Probability Value Library[1]

Peter N. Yianilos, NEC Research Institute
Eric Sven Ristad, Princeton University

Research Report CS-TR-472-94

October 1994

Abstract

Hidden Markov models (HMMs) and other time series models assign probabilities to long sequences of events. Avoiding underflow is arguably the central difficulty in calculating the probability of such sequences. This technical report presents an elegant and efficient C library for representing and manipulating probability values without underflow. Use of the library results in simpler code whose execution time compares favorably with traditional numerical methods. Thus, the library provides a strong foundation on which to build large scale stochastic modeling systems. Our abstraction also suggests a natural extension of the IEEE floating point standard, to better support statistical computation.

[1]This report was typeset in LaTeX using the NoWeb literate programming language (Norman Ramsey, norman@bellcore.com). The second author is supported by a Young Investigator Award IRI-9258517 from the National Science Foundation. The probability value library was designed, implemented and documented in Fall 1993.

1 Introduction

The pr_t module provides an abstraction for probability values. An object of type pr_t represents a real number in the closed interval $[0, 1]$. The principal benefit of using this abstraction is to avoid underflow in time series modeling. In these settings, one computes and manipulates probabilities of long state sequences. One quickly reaches the exponent range limit of double representation, as this is typically only ± 300. Use of the pr_t data type instead of traditional numerical methods results in simpler code and competitive execution times.

There are two traditional ways of dealing with underflow in statistical computation: scaling and logarithmic representation. The *scaling* approach keeps the probability values from underflowing by maintaining a scaling factor for each time step. This scaling factor is typically the smallest multiplier required to keep the largest probability value at unity. Calculating the scaling factor and applying it to the vector of probability values requires two to three times the execution time of simple double operations. More significantly, scaling is difficult to implement and even more difficult to validate. It is also inherently nonmodular.

In the *logarithmic representation* approach, probabilities are represented by their logarithms. Multiplication of probabilities is fast because it requires a single fixed point addition, but addition of probabilities is extremely slow because it requires two exponentiations, one floating point addition, and a logarithm operation. For example, a straightforward implementation of addition in the logarithmic approach is more than fifty times slower than standard floating point addition. The excessive cost of addition is often avoided by complex table look-up schemes [1]. These table lookup schemes have limited precision and they place a significant additional burden on the memory system. In order to achieve comparable precision, the logarithmic representation must be implemented with a large lookup table that is unlikely to fit in primary cache. As a result, the computation will be dominated by the high cost of cache misses. The logarithmic method is of less interest today because high-speed floating point operations are widely available.

The pr_t library is a direct and elegant solution to the large exponent range problem. All work necessary to support a sufficiently large exponent range is swept into the routines associated with the pr_t type. As a result, statistical applications are significantly easier to write and maintain, while the overall time/space penalty compares favorably with traditional numerical methods. Our current implementation of pr_t type requires 1.5 times as much space as a double precision floating point number while our arithmetic operations take roughly ten times longer than the standard floating point add/multiply. Thus, our approach is both faster and simpler than the logarithmic approach and considerably easier to use than the scaling approach.

The sole contribution of this work is its utility. We have used the pr_t library to implement a wide range of statistical models, including hidden Markov models, finite growth models, and a bevy of statistical language models. Given the difficulty of establishing the correctness of learning algorithms such as expectation maximization (EM) and Baum-Welch reestimation in complex settings with many thousands of states, a sparse transition graph, and millions of observations, we were glad to trade a modest amount of performance for demonstrable correctness and a significant reduction in coding complexity.

We have also used the pr_t library to more effectively teach statistical pattern recognition methods to undergraduate computer science students. Using the pr_t library, the students in our undergraduate pattern recognition class were able to implement semi-continuous hidden Markov models with full parameter tying. Without the pr_t library, students in the previous year had been limited to only implementing simple discrete hidden Markov models, in large part due to the complexities of implementing and validating scaling.

The remainder of our report is organized as follows. Section 2 provides a detailed specification for the pr_t library. Section 3 concludes with a discussion of related issues, including proposed extensions to the IEEE-754 floating point standard and timing information for our current pr_t implementation on a range of modern unix workstations. The appendix contains a user's guide and a complete implementation of the pr_t library, along with a comprehensive test suite and benchmark program.

2 Specification

The pr_t library consists of a small set of operations along with a some modest requirements on the pr_t data type implementation. The pr_t library provides operations to coerce types, to compare values, and to perform arithmetic. Each operation is available in two forms, one defined on pr_t objects and the other defined on pointers to pr_t objects.

2.1 Primitive Operations

The following primitive operations are defined directly on objects of type pr_t.

Initial probability values are created by the pr_double2pr() and pr_nats2pr() procedures. pr_double2pr() and pr_pr2double() convert between probability values represented as double-precision floating point numbers and those represented as pr_t. pr_nats2pr() and pr_pr2nats() convert between probability values represented as negative log likelihoods in base e (ie., codelengths) and those represented as pr_t.

$\langle$*pr operations* 3a$\rangle\equiv$
```
/* Constructors qua Type Coercion */
pr_t       pr_double2pr(double x);
double     pr_pr2double(pr_t px);

pr_t       pr_nats2pr(double x);
double     pr_pr2nats(pr_t px);
```

This definition is continued in chunks 3–6 and 17b.
This code is used in chunk 18.

For user convenience, we also provide constructors **pr_zero()** and **pr_unity()** for the distinguished probability values of zero and unity, respectively. **pr_epsilon()** returns the smallest possible mantissa increment, with zero exponent.

$\langle$*pr operations* 3a$\rangle+\equiv$
```
/* Distinguished Value Constructors */
pr_t       pr_zero(void);
pr_t       pr_unity(void);
pr_t       pr_epsilon(void);
```

The **pr_is_zero()** and **pr_is_unity()** predicates return **TRUE** iff their arguments are exactly equal to zero and unity, respectively. **pr_is_valid()** returns **TRUE** iff its **px** argument falls in the closed interval $[0-\epsilon, 1+\epsilon]$, for a user-supplied **epsilon** parameter. This predicate is particularly useful in catching incorrect memory references.

$\langle$*pr operations* 3a$\rangle+\equiv$
```
/* Predicates */
BOOLEAN    pr_is_zero(pr_t px);
BOOLEAN    pr_is_unity(pr_t px);
BOOLEAN    pr_is_valid(pr_t px, pr_t epsilon);
```

`pr_compare()` returns a negative quantity if `px` is less than `py`, a positive quantity if `px` is greater than `py`, and zero if they are approximately equal, as determined by the user-supplied `tolerance` argument. `pr_exact_compare()` is equivalent to `pr_compare()`, except that equality must be exact. These two procedures may be implemented as macros. `pr_difference()` returns the absolute value of the difference between its two arguments.

⟨*pr operations* 3a⟩+≡

```
/* Comparison */
int         pr_compare(pr_t px, pr_t py, pr_t tolerance);
int         pr_exact_compare(pr_t px, pr_t py);
pr_t        pr_difference(pr_t px, pr_t py);
```

The following procedures implement the standard binary arithmetic operations of addition, multiplication, and division for objects of type `pr_t`.

