returns a new priority queue containing n items initially, all of which initially have key $+\infty$. Any needed memory allocation for the priority queue must be done with the functions in *mem*. Returns a null pointer if the priority queue cannot be created.

void *discard* (**struct pq** **pq*);

Destroys priority queue *pq* and frees any associated memory.

int *extract_min* (**struct pq** **pq*);

Deletes the item within *pq* having minimum key and returns its vertex. If *pq* is empty, behavior is undefined.

void *decrease_key* (**struct pq** *pq, **int** *vertex*, **double** *key*);
Finds the item in *pq* with vertex *vertex* and changes its key to *key*. Behavior is undefined if *pq* does not contain an item with vertex *vertex* or if *key* is greater than that item's current key.

int *count* (**const struct pq** *pq);
Returns the number of items remaining in *pq*. This will always be less than or equal to *n* passed to *create*().

double *key* (**const struct pq** *pq, **int** *vertex*);
Returns the key of the item in *pq* with vertex *vertex*. Returns ⁻`DBL_MAX` if there is no vertex *vertex* in *pq*. Behavior is undefined if *vertex* is not in the valid range.

void *verify* (**const struct pq** *pq);
(For debugging purposes.) Verifies that *pq*'s state is internally consistent.

void *dump* (**const struct pq** *pq);
(For debugging purposes.) Prints a human-readable representation of *pq*'s internal state to *stdout*.

4.2 Binary Heap

File ⟨ pq-bin-heap.c 35 ⟩ implements binary heaps. It begins like this:

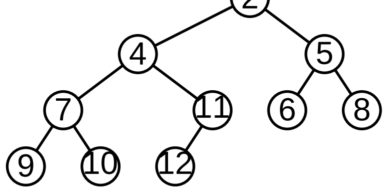
35. ⟨ '`pq-bin-heap.c`' 35 ⟩ ≡
⟨ BSD License 1 ⟩
#**include** ⟨ assert.h ⟩
#**include** ⟨ float.h ⟩
#**include** ⟨ stdio.h ⟩
#**include** "pq.h"
#**include** "uniformity.h"

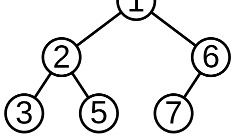
See also segments 36, 37, 38, 39, 40, 44, 45, 46, 47, 48, and 49.

The class of binary heaps is a subset of the class of binary trees. A binary tree is a binary heap if:

1. For every node except the root, the key value in its parent node is less or equal than its own key value (the "heap property"); *and*
2. The binary tree is "nearly complete," meaning that every level in the tree is filled, except possibly for the bottommost, which must be filled in left-to-right order without gaps.

It is not necessary, but here we will also assume that key values in heap nodes are distinct. Here are two examples of heaps, with the key values shown inside the node circles:





A heap can be conveniently represented as an array, with no need for pointers, by writing down its elements from left to right, top to bottom. The two heaps pictured above would have the following representations as arrays:

(2, 4, 5, 7, 11, 6, 8, 9, 10, 12)
(1, 2, 6, 3, 5, 7)

The first element of the array, at index 1, is the heap's root. To get from the i'th element to its left child, simply multiply i by 2; to get to its right child, multiply by 2 and add 1; to get to its parent, divide by 2 and discard any remainder. If the result is between 1 and the number of nodes in the heap, then it is the index of the desired node; otherwise, there is no such node in the heap. We can easily implement these "movement" operations in C:

36. ⟨'pq-bin-heap.c' 35⟩ +≡

```
/* Heap movement operations. */
#define parent(i) ((i) / 2) /* Get node i's parent. */
#define left(i) ((i) * 2) /* Get node i's left child. */
#define right(i) (left (i) + 1) /* Get node i's right child. */
```

See also segments 35, 37, 38, 39, 40, 44, 45, 46, 47, 48, and 49.

Our binary heap priority queue structure based on these ideas looks like this:

37. ⟨'pq-bin-heap.c' 35⟩ +≡

```
/* Binary heap priority queue. */
struct pq {
    struct unf_mem *mem; /* Memory allocator. */
    int n; /* Number of items in tree. */
    int m; /* Maximum number of items in tree. */

    /* Heap contents. */
    double *key; /* Keys. */
    int *vertex; /* Satellite information (vertex number). */

    /* Reverse index. */
    int *index; /* Heap index by vertex number; 0 = not present. */
};
```

See also segments 35, 36, 38, 39, 40, 44, 45, 46, 47, 48, and 49.

Most of this structure should be self-explanatory. Member *key* points to the heap itself; *vertex* is a parallel array that contains the companion vertex values. Both of these arrays have an extra unused element at the beginning, which allows for 1-based indexing and simplifies the movement macros above.[1] Member *index* is used to store the indexes into *key* and *vertex* that contain particular vertex values. This array is necessary for fast lookup of particular vertices, since there is no quick way to search a heap.

The following sections implement the abstract operations from **struct** *pq_class* on binary heaps.

References: [Cormen 1990], section 7.1; [Knuth 1973], section 5.2.3.

[1] On many architectures some pointer trickery could be used instead, but this is technically undefined according to [ISO 1990], section 6.3.6. See [Summit 1999], question 6.17, for more information.

4.2.1 Creation

Creation and initialization of the heap is straightforward:

38. ⟨‘pq-bin-heap.c’ 35⟩ +≡
```
static void pq_discard (struct pq *pq);
static struct pq *pq_create (struct unf_mem *mem, int n) {
    struct pq *pq;
    int i;
    /* Allocate memory for heap itself. */
    assert (n >= 0);
    pq = mem→unf_alloc (sizeof *pq);
    if (pq == NULL)
        return NULL;
    /* Allocate memory for heap's members. */
    pq→mem = mem;
    pq→key = mem→unf_alloc (sizeof *pq→key * (n + 1));
    pq→vertex = mem→unf_alloc (sizeof *pq→vertex * (n + 1));
    pq→index = mem→unf_alloc (sizeof *pq→index * (n + 1));
    if (pq→key == NULL || pq→vertex == NULL || pq→index == NULL) {
        pq_discard (pq);
        return NULL;
    }
    /* Initialize heap. */
    pq→n = pq→m = n;
    for (i = 1; i <= n; i++) {
        pq→key[i] = DBL_MAX;
        pq→vertex[i] = i;
        pq→index[i] = i;
    }
    return pq;
}
```
See also segments 35, 36, 37, 39, 40, 44, 45, 46, 47, 48, and 49.

4.2.2 Destruction

Destruction simply discards the memory that was allocated at creation time:

39. ⟨‘pq-bin-heap.c’ 35⟩ +≡
```
static void pq_discard (struct pq *pq) {
    assert (pq != NULL);
    pq→mem→unf_free (pq→key);
    pq→mem→unf_free (pq→vertex);
    pq→mem→unf_free (pq→index);
    pq→mem→unf_free (pq);
}
```
See also segments 35, 36, 37, 38, 40, 44, 45, 46, 47, 48, and 49.

