kaldi-math.h

Go to the documentation of this file.

```
// base/kaldi-math.h

// Copyright 2009-2011  Ondrej Glembek;  Microsoft Corporation;  Yanmin Qian;
//                      Jan Silovsky;   Saarland University
//
// See ../../COPYING for clarification regarding multiple authors
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//  http://www.apache.org/licenses/LICENSE-2.0
//
// THIS CODE IS PROVIDED *AS IS* BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
// KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED
// WARRANTIES OR CONDITIONS OF TITLE, FITNESS FOR A PARTICULAR PURPOSE,
// MERCHANTABLITY OR NON-INFRINGEMENT.
// See the Apache 2 License for the specific language governing permissions and
// limitations under the License.

#ifndef KALDI_BASE_KALDI_MATH_H_
#define KALDI_BASE_KALDI_MATH_H_ 1

#ifdef _MSC_VER
#include <float.h>
#endif

#include <cmath>
#include <limits>
#include <vector>

#include "base/kaldi-types.h"
#include "base/kaldi-common.h"


#ifndef DBL_EPSILON
#define DBL_EPSILON 2.2204460492503131e-16
#endif
#ifndef FLT_EPSILON
#define FLT_EPSILON 1.19209290e-7f
#endif

#ifndef M_PI
#  define M_PI 3.1415926535897932384626433832795
#endif

#ifndef M_SQRT2
#  define M_SQRT2 1.4142135623730950488016887
#endif


#ifndef M_2PI
#  define M_2PI 6.283185307179586476925286766559005
#endif

#ifndef M_SQRT1_2
# define M_SQRT1_2 0.7071067811865475244008443621048490
#endif

#ifndef M_LOG_2PI
#define M_LOG_2PI 1.8378770664093454835606594728112
#endif

#ifndef M_LN2
#define M_LN2 0.693147180559945309417232121458
#endif

#define KALDI_ISNAN std::isnan
```

```
#define KALDI_ISINF std::isinf
#define KALDI_ISFINITE(x) std::isfinite(x)

#if !defined(KALDI_SQR)
# define KALDI_SQR(x) ((x) * (x))
#endif

namespace kaldi {

#if !defined(_MSC_VER) || (_MSC_VER >= 1900)
inline double Exp(double x) { return exp(x); }
#ifndef KALDI_NO_EXPF
inline float Exp(float x) { return expf(x); }
#else
inline float Exp(float x) { return exp(static_cast<double>(x)); }
#endif // KALDI_NO_EXPF
#else
inline double Exp(double x) { return exp(x); }
#if !defined(__INTEL_COMPILER) && _MSC_VER == 1800 && defined(_M_X64)
// Microsoft CL v18.0 buggy 64-bit implementation of
// expf() incorrectly returns -inf for exp(-inf).
inline float Exp(float x) { return exp(static_cast<double>(x)); }
#else
inline float Exp(float x) { return expf(x); }
#endif // !defined(__INTEL_COMPILER) && _MSC_VER == 1800 && defined(_M_X64)
#endif // !defined(_MSC_VER) || (_MSC_VER >= 1900)

inline double Log(double x) { return log(x); }
inline float Log(float x) { return logf(x); }

#if !defined(_MSC_VER) || (_MSC_VER >= 1700)
inline double Log1p(double x) {  return log1p(x); }
inline float Log1p(float x) {  return log1pf(x); }
#else
inline double Log1p(double x) {
  const double cutoff = 1.0e-08;
  if (x < cutoff)
    return x - 2 * x * x;
  else
    return Log(1.0 + x);
}

inline float Log1p(float x) {
  const float cutoff = 1.0e-07;
  if (x < cutoff)
    return x - 2 * x * x;
  else
    return Log(1.0 + x);
}
#endif

static const double kMinLogDiffDouble = Log(DBL_EPSILON);  // negative!
static const float kMinLogDiffFloat = Log(FLT_EPSILON);  // negative!

// -infinity
const float kLogZeroFloat = -std::numeric_limits<float>::infinity();
const double kLogZeroDouble = -std::numeric_limits<double>::infinity();
const BaseFloat kLogZeroBaseFloat = -std::numeric_limits<BaseFloat>::infinity();

// Returns a random integer between 0 and RAND_MAX, inclusive
int Rand(struct RandomState* state=NULL);

// State for thread-safe random number generator
struct RandomState {
  RandomState();
  unsigned seed;
};

// Returns a random integer between min and max inclusive.
int32 RandInt(int32 min, int32 max, struct RandomState* state=NULL);

bool WithProb(BaseFloat prob, struct RandomState* state=NULL); // Returns true
 with probability "prob",
// with 0 <= prob <= 1 [we check this].
// Internally calls Rand().  This function is carefully implemented so
```

```
// that it should work even if prob is very small.

inline float RandUniform(struct RandomState* state = NULL) {
  return static_cast<float>((Rand(state) + 1.0) / (RAND_MAX+2.0));
}

inline float RandGauss(struct RandomState* state = NULL) {
  return static_cast<float>(sqrtf (-2 * Log(RandUniform(state)))
                            * cosf(2*M_PI*RandUniform(state)));
}

// Returns poisson-distributed random number.  Uses Knuth's algorithm.
// Take care: this takes time proportinal
// to lambda.  Faster algorithms exist but are more complex.
int32 RandPoisson(float lambda, struct RandomState* state=NULL);

// Returns a pair of gaussian random numbers. Uses Box-Muller transform
void RandGauss2(float *a, float *b, RandomState *state = NULL);
void RandGauss2(double *a, double *b, RandomState *state = NULL);

// Also see Vector<float,double>::RandCategorical().