⟨*pr operations* 3a⟩+≡

```
/* Arithmetic Operations */
pr_t        pr_add(pr_t px, pr_t py);
pr_t        pr_multiply(pr_t px, pr_t py);
pr_t        pr_divide(pr_t px, pr_t py);
```

The `pr_power()` procedure returns the result of raising the `px` argument to the $\mathtt{n}^{th}$ power. A domain error occurs if $\mathtt{px} = 0$ and $\mathtt{n} \leq 0$. Injudicious use of this procedure can easily give rise to floating point exceptions.

⟨*pr operations* 3a⟩+≡

```
pr_t        pr_power(pr_t px, double n);
```

2.2 Pointer Operations

In addition to the primitive operations specified above, we provide an equivalent set of pointer-based operations, that are passed the memory locations of their arguments and store their return values at a caller-supplied memory location. These operations are provided to support efficient manipulation of probability vectors. They may be implemented as macros.

`pr_compare_ptr()` and `pr_exact_compare_ptr()` are equivalent to the basic `pr_compare()` and `pr_exact_compare_ptr()`, respectively. `pr_difference_ptr()` places the magnitude of the difference between its first and second arguments in the location specified by the third argument, and returns the sign of their difference.

⟨*pr operations* 3a⟩+≡

```
/* Pointer Comparison */
int        pr_compare_ptr(pr_t *px_p, pr_t *py_p, pr_t *tolerance);
int        pr_exact_compare_ptr(pr_t *px_p, pr_t *py_p);
int        pr_difference_ptr(pr_t *px_p, pr_t *py_p, pr_t *pz_p);
```

The functions `pr_add_ptr()`, `pr_multiply_ptr()`, and `pr_divide_ptr()` always read their arguments from locations `px_p` and `py_p` and place their result in location `pz_p`. It is safe to use either `*px_p` or `*py_p` as an accumulator, as in `pr_add_ptr(&a, &b, &b)`.

⟨*pr operations* 3a⟩+≡

```
/* Pointer Arithmetic Operations */
void       pr_add_ptr(pr_t *px_p, pr_t *py_p, pr_t *pz_p);
void       pr_multiply_ptr(pr_t *px_p, pr_t *py_p, pr_t *pz_p);
void       pr_divide_ptr(pr_t *px_p, pr_t *py_p, pr_t *pz_p);
```

`pr_power_ptr()` raises the probability located at address `px_p` to the $\mathtt{n}^{\mathrm{th}}$ power and stores the result in location `pz_p`.

⟨*pr operations* 3a⟩+≡

```
void       pr_power_ptr(pr_t *px_p, double n, pr_t *pz_p);
```

2.3 Input/Output

The following two procedures print the value of a `pr_t` object in the format `<sig>b-<exp>` to remind the reader that the exponent radix is 2 (ie., `b`inary).

$\langle$*pr operations* 3a$\rangle$+$\equiv$

```
/* Input/Output */
void        pr_fprintf(pr_t px, FILE *stream);
void        pr_printf(pr_t px);
```

2.4 Data Type

In our introduction, we stated that probability values lie in the interval $[0, 1]$. In practice a somewhat larger range is required because accumulated round off error may result in probability values that exceed unity. Therefore, implementations of the `pr_t` data type must be prepared to deal with values somewhat larger than unity. Given a vector of probabilities that should sum to unity, we may wish to normalize it by first computing its actual sum, and then reducing or increasing some elements of the vector to bring its sum closer to unity.

In addition, statistical computations often must accumulate the expectations of events, in order to reestimate parameter values using expectation maximization. Accordingly, if the `pr_t` type implementation is able to represent values in the interval $[0,\infty)$, then objects of type `pr_t` may be used to accumulate parameter expectations in the expectation maximization computation. (These are the gamma values of Baum-Welch reestimation.)

A third requirement on the type implementation is that the `pr_t` type should appear as much as possible like the other numerical types in C. In particular, it must be possible to copy `pr_t` objects using the standard `=` assignment operator, to pass `pr_t` objects by value or reference, and to automatically allocate/deallocate space for objects of type `pr_t` on the stack.

3 Conclusion

We conclude by proposing an extension to the IEEE-754 Floating Point standard, to better support statistical computation. We also provide execution timings for our current implementation of the `pr_t` library across a range of modern unix workstations. The appendix contains a user's guide along with all source code.

3.1 Extensions to IEEE Floating Point

It is unfortunate that the IEEE floating point standard does not address the need for an extended exponent range. The *double extended* standard provides for 14 bit exponents but falls far short of the requirements imposed by statistical computation.

We therefore make the following two concrete proposals to amend the IEEE-754 floating point standard:

1. Implement a `balanced double` 64-bit format in which both significand and exponent are allocated 32 bits.

2. Implement a `quad` 128-bit format consisting of 1 sign bit, a *regular/extended* mode bit, 16/32 exponent bits, and, 110/94 significand bits. The mode bit selects one of two formats. An FPU's internal structure then supports 32 bit exponents and 110 bit significands – 142 bits in all. The conversion to 128 bits takes place as part of the load/store of a `quad` value.

   A side effect of this proposal is that it is in general no longer possible to sort `quad` values as integers – a property enjoyed by current formats. To address this concern, we suggest that a machine mode exist to force either a 16 or a 32 bit exponent memory representation. This allows instrument makers (and other interested parties) to support only one of these forms, sort them as integers, and yet produce data which may be directly processed by general purpose computers.

   Few machines have implemented the current standard's "extended" type. We suspect that this is the case in part because additional precision beyond that provided by `double` is important in very few applications. In contrast, additional exponent range is clearly beneficial to the increasingly important area of stochastic modeling. Nevertheless some applications would benefit from added precision, and it is for this reason that we advocate a two mode data format.

Either of these choices would alleviate the need for our implementation of the `pr_t` abstraction.

Implementing the 64-bit `balanced double` is the better of the two options since most stochastic modeling applications do not in fact require great precision. But there is perhaps a greater chance to influence the design of a 128-bit standard since none yet exists.

3.2 Execution Timings

The efficiency of our implementation across different compilers and machine architectures is documented in the following table. For each machine, we provide the internal clock rate of the CPU.

Machine	**Compiler**	**dbl**	**macro**	**ptr**	**func**	**log**
DEC 3000/700, 225MHz	gcc -O4	0.28	2.21	2.52	4.19	18.64
HP 755, 100Mhz	gcc -O4	0.41	2.75	4.37	9.55	30.04
SGI R4400, 150MHz	gcc -O4	0.60	3.20	4.90	8.40	35.80
SGI R4000, 100MHz	gcc -O4	0.91	4.92	7.32	12.53	53.48
Sun Sparc10, 50/100MHz	cc -fast	0.80	5.20	7.80	16.00	99.60
Weitek Sparc2, 80MHz	gcc -O4	0.77	4.95	6.74	26.38	81.55
Sun Sparc2, 40MHz	gcc -O4	2.08	13.54	17.81	33.27	140.74

The values shown represent seconds of user CPU time to perform $10,000,000$ addition operations. These results were obtained via the `pr.bench` executable described below, with `pr.o` compiled with the `-DNEBUG` option to remove all `assert()` statements. For each compiler, we chose the optimization level and compiler options that gave the best results when using the `libpr.a` archive. `gcc` nearly always produced faster `pr.bench` executables than the manufacturer's native compiler.