4.2.3 Minimum Value Extraction

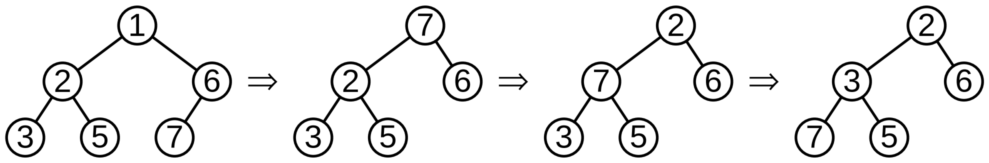
Extracting the minimum value from a binary heap can be divided into three steps.

40. ⟨ 'pq-bin-heap.c' 35 ⟩ +≡
```
static int
pq_extract_min (struct pq *pq)
{
    int min;
    assert (pq != NULL && pq->n > 0);
    ⟨ Save minimum value of binary heap 41 ⟩
    ⟨ Delete root from binary heap 42 ⟩
    ⟨ Restore binary heap property 43 ⟩
    return min - 1;
}
```
See also segments 35, 36, 37, 38, 39, 44, 45, 46, 47, 48, and 49.

The minimum value in a binary heap is the root, so finding the minimum value is easy:

41. ⟨ Save minimum value of binary heap 41 ⟩ ≡
```
min = pq->vertex[1];
```
This code is included in segment 40.

The interesting part is deleting the root node and rearranging the resultant binary tree so that it is still a heap. The successful approach is to move the last (n'th) node in the heap into the root position, then move down the tree swapping nodes until the heap property is satisfied. Here's an example of the effects of heap deletion, with the original heap at the left and the heap after at the right and the intermediate steps in between:



We start by deleting the minimum node from the reverse index, moving the last node to the root, and decrementing the tree's node count:

42. ⟨ Delete root from binary heap 42 ⟩ ≡
```
assert (min >= 1 && min <= pq->m);
pq->index[min] = 0;
pq->vertex[1] = pq->vertex[pq->n];
pq->key[1] = pq->key[pq->n];
pq->index[pq->vertex[1]] = 1;
pq->n--;
```
This code is included in segment 40.

To restore the heap property, we start from the root and move down the tree, swapping the current node with the smallest of its children. If the current node is smaller than its children, we can stop:

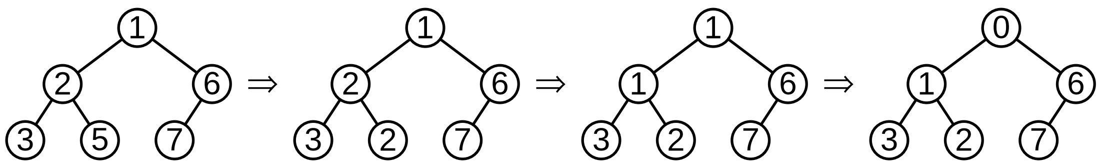
43. ⟨ Restore binary heap property 43 ⟩ ≡
```
{
    int i = 1;
    for (;;) {
        /* Find index of smallest of i or its children as smallest. */
        int l = left (i);
        int r = right (i);
        int smallest = i;
        if (l <= pq→n && pq→key[l] < pq→key[smallest])
            smallest = l;
        if (r <= pq→n && pq→key[r] < pq→key[smallest])
            smallest = r;
        if (smallest == i)
            break;
        /* Swap nodes i and smallest. */
        {
            int t_vertex = pq→vertex[i];
            double t_key = pq→key[i];
            pq→vertex[i] = pq→vertex[smallest];
            pq→key[i] = pq→key[smallest];
            pq→vertex[smallest] = t_vertex;
            pq→key[smallest] = t_key;
            pq→index[pq→vertex[i]] = i;
            pq→index[pq→vertex[smallest]] = smallest;
        }
        i = smallest;
    }
}
```

This code is included in segment 40.

References: [Cormen 1990], section 7.5; [Knuth 1973], section 5.2.3.

4.2.4 Decreasing Keys

To decrease the key value of a heap item, we cause it to “bubble up” the tree until its parent has a smaller value than it does or it reaches the root. We can always do this because we replace smaller values by larger values, so the proper relationship between the keys replaced and their children's keys is maintained. In the example below, the key of the node labeled 5 is decreased to 0:



No further discussion should be necessary:

44. ⟨‘pq-bin-heap.c’ 35⟩ +≡

```
static void pq_decrease_key (struct pq *pq, int vertex, double key) {
    int i, p;
    assert (pq != NULL);
    assert (vertex >= 0 && vertex < pq→m);
    assert (pq→index[vertex + 1] != 0);
    assert (key <= pq→key[pq→index[vertex + 1]]);
    for (i = pq→index[vertex + 1]; i > 1; i = p) {
        p = parent (i);
        if (pq→key[p] < key)
            break;
        pq→key[i] = pq→key[p];
        pq→vertex[i] = pq→vertex[p];
        pq→index[pq→vertex[i]] = i;
    }
    pq→key[i] = key;
    pq→vertex[i] = vertex + 1;
    pq→index[pq→vertex[i]] = i;
}
```

See also segments 35, 36, 37, 38, 39, 40, 45, 46, 47, 48, and 49.

References: [Cormen 1990], section 7.5, exercise 4.

4.2.5 Count

The count function just returns **struct pq**'s *n* value:

45. ⟨'pq-bin-heap.c' 35⟩ +≡

```
static int pq_count (const struct pq *pq) {
    assert (pq != NULL);
    return pq→n;
}
```

See also segments 35, 36, 37, 38, 39, 40, 44, 46, 47, 48, and 49.

4.2.6 Key Lookup

The *index* array is designed for key lookup, making this function easy:

46. ⟨'pq-bin-heap.c' 35⟩ +≡

```
static double pq_key (const struct pq *pq, int vertex) {
    int i;
    assert (pq != NULL && vertex >= 0 && vertex < pq→m);
    i = pq→index[vertex + 1];
    return i != 0 ? pq→key[i] : −DBL_MAX;
}
```

See also segments 35, 36, 37, 38, 39, 40, 44, 45, 47, 48, and 49.

4.2.7 Verification

The verification function checks that the heap property is satisfied and that the *index* array is properly set up:

47. ⟨'pq-bin-heap.c' 35⟩ +≡

```
static void pq_verify (const struct pq *pq) {
    int i;
    int error = 0;

    assert (pq != NULL);
    assert (pq→m > 0);
    assert (pq→n <= pq→m);

    for (i = 2; i <= pq→n; i++) {
        int parent = parent (i);
        if (pq→key[parent] > pq→key[i]) {
            printf ("Heap␣property␣for␣%d:%d␣violated:␣%g␣>␣%g\n",
                    parent, i, pq→key[parent], pq→key[i]);
            error = 1;
        }
    }

    for (i = 1; i <= pq→n; i++)
        if (pq→index[i] != 0 && pq→vertex[pq→index[i]] != i) {
            printf ("index␣for␣%d␣points␣to␣position␣%d,␣which␣contains␣%d\n",
                    i, pq→index[i], pq→vertex[pq→index[i]]);
            error = 1;
        }

    assert (error == 0);
}
```

See also segments 35, 36, 37, 38, 39, 40, 44, 45, 46, 48, and 49.