// This is a randomized pruning mechanism that preserves expectations,
// that we typically use to prune posteriors.
template<class Float>
inline Float RandPrune(Float post, BaseFloat prune_thresh, struct RandomState* state=NULL) {
  KALDI_ASSERT(prune_thresh >= 0.0);
  if (post == 0.0 || std::abs(post) >= prune_thresh)
    return post;
  return (post >= 0 ? 1.0 : -1.0) *
      (RandUniform(state) <= fabs(post)/prune_thresh ? prune_thresh : 0.0);
}


inline double LogAdd(double x, double y) {
  double diff;
  if (x < y) {
    diff = x - y;
    x = y;
  } else {
    diff = y - x;
  }
  // diff is negative.  x is now the larger one.

  if (diff >= kMinLogDiffDouble) {
    double res;
    res = x + Log1p(Exp(diff));
    return res;
  } else {
    return x;  // return the larger one.
  }
}


inline float LogAdd(float x, float y) {
  float diff;
  if (x < y) {
    diff = x - y;
    x = y;
  } else {
    diff = y - x;
  }
  // diff is negative.  x is now the larger one.

  if (diff >= kMinLogDiffFloat) {
    float res;
    res = x + Log1p(Exp(diff));
    return res;
  } else {
    return x;  // return the larger one.
  }
}

```

```cpp
// returns exp(x) - exp(y).
inline double LogSub(double x, double y) {
  if (y >= x) {  // Throws exception if y>=x.
    if (y == x)
      return kLogZeroDouble;
    else
      KALDI_ERR << "Cannot subtract a larger from a smaller number.";
  }

  double diff = y - x;  // Will be negative.
  double res = x + Log(1.0 - Exp(diff));

  // res might be NAN if diff ~0.0, and 1.0-exp(diff) == 0 to machine precision
  if (KALDI_ISNAN(res))
    return kLogZeroDouble;
  return res;
}


// returns exp(x) - exp(y).
inline float LogSub(float x, float y) {
  if (y >= x) {  // Throws exception if y>=x.
    if (y == x)
      return kLogZeroDouble;
    else
      KALDI_ERR << "Cannot subtract a larger from a smaller number.";
  }

  float diff = y - x;  // Will be negative.
  float res = x + Log(1.0f - Exp(diff));

  // res might be NAN if diff ~0.0, and 1.0-exp(diff) == 0 to machine precision
  if (KALDI_ISNAN(res))
    return kLogZeroFloat;
  return res;
}

static inline bool ApproxEqual(float a, float b,
                               float relative_tolerance = 0.001) {
  // a==b handles infinities.
  if (a==b) return true;
  float diff = std::abs(a-b);
  if (diff == std::numeric_limits<float>::infinity()
      || diff != diff) return false; // diff is +inf or nan.
  return (diff <= relative_tolerance*(std::abs(a)+std::abs(b)));
}

static inline void AssertEqual(float a, float b,
                               float relative_tolerance = 0.001) {
  // a==b handles infinities.
  KALDI_ASSERT(ApproxEqual(a, b, relative_tolerance));
}


// RoundUpToNearestPowerOfTwo does the obvious thing. It crashes if n <= 0.
int32 RoundUpToNearestPowerOfTwo(int32 n);

template<class I> I  Gcd(I m, I n) {
  if (m == 0 || n == 0) {
    if (m == 0 && n == 0) {  // gcd not defined, as all integers are divisors.
      KALDI_ERR << "Undefined GCD since m = 0, n = 0.";
    }
    return (m == 0 ? (n > 0 ? n : -n) : ( m > 0 ? m : -m));
    // return absolute value of whichever is nonzero
  }
  // could use compile-time assertion
  // but involves messing with complex template stuff.
  KALDI_ASSERT(std::numeric_limits<I>::is_integer);
  while (1) {
    m %= n;
    if (m == 0) return (n > 0 ? n : -n);
    n %= m;
    if (n == 0) return (m > 0 ? m : -m);
  }
}
```

```
template<class I> I  Lcm(I m, I n) {
  KALDI_ASSERT(m > 0 && n > 0);
  I gcd = Gcd(m, n);
  return gcd * (m/gcd) * (n/gcd);
}


template<class I> void Factorize(I m, std::vector<I> *factors) {
  // Splits a number into its prime factors, in sorted order from
  // least to greatest,  with duplication.  A very inefficient
  // algorithm, which is mainly intended for use in the
  // mixed-radix FFT computation (where we assume most factors
  // are small).
  KALDI_ASSERT(factors != NULL);
  KALDI_ASSERT(m >= 1);  // Doesn't work for zero or negative numbers.
  factors->clear();
  I small_factors[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };

  // First try small factors.
  for (I i = 0; i < 10; i++) {
    if (m == 1) return;  // We're done.
    while (m % small_factors[i] == 0) {
      m /= small_factors[i];
      factors->push_back(small_factors[i]);
    }
  }
  // Next try all odd numbers starting from 31.
  for (I j = 31;; j += 2) {
    if (m == 1) return;
    while (m % j == 0) {
      m /= j;
      factors->push_back(j);
    }
  }
}

inline double Hypot(double x, double y) {  return hypot(x, y); }
inline float Hypot(float x, float y) {  return hypotf(x, y); }




}  // namespace kaldi


#endif  // KALDI_BASE_KALDI_MATH_H_
```