The **dbl** column shows the time required to perform 10,0000 additions using double-precision floating point numbers.

The **macro** column shows the result of moving `pr_add()` inline so that function calling overhead is eliminated. This represents the performance that might be expected from a macro implementation.

The **ptr** column shows the timing results obtained using the `pr_add_ptr()` mutator, which is passed the memory addresses of its arguments and the address of the return value. Focusing on our fastest (and newest) machine, the DEC Alpha AXP 3000/700, we see that `pr_add_ptr()` is 9 times slower than the `double` additions. The 0.28 second time for the latter corresponds to 36 MFLOPS while `pr_add_ptr()` clocks in at 4 MFLOPS.

The **func** column shows the result of using a `pr_add()` function which is passed two `pr_t` objects (not pointers) and returns an object of the same type.

The **log** column shows the result for a straightforward implementation of addition for probability values in the logarithmic representation.

These timing results motivated us to support the pointer-based `pr_add_ptr()` form in addition to the functional `pr_add()` form as a reasonable compromise between the faster (but less safe) macro approach, and the slower but more modular functional design.

3.3 Where to get it

The `pr_t` library source code is freely available at cs.princeton.edu:/pub/pr.tar.gz via anonymous ftp. Be sure to transfer this file in binary mode. You will need `gunzip` and `tar` to unpack the distribution.

References

[1] X. D. Huang, Y. Ariki, and M. A. Jack, *Hidden Markov Models for Speech Recognition*. Edinburgh University Press, 1990.

A User's Guide

Our implementation provides a test suite `pr.test.c` and a benchmark `pr.bench.c` along with a header file `pr.h` and source code `pr.c`. To compile the code and create a `libpr.a` archive:

1. Set the environment variables `ARCH`, `SCC`, `UCC`, `LD`, and `RANLIB` in order to specify the target machine architecture and compiler. One way to do this is to `source cshrc.all` after creating the appropriate `cputype` shell script (see section B below).

2. Execute `make all`.

Once the programs are compiled, the test suite and benchmark programs are run as follows:

```
> pr.test
zero = 0.000000b-2147483648; unity = 0.500000b1; epsilon = 0.500000b-51
Precision test will involve 100000 pseudorandom numbers.
  Arithmetic agreement to 13.4 digits of precision
  Conversion agreement to 13.9 digits of precision with pr2double()
  Power agreement to 13.1 digits of precision with power < 7
Test complete.

> time pr.bench 3
pr_t pointer addition
2.523u 0.004s 0:02.53 99.6% 0+0k 0+0io 0pf+0w
```

The user should note that the `pr.c` source code contains a large number of `assert()` statements to ensure correctness and to detect possible caller errors. Once the user is confident of the correctness of his code, we strongly advise that the `libpr.a` archive be recompiled with the `-DNDEBUG` option in order to remove the `assert()` statements. Removing the `assert()` statements will nearly halve the execution time of the `pr_t` arithmetic operations. In our current environment, this is accomplished using the `make unsafe` option.

B Makefile

We achieve portability across different platforms using `make` in conjunction with the following five environment variables.

- `ARCH` contains the current machine architecture, eg., one of `sun4`, `alpha`, `hppa`, or `sgi` at our site.

- `SCC` contains the command and command-line arguments necessary for a safe compilation (no optimization, create symbol table for debugging) on the current machine architecture.
- `UCC` contains the command and command line arguments necessary for an *un*safe compilation (maximal optimization, `-DNDEBUG` to eliminate `assert()`, and no symbol table).
- `LD` contains the command and command line arguments necessary to create an executable by loading a set of object files.
- `RANLIB` contains the command necessary to convert an archive into a random-access archive.

The following shell script was used to set these environment variables during Fall 1994.

⟨*cshrc.all* 11⟩≡

```
##########################################################################
#
#  This file defines the following environment variables
#       SCC     - safe C compiler to produce debuggable .o object files
#       UCC     - unsafe C compile to produce optimized .o object files
#       LD      - loader to combine multiple .o files into an executable
#       CC      - bare C compiler (should not be used!)
#       RANLIB  - ranlib
#
#  In addition, your Makefile will probably have to define
#       IFLAGS  - all -I<directory> include options
#       LFLAGS  - all -L<directory> loader options
#       LLIBS   - all -l<archive> loader options
#
##########################################################################
setenv HOSTCPU `cputype`
switch (${HOSTCPU})
   case alpha:
        setenv ARCH    alpha        # OSF/1 v3.0
        setenv CC      'gcc -ansi -pedantic -Wall'
        setenv SCC     'gcc -ansi -pedantic -Wall -g'
        setenv UCC     'gcc -ansi -pedantic -Wall -O4 -DNDEBUG'
        setenv LD      'gcc -ansi -pedantic -Wall -static'
        setenv RANLIB  ranlib
        breaksw
   case sun4:
   case sparc:
        setenv ARCH    sun4         # SunOS 4.1.3
        setenv CC      'gcc -ansi -pedantic -Wall'
        setenv SCC     'gcc -ansi -pedantic -Wall -g -I/usr/local/include/ansi'
```

```
        setenv UCC     'gcc -ansi -pedantic -Wall -O4 -DNDEBUG -I/usr/local/include/ansi'
        setenv LD      'gcc -ansi -pedantic -Wall'
        setenv RANLIB  ranlib
        breaksw
    case hp9000s800:
    case hppa:
        setenv ARCH    hppa         # HPUX 9.0
        setenv LDOPTS  -L/usr/local/lib:-L/lib/pa1.1:-L/usr/lib/pa1.1:-L/lib:-L/usr/lib
        setenv PATH    /usr/X11R5/bin:${PATH}
        setenv CC      'gcc -ansi -pedantic -Wall'
        setenv SCC     'gcc -ansi -pedantic -Wall -g'
        setenv UCC     'gcc -ansi -pedantic -Wall -O4 -DNDEBUG'
        setenv LD      'gcc -ansi -pedantic -Wall'
        setenv RANLIB  ranlib
        breaksw
    case iris:
    case iris4d:
        setenv ARCH    sgi
        setenv CC      'gcc -ansi -pedantic -Wall'
        setenv SCC     'gcc -ansi -pedantic -Wall -g'
        setenv UCC     'gcc -ansi -pedantic -Wall -O4 -DNDEBUG'
        setenv LD      'gcc -ansi -pedantic -Wall'
        setenv RANLIB  ranlib
        breaksw
    default:
        echo "Warning: unknown architecture ${HOSTCPU}"
  endsw
```

This code is written to file `cshrc.all`.