4.2.8 Dumping

The dump function prints out an array representation of the binary heap to *stdout*. Vertex numbers and (finite) keys are printed separated by colons.

48. ⟨‘pq-bin-heap.c’ 35⟩ +≡

```
static void pq_dump (const struct pq *pq) {
    int i;

    for (i = 1; i <= pq→n; i++) {
        printf ("%d", pq→vertex[i]);
        if (pq→key[i] != DBL_MAX)
            printf (":%g", pq→key[i]);
        putchar ('␣');
    }
    putchar ('\n');
}
```

See also segments 35, 36, 37, 38, 39, 40, 44, 45, 46, 47, and 49.

4.2.9 Class Implementation

The class implementation of the binary heap is the final declaration in the file:

49. ⟨‘pq-bin-heap.c’ 35⟩ +≡

```
struct pq_class unf_pq_bin_heap = {
    pq_create, pq_discard, pq_extract_min, pq_decrease_key,
    pq_count, pq_key, pq_verify, pq_dump,
};
```

See also segments 35, 36, 37, 38, 39, 40, 44, 45, 46, 47, and 48.

4.3 Fibonacci Heap

File ⟨ pq-fib-heap.c 50 ⟩ implements Fibonacci heaps. It begins this way:

50. ⟨ ‘pq-fib-heap.c’ 50 ⟩ ≡

```
⟨ BSD License 1 ⟩
#include ⟨ assert.h ⟩
#include ⟨ float.h ⟩
#include ⟨ stdio.h ⟩
#include "pq.h"
#include "uniformity.h"
```

See also segments 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, and 66.

A Fibonacci heap is a complicated data structure, quite a bit more involved to explain than a simple binary heap. As a result, only the implementation will be presented here. For a complete description of the principles behind a Fibonacci heap, consult one of the references.

The Fibonacci heap priority queue structure is defined this way:

51. ⟨ ‘pq-fib-heap.c’ 50 ⟩ +≡

```
/* Fibonacci heap priority queue. */
struct pq {
    int n; /* Number of items in heap. */
    int m; /* Maximum number of items in heap. */

    struct unf_mem *mem; /* Memory allocator. */
    struct node *min; /* Minimum node. */
    struct node **index; /* Index by vertex number to get node. */
    struct node **rank; /* Used during extract_min(). */
    struct node *prealloc; /* Memory allocated for nodes. */
};
```

See also segments 50, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, and 66.

struct node has this form:

52. ⟨ ‘pq-fib-heap.c’ 50 ⟩ +≡

```
/* Fibonacci heap node. */
struct node {
    int vertex; /* Vertex value. */
    double key; /* Key value. */

    int rank; /* Number of direct children. */
    int mark; /* 1=marked, 0=unmarked. */

    struct node *parent; /* Parent node, NULL if a root. */
    struct node *child; /* One of our children, NULL if terminal. */
    struct node *next, *prev; /* Next and previous in circular list. */
};
```

See also segments 50, 51, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, and 66.

The last bit before the code itself is a set of prototypes for internal functions:

53. ⟨'pq-fib-heap.c' 50⟩ +≡

```
static void insert (struct pq *pq, int vertex, double key);
static void detach (struct node *node);
static void add_root (struct pq *pq, struct node *root);
static void add_child (struct node *parent, struct node *child);
```

See also segments 50, 51, 52, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, and 66.

References: [Cormen 1990], chapter 21; [Fredman 1987]; [Boyer 1997].

4.3.1 Creation

Creating a Fibonacci tree is a relatively simple matter of allocating memory and initializing all the fields of **struct pq** properly, then inserting all the nodes with initial keys of ∞:

54. ⟨'pq-fib-heap.c' 50⟩ +≡

```
static struct pq *pq_create (struct unf_mem *mem, int n) {
    struct pq *pq;
    int i;

    assert (n >= 0);

    pq = mem->unf_alloc (sizeof *pq);
    if (pq == NULL)
        return NULL;

    pq->mem = mem;
    pq->n = 0;
    pq->m = n;
    pq->min = NULL;
    pq->index = pq->mem->unf_alloc (sizeof *pq->index * n);
    pq->rank = pq->mem->unf_alloc (sizeof *pq->rank * n);
    pq->prealloc = pq->mem->unf_alloc (sizeof *pq->prealloc * n);
    if (pq->index == NULL || pq->prealloc == NULL) {
        mem->unf_free (pq);
        return NULL;
    }
    for (i = 0; i < n; i++) {
        pq->index[i] = NULL;
        pq->rank[i] = NULL;
    }
    for (i = 0; i < n; i++)
        insert (pq, i, DBL_MAX);

    return pq;
}
```

See also segments 50, 51, 52, 53, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, and 66.

The *insert*() helper function for *pq_create*() inserts the specified vertex/key pair into the root list, using the $(n + 1)$th element of *prealloc* to do it:

55. ⟨'pq-fib-heap.c' 50⟩ +≡

```
static void insert (struct pq *pq, int vertex, double key) {
    struct node *node;
    assert (pq != NULL);
    assert (vertex >= 0 && vertex < pq→m);
    assert (pq→index[vertex] == NULL);
    pq→index[vertex] = node = pq→prealloc + pq→n++;
    node→vertex = vertex;
    node→key = key;
    node→rank = 1;
    node→mark = 0;
    node→parent = node→child = NULL;
    add_root (pq, node);
    if (node→key < pq→min→key)
        pq→min = node;
}
```

See also segments 50, 51, 52, 53, 54, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, and 66.

4.3.2 Destruction

Destroying a Fibonacci heap is easy:

56. ⟨'pq-fib-heap.c' 50⟩ +≡

```
static void pq_discard (struct pq *pq) {
    assert (pq != NULL);
    pq→mem→unf_free (pq→index);
    pq→mem→unf_free (pq→prealloc);
    pq→mem→unf_free (pq→rank);
    pq→mem→unf_free (pq);
}
```

See also segments 50, 51, 52, 53, 54, 55, 57, 58, 59, 60, 61, 62, 63, 64, 65, and 66.

Notice that because all of the nodes were allocated at once from the *prealloc* array, there is no need to free each of them separately, nor would it be correct to do so.

4.3.3 Minimum Value Extraction

Extracting the minimum value from a Fibonacci heap is complicated. Consult one of the references for details. Armed with one of those, the comments in the code below should be sufficient explanation:

57. ⟨'pq-fib-heap.c' 50⟩ +≡

```
static int pq_extract_min (struct pq *pq) {
    struct node *min;
    assert (pq != NULL);
assert (pq→n > 0);
/* Grab the minimum value and remove it from the heap. */
min = pq→min;
pq→index[min→vertex] = NULL;
pq→n−−;
pq→min = min→next;
if (pq→min == min)
    pq→min = NULL;
detach (min);
/* Add all of the children of the former minimum as roots. */
for (;;) {
    struct node *child = min→child;
    if (child == NULL)
        break;
    detach (child);
    add_root (pq, child);
}
/* If the heap is now empty, return early. */
if (pq→min == NULL)
    return min→vertex;
/* Go through all the root nodes and put them into rank[],
   linking together the ones with equal rank fields. */
{
    struct node *w = pq→min;
    do {
        struct node *x = w;
        w = w→next;
        if (w == pq→min)
            w = NULL;
        for (;;) {
            struct node *y = pq→rank[x→rank];
            if (y == NULL || y == x)
                break;
            pq→rank[x→rank] = NULL;
            if (x→key > y→key) {
                struct node *t = x;
                x = y;
                y = t;
            }
            if (pq→min == y)
                pq→min = y→next;
            detach (y);
            add_child (x, y);
            y→mark = 0;
        }
```

```
            pq→rank[x→rank] = x;
        } while (w != NULL);
    }
    /* Go through the root nodes and unset the corresponding rank[] elements.
       Pick the next pq→min while we're at it. */
    {
        struct node *x;

        x = pq→min;
        for (;;) {
            pq→rank[x→rank] = NULL;
            x = x→next;
            if (x == pq→min)
                break;

            if (x→key < pq→min→key)
                pq→min = x;
        }
    }

    return min→vertex;
}
```

See also segments 50, 51, 52, 53, 54, 55, 56, 58, 59, 60, 61, 62, 63, 64, 65, and 66.

Function *pq_extract_min*() makes use of three helper functions. The first of these is *detach*(), which removes takes a specified node out of its parent's list of children:

58. ⟨'pq-fib-heap.c' 50⟩ +≡

```
/* Removes node from the heap, making it parentless. */
static void detach (struct node *node) {
    assert (node→next != NULL && node→prev != NULL);
    node→next→prev = node→prev;
    node→prev→next = node→next;

    if (node→parent != NULL) {
        if (node→parent→child == node) {
            node→parent→child = node→next;
            if (node→parent→child == node)
                node→parent→child = NULL;
        }

        node→parent→rank−−;
        node→parent = NULL;
    }
#ifndef NDEBUG
    node→next = node→prev = NULL;
#endif
}
```

See also segments 50, 51, 52, 53, 54, 55, 56, 57, 59, 60, 61, 62, 63, 64, 65, and 66.

The first of these adds a specified node to the priority queue's root list. This node must be parentless, meaning that it should earlier have been detached with *detach*():

59. ⟨'pq-fib-heap.c' 50⟩ +≡
```
/* Adds root as a root of pq.
   pq→min is changed only if it is NULL,
   so the caller is responsible for updating it if it should decrease. */
static void add_root (struct pq *pq, struct node *root) {
    assert (root→parent == NULL);

    if (pq→min == NULL) {
        pq→min = root;
        root→prev = root→next = root;
    } else {
        root→prev = pq→min;
        root→next = pq→min→next;
        pq→min→next→prev = root;
        pq→min→next = root;
    }
}
```
See also segments 50, 51, 52, 53, 54, 55, 56, 57, 58, 60, 61, 62, 63, 64, 65, and 66.

The second helper function adds a node to an arbitrary parent node's child list. Again, *child* must be parentless:

60. ⟨'pq-fib-heap.c' 50⟩ +≡
```
/* Adds child as a child of parent. */
static void add_child (struct node *parent, struct node *child) {
    assert (parent != NULL && child != NULL);
    assert (child→parent == NULL);

    child→parent = parent;
    if (parent→child == NULL) {
        child→next = child→prev = child;
        parent→child = child;
    } else {
        child→prev = parent→child;
        child→next = parent→child→next;
        parent→child→next→prev = child;
        parent→child→next = child;
    }
    parent→rank++;
}
```
See also segments 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 61, 62, 63, 64, 65, and 66.

4.3.4 Decreasing Keys

The function for decreasing keys in a Fibonacci tree is not as complicated as for extracting the minimum key, but it is still nontrivial, mostly due to the need for "cascading cuts" in some circumstances. Again, consult the references for full explanation:

61. ⟨'pq-fib-heap.c' 50⟩ +≡
```
static void pq_decrease_key (struct pq *pq, int vertex, double key) {
    struct node *x, *y;
    assert (pq != NULL);
    assert (vertex >= 0 && vertex < pq→m);
    assert (pq→index[vertex] != NULL);
    assert (key <= pq→index[vertex]→key);
    /* Find the node in question. */
    x = pq→index[vertex];
    x→key = key;
    y = x→parent;
    /* Make cascading cuts as necessary. */
    if (y != NULL && x→key < y→key)
        for (;;) {
            struct node *z;
            assert (x→parent != NULL);
            assert (x→parent == y);
            assert (pq→min != x);
            detach (x);
            add_root (pq, x);
            x→mark = 0;
            z = y→parent;
            if (z == NULL)
                break;
            if (y→mark == 0) {
                y→mark = 1;
                break;
            } else {
                x = y;
                y = z;
            }
        }
    /* Actually decrease the key's value. */
    if (key < pq→min→key)
        pq→min = pq→index[vertex];
}
```

See also segments 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 62, 63, 64, 65, and 66.

4.3.5 Count

The count is stored in **struct pq**, just as for the binary heap implementation:

62. ⟨'pq-fib-heap.c' 50⟩ +≡

```
static int pq_count (const struct pq *pq) {
    assert (pq != NULL);
    return pq→n;
}
```

See also segments 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 63, 64, 65, and 66.

4.3.6 Key Lookup

Looking up a key is just a matter of consulting the index:

63. ⟨ ‘pq-fib-heap.c’ 50 ⟩ +≡

```
static double pq_key (const struct pq *pq, int vertex) {
    struct node *node;
    assert (pq != NULL);
    assert (vertex >= 0 && vertex < pq->m);
    node = pq->index[vertex];
    return node != NULL ? node->key : -DBL_MAX;
}
```

See also segments 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 64, 65, and 66.

4.3.7 Verification

Extensive verification of the Fibonacci heap's structure can be done:

64. ⟨ ‘pq-fib-heap.c’ 50 ⟩ +≡

```
static void verify_recursive (const struct pq *pq, const struct node *first,
                              const struct node *parent, const struct node **tmp) {
    const struct node *iter;
    int count;
    if (first == NULL)
        return;
    /* Check that the forward links are circular. */
    iter = first;
    count = 0;
    do {
        assert (iter->parent == parent);
        assert (count < pq->m);
        tmp[count++] = iter;
        iter = iter->next;
    } while (iter != first);
    /* Check that the backward links are circular,
       and that they are the same ones as when we go forward. */
    iter = first;
    do {
        iter = iter->prev;
        assert (count > 0);
        count--;
        assert (tmp[count] == iter);
    } while (iter != first);
    assert (count == 0);
    /* Perform verification recursively on child nodes. */
    iter = first;
    do {
        verify_recursive (pq, iter→child, iter, tmp);
        iter = iter→next;
    } while (iter != first);
}
static void pq_verify (const struct pq *pq) {
    const struct node **tmp;
    assert (pq != NULL);
    assert (pq→n >= 0 && pq→n <= pq→m);
    tmp = pq→mem→unf_alloc (sizeof *tmp * pq→n);
    if (pq→n != 0 && tmp == NULL) {
        printf ("virtual␣memory␣exhausted\n");
        abort ();
    }
    verify_recursive (pq, pq→min, NULL, tmp);
    pq→mem→unf_free (tmp);
}
```

See also segments 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 65, and 66.