Our `Makefile` contains targets for safe and unsafe `libpr.a`, along with the `pr.test` test suite and the `pr.bench` benchmark.

⟨*Makefile* 13⟩≡

```
LIBINCLUDE  = ../../include
LIBARCHIVE  = ../../lib.${ARCH}

IFLAGS  =
LFLAGS  =
LLIBS   = -lm

ARCHIVE = libpr.a
SOURCES = pr.c pr.h
OBJECTS = pr.o

TEST_SOURCES = pr.test.c pr.bench.c

TARGETS = ${ARCHIVE} pr.test pr.bench

all: $(TARGETS)

pr.o:   pr.c pr.h
        ${SCC} ${IFLAGS} -c pr.c

${ARCHIVE}:     ${OBJECTS}
        ar rc $@ $?
        ${RANLIB} $@

# Test programs

pr.test:        pr.test.o ${ARCHIVE}
        ${LD} -o $@ pr.test.o ${ARCHIVE} ${LFLAGS} ${LLIBS}

pr.test.pure:   pr.test.o ${ARCHIVE}
        purify ${LD} -o $@ pr.test.o ${ARCHIVE} ${LFLAGS} ${LLIBS}

pr.test.o:      pr.test.c pr.h
        ${SCC} ${IFLAGS} -c pr.test.c

pr.bench:       pr.bench.c pr.c
        rm -f pr.o pr.bench.o
        ${UCC} ${IFLAGS} -c pr.c
        ${UCC} ${IFLAGS} -c pr.bench.c
        ${LD} -o $@ pr.bench.o pr.o ${LFLAGS} ${LLIBS}
        rm -f pr.o pr.bench.o

# Installation
```

```
unsafe: pr.c pr.h
        ${UCC} ${IFLAGS} -c pr.c
        ar rc ${ARCHIVE} ${OBJECTS}
        @chgrp hand ${ARCHIVE}
        @rm -f ${LIBARCHIVE}/unsafe/${ARCHIVE}
        cp ${ARCHIVE} ${LIBARCHIVE}/unsafe
        @chmod 664 ${LIBARCHIVE}/unsafe/${ARCHIVE}

clean:
        @rm -f ${TARGETS} pr.test.pure *.o

install: ${ARCHIVE}
        @chgrp hand pr.h ${ARCHIVE}
        @rm -f ${LIBARCHIVE}/${ARCHIVE}
        cp ${ARCHIVE} ${LIBARCHIVE}
        @rm -f ${LIBINCLUDE}/pr.h
        cp pr.h ${LIBINCLUDE}
        @chmod 664 ${LIBINCLUDE}/pr.h ${LIBARCHIVE}/${ARCHIVE}
```

This code is written to file **`Makefile`**.

C Implementation

Our implementation consists of three parts. Firstly, we provide a concrete implementation of the `pr_t` type. Secondly, we implement the actual `pr_t` operations. Thirdly, we implement a comprehensive test suite. Finally, we implement a comprehensive benchmark in order to compare the performance of `double` operations, the pointer-based arithmetic operations, and the basic arithmetic operations without pointers.

C.1 Data Type

We have considered the following four implementations of the `pr_t` data type.

1. Probabilities are represented straightforwardly as double precision floating point numbers. Addition is by floating point addition, multiplication is by floating point multiplication. Of course in this case the exponent range is not extended at all.

2. Probabilities are represented as their natural logarithms, using double precision floating point numbers. Addition is by exponentiation of the addend and augend, followed by floating point addition and natural logarithm. Multiplication is by floating point addition. A major flaw with this implementation is that floating point exponentiation is extremely slow. It should be noted however that this option would be the most attractive if special purpose hardware to compute double precision 'T' values were available (see next item).

3. Probabilities are represented as integral logarithms. The integers may be 32 bits or 64 bits, depending on the machine. Multiplication is by fixed point addition. Addition of probabilities is performed using a lookup table as follows [1]. Given two probabilities in logarithmic representation, $\log_b(p_1)$ and $\log_b(p_2)$, such that $p_1 \geq p_2$, we calculate $\log_b(p_1 + p_2)$ as $\log_b p_1 + T(\log_b p_2 - \log_b p_1)$ using the lookup table defined as $T(n) = \log_b(1 + b^n)$ if $T(n) \geq 0.5$, otherwise $T(n) = 0$. (For 32-bit integers and $b \in [1.0001, 1.00001]$, $|T(n)| \in [99041, 1220614]$.) In this implementation, `pr_add()` would require two fixed point additions, two comparisons, and one table lookup in this implementation. In order to fit the lookup table in cache, the table may be further compacted using interpolation. A major flaw with this implementation is that 32-bit probability values with enough range to be generally useful must also lack the high precision and high speed that we can easily obtain using the IEEE double precision hardware in most systems.

4. Probabilities are represented by a pair of numbers: a floating point number for the significand and a fixed point number for the exponent. For the significand, we may use either a `float` or a `double`. The former choice will give us improved time and space performance, particularly on a machine with 64-bit data paths, while the latter choice will give us greater precision.

We choose the fourth method with maximal precision and the following type implementation results.

⟨*pr typedefs* 16⟩≡

```
typedef struct pr_tag
{
  double  sig;
  int     exp;
} pr_t;
```

This code is used in chunk 18.

This is a floating point format supporting a huge exponent range. When used for expectation maximization, overflow will be avoided provided that the number of observations T, times $-\log_2(p)$, remains smaller than this limit (p denotes the smallest parameter in the model). On machines with 32 bit integers, and assuming that the model's parameters lie within the range of a `double`, observation sequences of over $3,000,000$ may be processed without underflow. If the model's parameters are additionally constrained after each reestimation to fall in a smaller range, say no smaller than `1E-10`, then observation sequences of up to $32,000,000$ may be processed. Given 64 bits of exponent, no practical limit is in sight.

Notice that our type implementation supports values in the interval $[0, +\infty)$, and so our `pr_t` library has none of the 'beyond unity' problems discussed above.

The following constants are provided in the implementation.

`PR_RADIX`	exponent radix
`PR_PRECISION`	open upper bound on number of bits in mantissa
`PR_MANT_LOW`	closed lower bound on value of balanced mantissa
`PR_MANT_HIGH`	open upper bound on value of balanced mantissa

Let `p` be an object of type `pr_t`. Then `p` represents the real number $\texttt{p.sig} \cdot 2^{\texttt{p.exp}}$ for radix 2. In order to maintain precision, we will attempt to keep the significand in the interval $[0.5, 1)$ (assuming a radix of 2). The `PR_MANT_LOW` and `PR_MANT_HIGH` constants store the lower and upper bounds of this interval. These constants are currently implemented as follows.

⟨*pr constants* 17a⟩≡
```
#define PR_RADIX        2.0
#define PR_PRECISION    (53+1)
#define PR_MANT_LOW     (1/PR_RADIX)
#define PR_MANT_HIGH    1.0
```

This code is used in chunk 18.