4.3.8 Dumping

We dump out the full multilevel recursive structure of the Fibonacci tree:

65. ⟨'pq-fib-heap.c' 50⟩ +≡

```
static void dump_recursive (const struct node *p) {
    const struct node *q = p;
    putchar ('(');
    do {
        if (q != p)
            fputs (",␣", stdout);
        printf (q→key == DBL_MAX ? "+oo" : "%g", q→key);
        if (q→child != NULL)
            dump_recursive (q→child);
        q = q→next;
    } while (q != p);
    putchar (')');
}
static void pq_dump (const struct pq *pq) {
    dump_recursive (pq→min);
}
```

See also segments 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, and 66.

4.3.9 Class Implementation

The class implementation of the Fibonacci heap is the final declaration in the file:

66. ⟨'pq-fib-heap.c' 50⟩ +≡

```
struct pq_class unf_pq_fib_heap = {
    pq_create, pq_discard, pq_extract_min, pq_decrease_key,
    pq_count, pq_key, pq_verify, pq_dump,
};
```

See also segments 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, and 65.

5 Other Classes

This chapter contains descriptions and implementations of the three classes that have not been treated yet: memory allocation, random number generation, and point set handling.

5.1 Memory Allocation

struct unf_mem provides a "class" structure for memory allocators within the uniformity testing library. It is declared as follows:

67. ⟨ Test options 6 ⟩ +≡

```
/* Memory allocator class. */
struct unf_mem {
      void *(*unf_alloc) (size_t);
      void (*unf_free) (void *);
};
```

See also segments 6, 7, 8, 25, 69, and 71. This code is included in segment 3.

In an implementation, *unf_alloc* must be a function that takes a count of bytes to allocate and returns a pointer to the allocated memory if successful or `NULL` if no memory is available. Companion member *unf_free* must point to a function that frees memory allocated with *unf_alloc*.

File ⟨ mem-std.c 68 ⟩ provides an implementation of the memory allocator class that uses the standard C *malloc*() and *free*() functions:

68. ⟨ 'mem-std.c' 68 ⟩ ≡

```
⟨ BSD License 1 ⟩
#include ⟨ stdlib.h ⟩
#include "uniformity.h"
static void *standard_alloc (size_t n) {
      return malloc (n);
}
static void standard_free (void *p) {
      free (p);
}
struct unf_mem unf_mem_malloc = {standard_alloc, standard_free};
```

5.2 Random Number Generator

The class interface for random number generation contains one important member *unf_get*, which, when called, must return a pseudo-random number in the closed range [0, 1]. An additional member *unf_seed* is provided for seeding the generator for the convenience of calling code, but it is not used by the uniformity testing library itself.

69. ⟨ Test options 6 ⟩ +≡

```
struct unf_rng {
      void (*unf_seed) (unsigned long);
      double (*unf_get) (void);
};
```

See also segments 6, 7, 8, 25, 67, and 71. This code is included in segment 3.

File ⟨ rng-std.c 70 ⟩ uses the standard C pseudo-random number generator (PRNG) *rand*() for random number generation. This is not a good choice on many systems, where *rand*() may have very poor randomness properties.

70. ⟨ ‘`rng-std.c`’ 70 ⟩ ≡

```
⟨ BSD License 1 ⟩
#include ⟨ stdlib.h ⟩
#include "uniformity.h"
static void rng_seed (unsigned long seed) {
    srand (seed);
}
static double rng_get (void) {
    return (double) rand () / RAND_MAX;
}
struct unf_rng unf_rng_system = {rng_seed, rng_get};
```

An additional random number generator is provided in ‘`rng-mt.c`’. It is based on the fast and high-quality Mersenne Twister random number generator.

References: [Matsumoto 1997].

5.3 Point Set

The purpose of the point set class is to represent an infinite set of points that includes the convex hull of a provided finite collection of points, but may include other points as well. For instance, it could be implemented as the smallest hypersphere or hyper-rectangle containing all the provided points. The class is represented as **struct unf_set**:

71. ⟨ Test options 6 ⟩ +≡

```
/* Point set. */
struct unf_set {
    struct unf_a_set *(*unf_create) (struct unf_mem *, const double *, int n, int d);
    void (*unf_discard) (struct unf_a_set *);
    void (*unf_random) (const struct unf_a_set *, struct unf_rng *, double *);
};
```

See also segments 6, 7, 8, 25, 67, and 69. This code is included in segment 3.

The members of **struct unf_set** are abstractly described as follows:

struct unf_a_set **unf_create* (**struct unf_mem** **mem*, **const double** **p*, **int** *n*, **int** *d*)
: Creates and returns a new point set that includes the *n* *d*-dimensional points at *p*. Memory allocation is performed using *mem*. Returns `NULL` if unsuccessful due to lack of memory or for some other reason.

void *unf_discard* (**struct unf_a_set** **set*);
: Destroys point set *set*.

void *unf_random* (**const struct unf_a_set** **set*, **struct unf_rng** **rng*, **double** **p*);
: Selects a random point from a uniform distribution over the set of points represented by *set*. Pseudo-random number generator *rng* is used in selection of

the point. The point is stored into the vector at *p*, which must have room for *d* elements, *d* being the same value passed to *unf_create*().

The implementation provided here in ⟨ set-rect.c 72 ⟩ uses a minimum-size hyper-rectangle to surround the points. The implementation begins with the definition of **struct unf_a_set**:

72. ⟨ 'set-rect.c' 72 ⟩ ≡

```
⟨ BSD License 1 ⟩
#include ⟨ assert.h ⟩
#include "uniformity.h"
/* Hyper-rectangular point set. */
struct unf_a_set {
    struct unf_mem mem; /* Memory allocator. */
    int d; /* Number of dimensions. */
    double *min; /* Minimum values for each of d dimensions. */
    double *max; /* Maximum values for each of d dimensions. */
};
```

See also segments 73, 74, 75, and 76.

To create a set, we allocate memory for it, then iterate through the set of points provided setting up the *min* and *max* arrays:

73. ⟨ 'set-rect.c' 72 ⟩ +≡

```
struct unf_a_set *set_create (struct unf_mem *mem, const double *p, int n, int d) {
    struct unf_a_set *set;
    int i;
    assert (p != NULL && n > 0 && d > 0);
    /* Memory allocation. */
    set = mem→unf_alloc (sizeof *set);
    if (set == NULL)
        return NULL;
    set→mem = *mem;
    set→d = d;
    set→min = mem→unf_alloc (sizeof *set→min * d * 2);
    if (set→min == NULL) {
        mem→unf_free (set→min);
        mem→unf_free (set);
        return NULL;
    }
    set→max = set→min + d;
    /* Construct hyper-rectangle. */
    for (i = 0; i < d; i++)
        set→min[i] = set→max[i] = *p++;
    for (i = 1; i < n; i++) {
        int j;
        for (j = 0; j < d; j++) {
            if (*p < set→min[j])
                set→min[j] = *p;
                else if (*p > set→max[j])
                    set→max[j] = *p;
                p++;
            }
        }
        return set;
}
```

See also segments 72, 74, 75, and 76.

Destroying a set is just a matter of freeing its memory:

74. ⟨'`set-rect.c`' 72⟩ +≡

```
void set_discard (struct unf_a_set *set) {
    assert (set != NULL);
    set→mem.unf_free (set→min);
    set→mem.unf_free (set);
}
```

See also segments 72, 73, 75, and 76.

To generate a random point in a hyper-rectangle, we generate an uniform random variate along each axis independently:

75. ⟨'`set-rect.c`' 72⟩ +≡

```
void set_random (const struct unf_a_set *set, struct unf_rng *rng, double *v) {
    int i;
    for (i = 0; i < set→d; i++)
        v[i] = set→min[i] + rng→unf_get () * (set→max[i] − set→min[i]);
}
```

See also segments 72, 73, 74, and 76.

Finally, we declare the class structure implementation:

76. ⟨'`set-rect.c`' 72⟩ +≡

```
struct unf_set unf_set_rectangular = {
    set_create, set_discard, set_random,
};
```

See also segments 72, 73, 74, and 75.

6 Discussion

This chapter discusses the library for uniformity testing. It begins with a description of the provided programs for testing. Sample results from these programs are then displayed and discussed. Next, implications for the correctness of the library and its components are suggested. The library's performance is analyzed. Finally, some directions for further work are mentioned.

6.1 Programs for Testing

This section briefly discusses the programs for testing the uniformity testing library as a whole or in part. The source code for these programs is not presented due to its length and tedium.

6.1.1 Convex Hull Estimation

The first step in the uniformity testing algorithm is to select a random set of points within the approximate convex hull of the points to be tested. It is worthwhile to test that these points are generated properly.

The `hull-test` program tests the convex hull estimation visually. It picks an initial set of points within a triangular area, then generates another set in their approximate convex hull using *unf_inside_hull*(). It prints a graphical representation of both sets of points to *stdout* in a format suitable for the `jgraph` graphing utility. The user can then judge for himself whether the points were correctly generated.

6.1.2 Minimum Spanning Trees

The code for priority queues for the Prim MST is somewhat complex, and therefore error-prone. A simple test to make sure that the priority queues are probably implemented correctly is helpful. This is what the `pq-test` program does.

When run without command-line arguments, `pq-test` will perform its test 16 times for a priority queue of 128 items and print a success or failure message for each test. To see a list of other options, use the command '`pq-test -h`'.

The minimum spanning tree algorithm as a whole is also worth testing. The `mst-test` program does this. It works similarly to the `hull-test` program, generating a random set of points, calculating their MST, and producing as output a graphical representation of the MST.

6.1.3 Uniformity Tester

Testing the uniformity testing library itself is important, too. The `unf-test` program does this. Each time it is run it generates a set of points randomly within user-specified parameters, then runs a uniformity test on it and reports the results. The user can specify several parameters for the test using command-line arguments:

`-d` *dims*
`--dims=`*dims*
    d, the number of dimensions.

`-c` *count*
`--count=`*count*
: n, the number of points to test.

`-a` *avg*
`--avg=`*avg*
: Optionally, a number of calls to *unf_test*() over which to average s. By default only one call is used per data set.

`-S` *sig*
`--sig=`*sig*
: The minimum value of s that is considered "uniform," .05 by default. Increasing this value will decrease the chance of non-uniform data sets considered uniform and increase the chance of uniform data sets considered non-uniform.

`-D` *dist*
`--dist=`*dist*
: The distribution from which the test points are picked. Argument *dist* may be one of the following:

  `uniform` The uniform distribution on [0, 1].

  `decreasing`
  The distribution of decreasing values such that $x_1 > x_2 > \ldots > x_d$. The first variate x_1 is picked uniformly between 0 and 1, the next variate x_2 between 0 and x_1, and so on. This distribution is so non-uniform that it should essentially *never* be detected as uniform.

  `normal` The multivariate normal distribution with mean at the origin and identity covariance matrix.

`-m` *mst*
`--mst=`*mst*
: Selects the minimum spanning tree algorithm to use, one of:

  `prim-bin` Prim's algorithm with a binary heap priority queue.

  `prim-fib` Prim's algorithm with a Fibonacci heap priority queue.

`-p` *proximity*
`--prox=`*proximity*
: The default behavior is to select all of the test points from a single distribution of the type specified. When this option is specified along with a positive value for *proximity*, the data points are instead divided into two equal-size groups, the means of which are displaced apart a distance of *proximity*.

`-r` *runs*
`--runs=`*runs*
: Perform the specified tests on *runs* sets of data. The default is 16.

`-s` *seed*
`--seed=`*seed*
: Use *seed* as the initial random number seed for the first run. The seed is incremented for each succeeding run. The default is based on the current time.

`-h`
`--help` Outputs a brief help screen and exits.

`-V`
`--version`
Outputs version and licensing information and exits.

These features will be used extensively to produce data for the discussion below.

6.2 Testing Results

The test programs above can be used to analyze the correctness and performance of the uniformity testing library. This section presents results obtained using them along with a discussion of these results' meaning.

6.2.1 Correctness

Correctness of the testing algorithm can be tested both when the sampling window is known and when it is unknown. The ordinary `unf-test` program can be used for the latter; the former can be tested the same way by temporarily commenting out the line

if (*unf_inside_hull* (*p*, *n*, *d*, *y*, *tmp*))

in ⟨ Generate points for uniformity test 13 ⟩ and recompiling.[1]

Tables below show the results of uniformity testing for cells with various dimensions d and point counts n. Each cell is of the form $(R(0.05), R(0.02))$, where $R(s)$ indicates the percentage of uniform distributions wrongly detected as non-uniform at the given level s. Each entry corresponds to 10,000 runs.

6.2.1.1 Known Sampling Window

This table shows the results of uniformity testing for uniform samples when the sampling window is known:

		d		
		2	5	10
	50	(4.3,1.5)	(4.2,1.7)	(4.2,1.6)
n	100	(5.0,1.9)	(5.2,2.0)	(4.6,2.0)
	200	(4.8,2.0)	(4.5,2.1)	(5.0,1.9)

The expected values for R(0.05) and R(0.02) are 5 and 2, respectively. Considering each uniformity test R(s) as a binomial trial with probability of success $p = s$ and selecting a significance level $\alpha = 0.05$, the allowed intervals to accept the hypothesis $p = \widehat{p}$ are $4.57 < R(0.05) < 5.43$ and $1.73 < R(0.02) < 2.27$.

For $n = 50$, few of the values in the table are in the range for acceptance, but for $n = 100$ and $n = 200$ most of them are. This indicates that the algorithm is probably correct. A later section will discuss possible sources of error.