The following predicate returns `TRUE` iff its argument satisfies the implementation-specific constraint that the significand falls within the interval $[0.5, 1)$, or the exponent is the smallest representable value. Note that it is not possible to balance the significand for extremely small probability values, and so `pr_is_balanced()` returns `TRUE` iff either the `px` argument is balanced or it is not possible to further balance the argument.

⟨*pr operations* 3a⟩+≡
```
BOOLEAN    pr_is_balanced(pr_t px);
```

⟨*pr.h* 18⟩≡
```
/*
 * COPYRIGHT NOTICE, LICENSE AND DISCLAIMER
 *
 * (c) 1994 by Peter Yianilos and Eric Sven Ristad.  All rights reserved.
 *
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation without fee for not-for-profit academic or research
 * purposes is hereby granted, provided that the above copyright notice
 * appears in all copies and that both the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that neither the authors' names nor the name of any
 * entity or institution to which they are related be used in advertising
 * or publicity pertaining to distribution of the software without
 * specific, written prior permission.
 *
 * The authors disclaim all warranties with regard to this software,
 * including all implied warranties of merchantability and fitness.  In
 * no event shall the authors or any entities or institutions to which
 * they are related be liable for any special, indirect or consequential
 * damages or any damages whatsoever resulting from loss of use, data or
 * profits, whether in an action of contract, negligence or other
 * tortious action, arising out of or in connection with the use or
 * performance of this software.
 *
 */

#ifndef PR_INCLUDED
#define PR_INCLUDED

#include <stdio.h>

#ifndef COMMON_INCLUDED
   typedef int    BOOLEAN;
#endif
#ifndef FALSE
   #define FALSE  0
#endif
#ifndef TRUE
   #define TRUE   1
#endif
```
⟨*pr constants* 17a⟩
⟨*pr typedefs* 16⟩
⟨*pr operations* 3a⟩

```
#endif  /* PR_INCLUDED */
```

This code is written to file pr.h.