[1] The proper way to do this would be to avoid use of *unf_test*(), using the low-level functions prototyped in ⟨ uniformity.h 3 ⟩ instead. This is not done in `unf-test` because one of its purposes is to ensure that *unf_test*() works.

In the original paper, all of the values are within the range to allow acceptance, but only 100 runs were made per entry, instead of 10,000, presumably because of time constraints. When only 100 runs are made, it is easy to accidentally get results that are all within the acceptable range.

References: [Smith 1984], table I; [Zwillinger 1996], sections 7.9.1 and 7.12.2.

6.2.1.2 Unknown Sampling Window

When the sampling window is unknown, the results are like this:

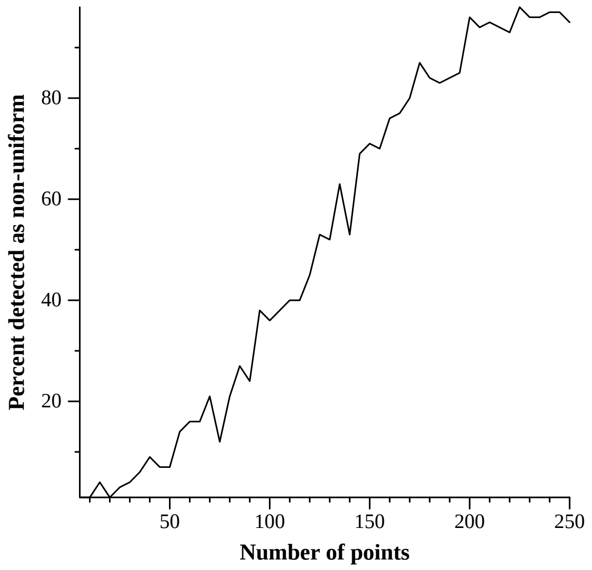
		d		
		2	5	10
	50	(4.2,1.6)	(2.3,0.9)	(0.5,0.1)
n	100	(5.3,1.8)	(3.0,1.0)	(0.7,0.2)
	200	(4.6,1.9)	(3.3,1.1)	(0.5,0.2)

For small d, entries are in the same general range as for the known sampling window data, and as d increases, they become more conservative. This is the expected behavior according to the references.

References: [Smith 1984], table IV.

6.2.2 Normal Distributions

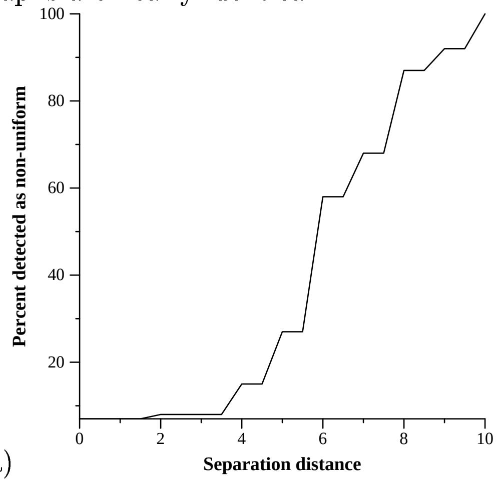
We can run the uniformity test on points selected from a normal distribution. If this is done for several values of n with $d = 5$ and a desired 5% false detection rate, this graph results:



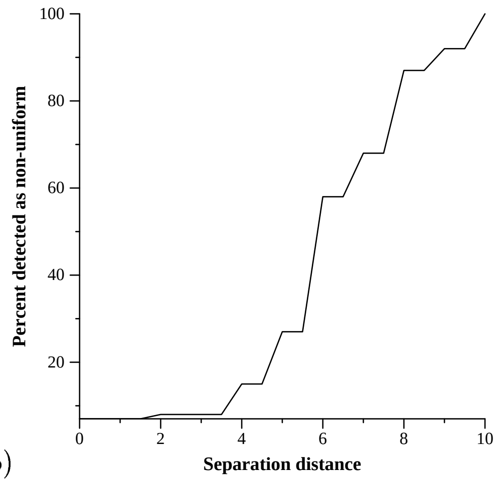
In words, for small n, uniform and normal distributions are almost indistinguishable. As n increases, the distribution is detected more and more often as non-uniform, until the time that $n = 200$ is reached, at which it is essentially always found to be non-uniform.

6.2.3 Testing Separation

An additional interesting test is to divide the test points into two sets and separate their means by increasing distances. The graphs below show the results for (a) normal distributions and (b) uniform distributions, each with a total of $n = 50$ points. Note that the graphs are nearly identical.



(a)



(b)

6.3 Sources of Error

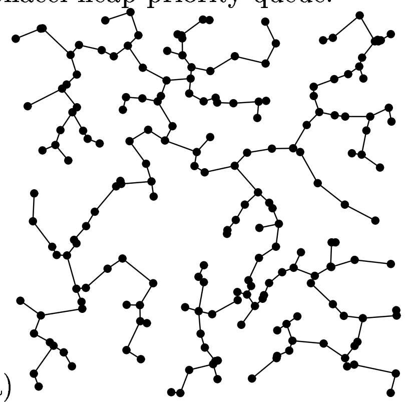
Based on results above, the code implementing the uniformity testing algorithm seems to work properly. This section discusses possible sources of error that were considered during design and implementation but have not already been discussed.

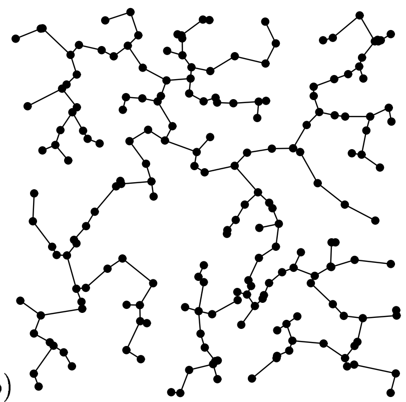
6.3.1 Bugs in MST Implementation

There are two programs to test the minimum spanning tree algorithms: `pq-test` for the priority queues and `mst-test` for the spanning tree itself.

Testing the queue implementations for obvious flaws by running the `pq-test` program with several sets of parameters does not reveal any problems in either priority queue implementation. This does not mean that none exist, but it is a good sign.

Running `mst-test` shows that it produces correct minimum spanning trees on test data. Minimum spanning tree (a) below was produced with the binary heap priority queue, (b) with the Fibonacci heap priority queue:


(a)


(b)

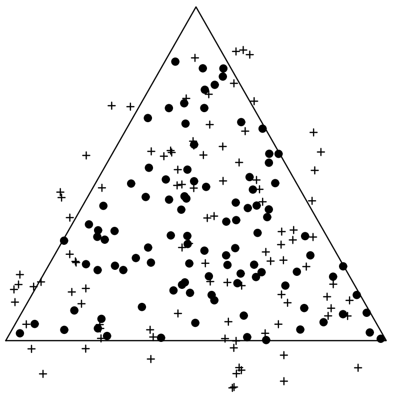
6.3.2 Bugs in Convex Hull Insideness Test

The test whether a point is inside the approximate convex hull of the set of points is suspicious from the beginning because of the way that the equation for the test is written in both the dissertation and the paper describing the algorithm, using a subscript '2' in a confusing way. Presumably a superscript '2' was intended; it was interpreted this way in the implementation. The original form given for this equation was:

$$\widehat{n}^* = N^{-1} \sum_{i=1}^{N} (Xi - Y)/(\|Xi - Y\|_2)^{K+1}.$$

A search of the literature did not turn up use of any similar tests, so no comparisons could be made. Several possible variants of this test were tried, but none produced better results.

If the `hull-test` program is used to visually test the convex hull estimation, the results are disappointing:



The original points are marked with circles, the ones generated within the "approximate convex hull" with crosses. Notice how far away from the triangular border some of the generated points fall. None of generated points in a similar illustration in the original paper fall as far away as these.

References: [Smith 1984], figure 3.

6.3.3 Imprecise Arithmetic

The uniformity testing library uses **double** for all of its floating-point calculations. It is possible that additional precision might increase accuracy in calculations.

To experiment with this idea, all **double** (64-bit IEEE 754) quantities were replaced by **long double** (80-bit IEEE 754) quantities. In addition, sums were performed both using Kahan summation and in least-to-greatest order (by performing a sort operation).

There was no noticeable change in behavior from these increases in precision. In conclusion, for the current data sets, arithmetic precision is not a limiting factor.

References: [Goldberg 1991].

6.4 Performance

The two significant steps in the uniformity test whose performance depend on n and d are the generation of random points and calculation of the minimum spanning tree. The first of these has asymptotic performance $O(n^2)$, the latter $O(n^2 \log n)$ for Prim's algorithm with a binary heap or $O(n^2)$ with a Fibonacci heap.

When `unf-test` is compiled with GCC 2.95.4 prerelease 20010319 using options `-O3 -DNDEBUG` and run on a Pentium II/233 MHz, tests with various n and d using the binary heap priority queue, run 50 times each, take the times shown in the table below, in seconds:

			d	
		2	5	10
	50	0.3	0.4	0.7
n	100	1.2	1.4	2.2
	200	4.3	5.1	8.0
	400	17.2	20.5	30.2
	800	69.1	85.2	116.6

Note that doubling n causes the time to quadruple, almost exactly, so the practical performance of the code for n in these ranges is $O(n^2)$ under these conditions.

When a Fibonacci heap is used instead, the times *increase* slightly:

			d	
		2	5	10
	50	0.4	0.4	0.7
n	100	1.2	1.4	2.3
	200	4.6	5.5	8.2
	400	18.4	21.7	31.1
	800	72.5	86.8	119.6

This effect is presumably due to the increased complexity of a Fibonacci heap. According to [Cormen 1990], chapter 21, this is not unexpected:

> From a practical point of view, however, the constant factors and programming complexity of Fibonacci heaps make them less desirable than ordinary binary (or k-ary) heaps for most applications. Thus, Fibonacci heaps are predominantly of theoretical interest.

6.5 Future Work

Currently the code for the uniformity testing library is very general, because it was written with little idea of how it was going to be used. Specializing it for particular situations may be one task for future work. For instance, some applications may prefer the use of a bounding hypersphere instead of the provided hyper-rectangle implementation.

There is much potential for optimization. Profiling shows that generation and testing of points and the minimum spanning tree computation take roughly equal amounts of time, so improvement in either one would improve performance. A few minutes work on the implementation of the Prim MST in particular might help. Some experimentation shows that a 10% improvement there, by modifying its interface to the priority queue, is fairly simple to achieve, and a larger improvement might not be much more difficult. Replacement of Prim's algorithm by one of Bentley's algorithms might reap much greater benefits.

This document could also use some elaboration for the benefit of less knowledgeable readers. For instance, a better explanation of Prim's algorithm for constructing MST and for Fibonacci heap algorithms would be helpful.

Appendix A References

[Abramowitz 1972]. Abramowitz, M., and I. A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. U. S. Govt. Print. Off., Washington, D.C., 1972.

[Bentley 1978]. Bentley, J. L., and J. H. Friedman, "Fast Algorithms for Constructing Minimal Spanning Trees in Coordinate Spaces," *IEEE Trans. on Comp.* 2 (1984), pp. 97–105.

[Boyer 1997]. Boyer, J., "Algorithm Alley: The Fibonacci Heap," *Dr. Dobb's Journal*, Jan. 1997, `http://www.ddj.com/articles/1997/9701/9701o/9701o.htm?topic=algorithms`.

[Cormen 1990]. Cormen, C. H., C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. McGraw-Hill, 1990. ISBN 0-262-03141-8.

[FSF 2001]. Free Software Foundation, *GNU Coding Standards*, edition 23 Mar 2001. `http://www.gnu.org/prep/standards_toc.html`.

[Goldberg 1991]. Goldberg, D., "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, 23 (1991), pp. 5–48.

[ISO 1990]. International Organization for Standardization, *ANSI/ISO 9899-1990: American National Standard for Programming Languages—C*, 1990. Reprinted in *The Annotated ANSI C Standard*, ISBN 0-07-881952-0.

[ISO 1999]. International Organization for Standardization, *International Standard ISO/IEC 9899:1999(E): Programming Languages—C*, 1999.

[Knuth 1992]. Knuth, D. E., *Literate Programming*, CSLI Lecture Notes Number 27. Center for the Study of Language and Information, Leland Stanford Junior University, 1992. ISBN 0-9370-7380-6.

[Knuth 1973]. Knuth, D. E., *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973. ISBN 0-201-03803-X. 2nd ed.: ISBN 0-201-89685-0, 1998.

[Matsumoto 1997]. Matsumoto, M., and T. Nishimura, "Mersenne Twister" pseudo-random number generator. `http://www.math.keio.ac.jp/~matumoto/emt.html`.

[Pfaff 2000]. Pfaff, B. L., LIBAVL *library for manipulation of binary trees*, version 2.0 ALPHA, edition 13 Dec 2000. `http://www.msu.edu/~pfaffben/avl/`.

[Pfaff 2001]. Pfaff, B. L., *Personal Coding Standards*, version 1.2, edition 5 Jan 2001. `http://www.msu.edu/~pfaffben/writings/`.

[Smith 1984]. Smith, S. P., and A. K. Jain, "Testing for Uniformity in Multidimensional Data," *IEEE Trans. on Pattern Analysis and Machine Intelligence* 6 (1984), pp. 73–81.

[Summit 1999]. Summit, S., *comp.lang.c Answers to Frequently Asked Questions*, version 3.5. `http://www.eskimo.com/~scs/C-faq/top.html`. ISBN 0-201-84519-9.

[Fredman 1987]. Fredman, L. F., and R. E.: Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *J. ACM*, 3 (1987), pp. 596–615.

[Zwillinger 1996]. Zwillinger, D., *CRC Standard Mathematical Tables and Formulae*, 30th ed. CRC Press, Boca Raton, 1996. ISBN 0-8493-2479-3.

Appendix B Index