C.2 Operations

⟨*pr.c* 20⟩≡

```
/*
 * COPYRIGHT NOTICE, LICENSE AND DISCLAIMER
 *
 * (c) 1994 by Peter Yianilos and Eric Sven Ristad.  All rights reserved.
 *
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation without fee for not-for-profit academic or research
 * purposes is hereby granted, provided that the above copyright notice
 * appears in all copies and that both the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that neither the authors' names nor the name of any
 * entity or institution to which they are related be used in advertising
 * or publicity pertaining to distribution of the software without
 * specific, written prior permission.
 *
 * The authors disclaim all warranties with regard to this software,
 * including all implied warranties of merchantability and fitness.  In
 * no event shall the authors or any entities or institutions to which
 * they are related be liable for any special, indirect or consequential
 * damages or any damages whatsoever resulting from loss of use, data or
 * profits, whether in an action of contract, negligence or other
 * tortious action, arising out of or in connection with the use or
 * performance of this software.
 *
 */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>          /* for abort() only */
#include <math.h>
#include <limits.h>
#include <float.h>
#include "pr.h"

static char pr_rcsid[] = "$Id: pr.nw,v 1.51 1994/10/30 01:17:36 ristad Exp $";

#define ANNOUNCE_RCSID      TRUE

/* Note: these choices seem to have no effect on precision.  */
#define MAXIMUM_ACCURACY    FALSE
#define ADD_BEFORE_COERCE   FALSE

#if MAXIMUM_ACCURACY
 extern double expm1();
```

```
 extern double log1p();
 #define EXP(x)       (expm1(x)+1)
 #define LOG(x)       (log1p((x)-1))
#else
 #define EXP(x)       (exp(x))
 #define LOG(x)       (log(x))
#endif

#define NATS_NEG_INFINITY   -HUGE_VAL    /* nats for zero probability */

double Pr_Table[PR_PRECISION+1];  /* used by pr.bench.c so can't be static */
static double   Pr_ln2;
static BOOLEAN  Pr_Is_Initialized = FALSE;

/*******************************
 *     Internal Utilities      *
 *******************************/

static void pr_initialize(void)
{
  double   x;
  unsigned u;
  int      i, j;

  Pr_Is_Initialized = TRUE;

  for (i = 0, x = 1.0; i <= PR_PRECISION; ++i, x /= PR_RADIX)
    Pr_Table[i] = x;

  Pr_ln2 = LOG(2.0);
  assert(log(2.0) == LOG(2.0));

  i = INT_MAX;
  j = INT_MIN;
  u = (unsigned)(i - j);
  if (u != UINT_MAX)    abort();
  if (sizeof(int) < 4)  abort();
  if (PR_MANT_LOW <= 0) abort();

#ifdef FLT_RADIX
  if (FLT_RADIX != 2)   abort();
#endif

#ifdef DBL_MANT_DIG
  if (DBL_MANT_DIG != (PR_PRECISION - 1)) abort();
#endif
```

```
#if ANNOUNCE_RCSID
#ifndef NDEBUG
  fprintf(stderr, "%s\n", pr_rcsid);
#else
  fprintf(stderr, "%s\t>unsafe<\n", pr_rcsid);
#endif
#endif

}

BOOLEAN pr_is_balanced(pr_t px)
{
  return(((PR_MANT_LOW <= (px).sig) && ((px).sig < PR_MANT_HIGH))
         || ((px).exp == INT_MIN));
}


void pr_fprintf(pr_t px, FILE *stream)
{
  assert(pr_is_balanced(px));
  fprintf(stream, "%fb%d", px.sig, px.exp);
}

void pr_printf(pr_t px)
{
  pr_fprintf(px, stdout);
}


/*******************************
 *     Basic Type Coercion     *
 *******************************/

pr_t pr_double2pr(double x)
{
  pr_t px;

  if (!Pr_Is_Initialized) pr_initialize();

  if (x == 0)
    {
      px.exp = INT_MIN;
      px.sig = 0;
    }
  else
    {
      px.sig = frexp(x, &px.exp);
```

```
    }

  assert(pr_is_balanced(px));
  return(px);
}

double pr_pr2double(pr_t px)
{
  return(ldexp(px.sig, px.exp));
}


pr_t pr_nats2pr(double x)
{
  pr_t px;

  if (!Pr_Is_Initialized) pr_initialize();

  if (x == NATS_NEG_INFINITY)   /* largest codelength = smallest pr */
    {
      px = pr_double2pr(0.0);
      assert(px.exp == INT_MIN);
      assert(px.sig == 0);
    }
  else  /* valid codelength for nonzero probability */
    {
      x /= Pr_ln2;
      px.exp = floor(x);
      px.sig = exp((x - px.exp) * Pr_ln2);
      if (px.sig >= PR_MANT_HIGH)
        {
          px.sig /= PR_RADIX;
          ++px.exp;
        }
    }
  assert(pr_is_balanced(px));
  return(px);
}

double pr_pr2nats(pr_t px)
{
  double a, b;

  if (pr_is_zero(px)) return(NATS_NEG_INFINITY);

  /* Note: b = log1p(px.sig - 1) does not improve accuracy */
  a = px.exp * Pr_ln2;
```

```
  b = LOG(px.sig);
  return(a+b);
}


/************************************
 * Distinguished Value Constructors *
 ************************************/

pr_t pr_zero(void)
{
  return(pr_double2pr(0.0));
}

pr_t pr_unity(void)
{
  return(pr_double2pr(1.0));
}

pr_t pr_epsilon(void)
{
#ifdef DBL_EPSILON
  return(pr_double2pr(DBL_EPSILON));
#else
  return(pr_double2pr(2.2204460492503131000e-16));
#endif
}

/********************************
 *     Basic Predicates        *
 ********************************/

BOOLEAN pr_is_zero(pr_t px)
{
  return((px.sig == 0) && (px.exp == INT_MIN));
}

BOOLEAN pr_is_unity(pr_t px)
{
  return((px.sig == 0.5) && (px.exp == 1));
}


BOOLEAN pr_is_valid(pr_t px, pr_t epsilon)
{
  pr_t pv;
  int cmp;
```

```
  cmp = pr_exact_compare_ptr(&px,&epsilon);
  if (cmp < 0)
    {
      pv = pr_double2pr(0.0);
      pr_add_ptr(&px,&epsilon,&px);
      return(pr_exact_compare_ptr(&pv,&px) <= 0);
    }
  else if (cmp > 0)
    {
      pv = pr_double2pr(1.0);
      pr_add_ptr(&pv,&epsilon,&pv);
      return(pr_exact_compare_ptr(&px,&pv) <= 0);
    }
  else /* cmp == 0 */
    {
      return(TRUE);
    }
}


/*******************************
 *      Basic Comparison       *
 *******************************/

int pr_compare(pr_t px, pr_t py, pr_t tolerance)
{
  return(pr_compare_ptr(&px,&py,&tolerance));
}

int pr_exact_compare(pr_t px, pr_t py)
{
  return(pr_exact_compare_ptr(&px,&py));
}

pr_t pr_difference(pr_t px, pr_t py)
{
  pr_t pz;
  (void) pr_difference_ptr(&px,&py,&pz);
  return(pz);
}

/*******************************
 *      Basic Arithmetic       *
 *******************************/

pr_t pr_add(pr_t px, pr_t py)
```

```
{
  pr_t pz;
  pr_add_ptr(&px,&py,&pz);
  return(pz);
}

pr_t pr_multiply(pr_t px, pr_t py)
{
  pr_t pz;
  pr_multiply_ptr(&px,&py,&pz);
  return(pz);
}

pr_t pr_divide(pr_t px, pr_t py)
{
  pr_t pz;
  pr_divide_ptr(&px,&py,&pz);
  return(pz);
}

pr_t pr_power(pr_t px, double n)
{
  pr_t pz;
  pr_power_ptr(&px,n,&pz);
  return(pz);
}


/*******************************
 *   Pointer-Based Comparison   *
 *******************************/

int pr_compare_ptr(pr_t *px_p, pr_t *py_p, pr_t *epsilon_p)
{
  int sign;
  pr_t magnitude;

  sign = pr_difference_ptr(px_p,py_p,&magnitude);
  if (pr_exact_compare_ptr(&magnitude,epsilon_p) <= 0)
    return(0);
  else
    return(sign);
}

int pr_exact_compare_ptr(pr_t *px_p, pr_t *py_p)
{
  if (px_p->exp > py_p->exp) return 1;
```

```
  if (px_p->exp < py_p->exp) return -1;
  if (px_p->sig > py_p->sig) return 1;
  if (px_p->sig < py_p->sig) return -1;
  return 0;
}

int pr_difference_ptr(pr_t *px_p, pr_t *py_p, pr_t *pz_p)
{
  int sign;
  unsigned pr_diff;

  /* store negative of smaller value in pz and add to larger value */

  if (px_p->exp > py_p->exp)
    {
      sign = 1;
      pr_diff = (unsigned) (px_p->exp - py_p->exp);
      pz_p->exp = px_p->exp;
      if (pr_diff <= PR_PRECISION)
        pz_p->sig = px_p->sig - (py_p->sig * Pr_Table[pr_diff]);
      else
        pz_p->sig = px_p->sig;
    }
  else if (px_p->exp < py_p->exp)
    {
      sign = -1;
      pr_diff = (unsigned) (py_p->exp - px_p->exp);
      pz_p->exp = py_p->exp;
      if (pr_diff <= PR_PRECISION)
        pz_p->sig = py_p->sig - (px_p->sig * Pr_Table[pr_diff]);
      else
        pz_p->sig = py_p->sig;
    }
  else if (px_p->sig > py_p->sig)
    {
      pz_p->exp = py_p->exp;
      pz_p->sig = px_p->sig - py_p->sig;
      sign = 1;
    }
  else if (px_p->sig < py_p->sig)
    {
      pz_p->exp = px_p->exp;
      pz_p->sig = py_p->sig - px_p->sig;
      sign = -1;
    }
  else
    {
```

```
      pz_p->sig = 0;      /* pz_p->exp = INT_MIN occurs below */
      sign = 0;
    }

  /* now partially balance significand/exponent to minimize rounding error */

  if (pz_p->sig < PR_MANT_LOW)
    {
      if (pz_p->sig == 0)
        {
          pz_p->exp = INT_MIN;
        }
      else
        {
          while (pz_p->sig < PR_MANT_LOW)
            {
              assert(pz_p->exp > INT_MIN);  /* else Exponent Underflow */
              pz_p->sig *= PR_RADIX;
              --(pz_p->exp);
            }
        }
    }

  assert(pr_is_balanced(*pz_p));
  return(sign);
}

/********************************
 *   Pointer-Based Arithmetic   *
 ********************************/


void pr_add_ptr(pr_t *px_p, pr_t *py_p, pr_t *pz_p)
{
  unsigned pr_diff;

  assert(Pr_Is_Initialized);

  if (px_p->exp >= py_p->exp)
    {
      pr_diff = (unsigned) (px_p->exp - py_p->exp);
      pz_p->exp = px_p->exp;

      if (pr_diff <= PR_PRECISION)    /* could smaller affect larger? */
        pz_p->sig = px_p->sig + (py_p->sig * Pr_Table[pr_diff]);
      else
        pz_p->sig = px_p->sig;
```

```
    }
  else  /* py greater than px */
    {
      pr_diff = (unsigned) (py_p->exp - px_p->exp);
      pz_p->exp = py_p->exp;

      if (pr_diff <= PR_PRECISION)
        pz_p->sig = py_p->sig + (px_p->sig * Pr_Table[pr_diff]);
      else
        pz_p->sig = py_p->sig;
    }

  /* now partially balance exponent and significand */
  if (pz_p->sig >= PR_MANT_HIGH)
    {
      pz_p->sig /= PR_RADIX;
      ++(pz_p->exp);
    }

  assert(pr_is_balanced(*pz_p));
}

void pr_multiply_ptr(pr_t *px_p, pr_t *py_p, pr_t *pz_p)
{
  pz_p->sig = px_p->sig * py_p->sig;
  pz_p->exp = px_p->exp + py_p->exp;

  if (pz_p->sig < PR_MANT_LOW)
    {
      if (pz_p->sig == 0)
        {
          pz_p->exp = INT_MIN;
        }
      else
        {
          assert(pz_p->exp > INT_MIN);  /* else Exponent Underflow */
          pz_p->sig *= PR_RADIX;
          --(pz_p->exp);
        }
    }

  assert(pr_is_balanced(*pz_p));
}

void pr_divide_ptr(pr_t *px_p, pr_t *py_p, pr_t *pz_p)
{
  pz_p->sig = px_p->sig / py_p->sig;
```

```
  pz_p->exp = px_p->exp - py_p->exp;

  if (pz_p->sig >= PR_MANT_HIGH)
    {
      pz_p->sig /= PR_RADIX;
      ++(pz_p->exp);
    }
  else if (pz_p->sig == 0)
    {
      pz_p->exp = INT_MIN;
    }

  assert(pr_is_balanced(*pz_p));
}


void pr_power_ptr(pr_t *px_p, double n, pr_t *pz_p)
{
  double fractional, whole, base, residue;

  fractional = modf(px_p->exp * n, &whole);
  residue = EXP(Pr_ln2 * fractional);

  base = pow(px_p->sig, n) * residue;
  pz_p->sig = frexp(base, &(pz_p->exp));

  if (pz_p->sig == 0)
    {
      pz_p->exp = INT_MIN;
    }
  else
    {
#if ADD_BEFORE_COERCE
      whole = whole + (double)pz_p->exp;
      assert((INT_MIN < whole) && (whole < INT_MAX));
      pz_p->exp = (int)whole;
#else
      pz_p->exp = pz_p->exp + (int)whole;
#endif
    }

  assert(pr_is_balanced(*pz_p));
}
```

This code is written to file pr.c.

D Test Suite

⟨*pr.test.c* 31⟩≡

```
/*
 * COPYRIGHT NOTICE, LICENSE AND DISCLAIMER
 *
 * (c) 1994 by Peter Yianilos and Eric Sven Ristad.  All rights reserved.
 *
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation without fee for not-for-profit academic or research
 * purposes is hereby granted, provided that the above copyright notice
 * appears in all copies and that both the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that neither the authors' names nor the name of any
 * entity or institution to which they are related be used in advertising
 * or publicity pertaining to distribution of the software without
 * specific, written prior permission.
 *
 * The authors disclaim all warranties with regard to this software,
 * including all implied warranties of merchantability and fitness.  In
 * no event shall the authors or any entities or institutions to which
 * they are related be liable for any special, indirect or consequential
 * damages or any damages whatsoever resulting from loss of use, data or
 * profits, whether in an action of contract, negligence or other
 * tortious action, arising out of or in connection with the use or
 * performance of this software.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <limits.h>

#include "pr.h"

#define MINPREC         10
#define DUMPREC         1000
#define POWER_LIMIT     7         /* must use 7 on sun4, 10 on alpha */
#define USE_NATS        FALSE     /* don't use nats for max accuracy */

#if USE_NATS
  #define Pr2Double(PX)   (exp(pr_pr2nats(PX)))
#else
  #define Pr2Double(PX)   (pr_pr2double(PX))
#endif
```

```
extern void     srand48(long seedval);
extern double   drand48(void);

double  worstdigerr = DUMPREC;

int count;

void checknormal(pr_t *px)
{
  if (px->sig == 0)
    {
      assert(px->exp == INT_MIN);
    }
  else
    {
      assert(px->sig >= PR_MANT_LOW);
      assert(px->sig < PR_MANT_HIGH);
    }
}

void logerror(double x, double y)
{
  double abserr, relerr, digerr;

  abserr = fabs(x-y);

  if (x != 0)
    {
      relerr = abserr/x;
      if (relerr == 0)
        digerr = DUMPREC;
      else
        digerr = -log10(relerr);

      if (digerr < worstdigerr) worstdigerr = digerr;

    }
  else
    {
      assert(abserr == 0);
      digerr = DUMPREC;
    }

  assert(digerr > MINPREC);
}
```

```
void draw(double *x, pr_t *px)
{
  int sign;
  double e, m;

  if (drand48() < .01)
    {
      *x = 0;
      *px = pr_double2pr(0.0);
      return;
    }

  if (drand48() < 0.5)
    sign = 1;
  else
    sign = -1;

  e = drand48() * (100*drand48()) * sign;
  m = drand48();
  *x = exp(e + m);
  *px = pr_nats2pr(e+m);
}

int main(int argn, char *argv[])
{
  int i, n;
  double x, y, z;
  pr_t px, py, pz;

  if (argn == 1)
    n = 100000;
  else
    n = atoi(argv[1]);

#ifdef NDEBUG
  abort();          /* assert() required for correct testing */
#endif

  /* Verify Coercion and Predicates */
  px = pr_double2pr(0.0);
  assert(pr_is_valid(px,px));
  assert(pr_is_balanced(px));
  assert(pr_is_zero(px));
  assert(pr_is_zero(pr_power(px, 99999.0)));
  printf("zero = ");
  pr_printf(px);
```

```
  py = pr_double2pr(1.0);
  assert(pr_is_valid(py,px));
  assert(pr_is_balanced(py));
  assert(pr_is_unity(py));
  assert(pr_is_zero(pr_multiply(px,py)));
/*
  assert(pr_is_unity(pr_power(py, 999.0)));
*/
  printf("; unity = ");
  pr_printf(py);

  pz = pr_epsilon();
  assert(pr_is_valid(pz,px));
  assert(pr_is_balanced(pz));
  assert(pr_exact_compare(pz,px) > 0);
  assert(pr_is_unity(pr_divide(pz,pz)));
  assert(pr_is_unity(pr_power(pz,0)));
  printf("; epsilon = ");
  pr_printf(pz);
  printf("\n");

  /* Precision Test */
  printf("Precision test will involve %d pseudorandom numbers.\n", n);

  srand48((long) 12345);

  px = pr_double2pr(0.0);
  assert(px.sig == 0);
  assert(px.exp == INT_MIN);

  for (i = 0; i < n; ++i)
    {
      draw(&x, &px);
      draw(&y, &py);

      ++count;

      checknormal(&px);
      checknormal(&py);
      assert(pr_is_balanced(px));
      assert(pr_is_balanced(py));

      assert(pr_is_zero(px) == (x == 0));

      pz = pr_double2pr(x);
      checknormal(&pz);
      assert(pr_is_balanced(pz));
```

```
logerror(x, Pr2Double(pz));

pz = pr_add(px, py);
checknormal(&pz);
assert(pr_is_balanced(pz));
logerror(x+y, Pr2Double(pz));

pz = pr_multiply(px, py);
checknormal(&pz);
assert(pr_is_balanced(pz));
logerror(x*y, Pr2Double(pz));

if (!pr_is_zero(py))
  {
    pz = pr_divide(px, py);
    checknormal(&pz);
    assert(pr_is_balanced(pz));
    logerror(x/y, Pr2Double(pz));
  }

/* used to verify reimplementation of pr_difference_ptr */
assert(pr_exact_compare(pr_difference(px,py), pr_difference(py,px)) == 0);
assert(pr_exact_compare(px, px) == 0);
assert(pr_difference_ptr(&px, &px, &pz) == 0);
checknormal(&pz);
assert(pr_is_zero(pz));

if ((x != 0) || (y != 0))
  {
    if ((-log10(fabs(x-y)/(x+y))) <= MINPREC)
      {
        assert(pr_exact_compare(px, py) == (x > y ? 1: -1));
        assert(pr_difference_ptr(&px, &py, &pz) == (x > y ? 1 : -1));
        checknormal(&pz);
        assert(pr_is_balanced(pz));

        z = Pr2Double(pz);

        if (x > y)
          {
            if (px.sig-py.sig >= 0.1) logerror(x-y, z);
          }
        else
          {
            if (py.sig-px.sig >= 0.1) logerror(y-x, z);
          }
      }
```

```
        }
    }
  printf("  Arithmetic agreement to %.1f digits of precision\n", worstdigerr);

  worstdigerr = DUMPREC;
  for (i = 0; i < n; ++i)
    {
      draw(&x, &px);
      draw(&y, &py);

      logerror(x, Pr2Double(px));
      logerror(y, Pr2Double(py));
    }
  printf("  Conversion agreement to %.1f digits of precision with %s\n",
         worstdigerr, (USE_NATS ? "pr2nats()" : "pr2double()"));


  worstdigerr = DUMPREC;
  for (i = 0; i < n; ++i)
    {
      draw(&x, &px);
      draw(&y, &py);

      if ((x > 0) && (y > 0))
        {
          if (y < POWER_LIMIT)
            {
              logerror(pow(x,y), Pr2Double(pr_power(px,y)));
              logerror(pow(x,-y), Pr2Double(pr_power(px,-y)));
            }
          if (x < POWER_LIMIT)
            {
              logerror(pow(y,x), Pr2Double(pr_power(py,x)));
              logerror(pow(y,-x), Pr2Double(pr_power(py,-x)));
            }
        }
    }
  printf("  Power agreement to %.1f digits of precision with power < %d\n",
         worstdigerr, POWER_LIMIT);
  printf("Test complete.\n");
  return 0;
}
```

This code is written to file `pr.test.c`.

E A Benchmark Program

⟨*pr.bench.c* 37⟩≡

```
/*
 * COPYRIGHT NOTICE, LICENSE AND DISCLAIMER
 *
 * (c) 1994 by Peter Yianilos and Eric Sven Ristad.  All rights reserved.
 *
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation without fee for not-for-profit academic or research
 * purposes is hereby granted, provided that the above copyright notice
 * appears in all copies and that both the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that neither the authors' names nor the name of any
 * entity or institution to which they are related be used in advertising
 * or publicity pertaining to distribution of the software without
 * specific, written prior permission.
 *
 * The authors disclaim all warranties with regard to this software,
 * including all implied warranties of merchantability and fitness.  In
 * no event shall the authors or any entities or institutions to which
 * they are related be liable for any special, indirect or consequential
 * damages or any damages whatsoever resulting from loss of use, data or
 * profits, whether in an action of contract, negligence or other
 * tortious action, arising out of or in connection with the use or
 * performance of this software.
 *
 */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>          /* for abort() only */
#include <math.h>

#include "pr.h"

#define OUTER 100000
#define INNER 100

#define BE_INFORMATIVE     TRUE

extern double Pr_Table[];

int main(int argn, char *argv[])
{
  int    i, j, s;
  double x, all, result;
```

```
  double d[INNER], ld[INNER];
  pr_t   px, pd[INNER];
  unsigned pr_diff;

  if (argn == 1)
    {
      printf("Usage: pr.bench 1|2|3|4|5 \n");
      exit(-1);
    }

  all = 0;
  for (i = 0; i < INNER; ++i)
    {
      d[i] = (1+i);
      d[i] /= INNER;
      ld[i] = log(d[i]);
      pd[i] = pr_nats2pr(ld[i]);
      all += d[i];
    }

  s = atoi(argv[1]);

  switch (s)
    {
    case 1: /* Simple (double) addition */
#if BE_INFORMATIVE
      printf("double addition\n");
#endif
      x = 0;
      for (i = 0; i < OUTER; ++i)
        {
          for (j = 0; j < INNER; ++j)
            {
              x += d[j];
            }
        }
      result = x;
      break;

    case 2: /* In-line (macro) form */
#if BE_INFORMATIVE
      printf("pr_t macro addition\n");
#endif
      px = pr_double2pr(0.0);
      for (i = 0; i < OUTER; ++i)
        {
          for (j = 0; j < INNER; ++j)
```

```
          {
            if (px.exp >= pd[j].exp)
              {
                pr_diff = (unsigned) (px.exp - pd[j].exp);
                if (pr_diff <= PR_PRECISION)
                  px.sig += pd[j].sig * Pr_Table[pr_diff];
              }
            else
              {
                pr_diff = (unsigned) (pd[j].exp - px.exp);
                px.exp = pd[j].exp;
                if (pr_diff <= PR_PRECISION)
                  px.sig = pd[j].sig + px.sig * Pr_Table[pr_diff];
                else
                  px.sig = pd[j].sig;
              }
            if (px.sig >= PR_MANT_HIGH)
              {
                px.sig /= PR_RADIX;
                ++px.exp;
              }
          }
      }

    result = exp(pr_pr2nats(px));
    break;

  case 3: /* argument pointer form */
#if BE_INFORMATIVE
    printf("pr_t pointer addition\n");
#endif
    px = pr_double2pr(0.0);
    for (i = 0; i < OUTER; ++i)
      {
        for (j = 0; j < INNER; ++j)
          {
            pr_add_ptr(&px, &pd[j], &px);
          }
      }
    result = exp(pr_pr2nats(px));
    break;

  case 4:  /* Pure functional form */
#if BE_INFORMATIVE
    printf("pr_t functional addition\n");
#endif
    px = pr_double2pr(0.0);
```

```
      for (i = 0; i < OUTER; ++i)
        {
          for (j = 0; j < INNER; ++j)
            {
              px = pr_add(px, pd[j]);
            }
        }

      result = exp(pr_pr2nats(px));
      break;

    case 5: /* logarithmic addition */
#if BE_INFORMATIVE
      printf("logarithmic addition\n");
#endif
      x = log(1.0);    /* this is unity probability  */
      for (i = 0; i < OUTER; ++i)
        {
          for (j = 0; j < INNER; ++j)
            {
              x = log( exp(ld[j]) + exp(x) );
            }
        }
      result = exp(x) - 1;
      break;

    default:
      printf("Please specify a number between 1 and 5.\n");
      exit(-1);
    }

  if (fabs(result - all*OUTER) >= 1)
    printf("invalid result %e should be %e", result, all*OUTER);
  return 0;
}
```

This code is written to file `pr_bench.c`.