Обработка изображений с плавающей точкой

Evgeny.Latkin@Itseez.com

2016, Февраль 4

Про что эта лекция

Цитата из одной дипломной работы (неточно):

> «…число π для Гренландского кита равно 3…»

Смысл шутки: чрезмерная точность не нужна

На этой лекции мы обсудим:

- Какая точность нужна для обработки изображений
  - Примеры вычислений с различной точностью
- Арифметика с плавающей точкой (floating-point)
- Краткий список литературы и Web ресурсов

Содержание

- Процессоры
- Смотри OpenVX
- Пример алгоритма
  - Градиенты по Собелю
  - Детектор углов Харриса
  - Какая нужна точность?
  - Как вычислить без `float`
  - Как вычислить $k \cdot \mathrm{trace}(A)^2$
  - Симуляция float на DSP
- Форматы и арифметика
  - Целые числа с насыщением
  - Fixed-point арифметика, DSP
  - Симуляция floating-point, DSP
  - Float-16 половинной точности
  - Стандартные `float` и `double`
  - Зачем нужна стандартизация
  - Float-128 и другие форматы
- Тонкости арифметики
  - Суммирование целых на DSP
  - Суммирование с floating-point
  - Нарушение ассоциативности
  - Главный источник неточности
  - Cancellation и лемма Sterbenz
  - Уточнённое суммирование
  - Точный остаток деления
  - Опции компилятора
- Литература и Web ресурсы
  - Процессоры: CPU, GPU, DSP
  - Обработка изображений
  - Плавающая точка
  - Расширения

Процессоры

Процессор	Особенности
Центральный - CPU	• Универсальный: 10-100 гига-flops • Быстрый `float`, и быстрый `double` • Быстрый `int16_t`, `int32_t`, и `int64_t`
Графический - GPU	• Специализированный: ½ - 5 тера-flops • Быстрый `float`, но `double` нет или медленный • Быстрый `int16_t` и `int32_t`, медленный `int64_t`
Сигнальный - DSP	• Экономичный: 10-100 миллиардов целых в секунду • Отсутствует или очень медленный `float`, нет `double` • Быстрый `int16_t`, неплохой `int32_t`, слабый `int64_t`

=>
Видео-вычисления на графическом или сигнальном процессоре обычно проводят с одинарной (32-бит) или даже половинной (16-бит) точностью

Смотри OpenVX

Возьмём примеры задач из OpenVX
https://khronos.org/openvx/

OpenVX это:
- Новый стандарт C/C++ интерфейса для компьютерного зрения
- Можно скачать бесплатную спецификацию и пример реализации
- Но в текущей первой версии, набор функций довольно ограничен

Ещё бесплатная библиотека
http://opencv.org/

OpenCV это:
- Обширная и популярная библиотека для компьютерного зрения

Пример алгоритма

Градиенты по Sobel

Градиенты яркости G_X и G_Y

Свёртка с фильтром Собеля

$$G_X = I * S_X$$
$$G_Y = I * S_Y$$

Включает сглаживание по Гауссу

Изображение I(x,y) полутоновое, обычно 8 бит-на-пиксель (bpp), значение яркости от 0 до 255

Используем `int16_t` или `float`

Следим за “насыщением”: пиковые значения G_X и G_Y могут не влезть в `int16_t` для большого Sobel 7x7

Фильтр Собеля

Sobel 3x3

$$S_X = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \quad S_Y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

Sobel 5x5

$$S_X = \begin{pmatrix} 1 & 2 & 0 & -2 & -1 \\ 4 & 8 & 0 & -8 & -4 \\ 6 & 12 & 0 & -12 & -6 \\ 4 & 8 & 0 & -8 & -4 \\ 1 & 2 & 0 & -2 & -1 \end{pmatrix} \quad S_Y = \mathrm{transpose}(S_X)$$

Sobel 7x7

$$S_X = \begin{pmatrix} 1 & 4 & 5 & 0 & -5 & -4 & -1 \\ 6 & 24 & 30 & 0 & -30 & -24 & -6 \\ 15 & 60 & 75 & 0 & -75 & -60 & -15 \\ 20 & 80 & 100 & 0 & -100 & -80 & -20 \\ 15 & 60 & 75 & 0 & -75 & -60 & -15 \\ 6 & 24 & 30 & 0 & -30 & -24 & -6 \\ 1 & 4 & 5 & 0 & -5 & -4 & -1 \end{pmatrix}$$

Детектор углов Харриса

Алгоритм: Harris и Stephens

Матрица A(x,y) размера 2x2

$A_{11} = \langle (G_X)^2 \rangle$

$A_{22} = \langle (G_Y)^2 \rangle$

$A_{12} = A_{21} = \langle G_X \cdot G_Y \rangle$

Усреднение <...> по окну NxN

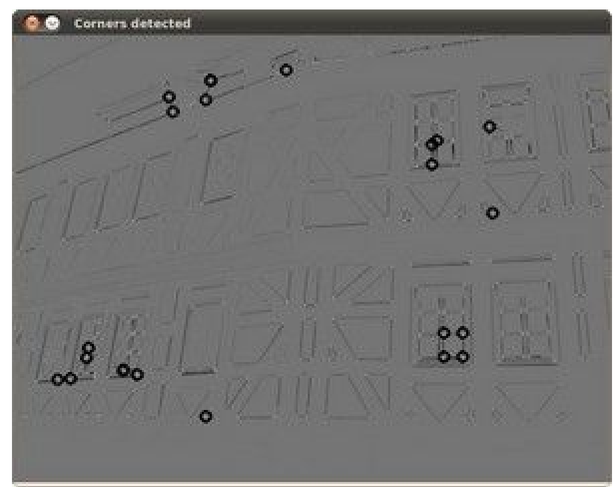
Сила отклика (“strength”)

$M_C(x,y) = \det(A) - k \cdot \mathrm{trace}(A)^2$

$V_C = M_C$, если выше порога

Эмпирически: k от 0.04 до 0.06

Результат: упорядоченный список узлов $(x,y,V_C(x,y))$ с ненулевым V_C



Пример из http://docs.opencv.org/

Какая нужна точность для алгоритма Харриса?

Достаточно 32-битового `float`

OpenVX тесты допускают ±10% в числе и “силе” найденных узлов

Трудно вычислить малые значения M_C, если det(A) и $k \cdot trace(A)^2$ близки:

$$M_C(x,y) = det(A) - k \cdot trace(A)^2$$

Но нас интересуют только большие отклики $M_C>0$, и проблемой может стать только вычитание:

$$det(A) = A_{11} \cdot A_{22} - A_{12} \cdot A_{21}$$

Но здесь нас интересуют только большие значения det(A) > 0

Справка: почему “плохо” вычитать почти одинаковые числа? Пример:

```
  3.141592653589793
- 3.141592
===================
≈ 0.000000653589 8
```

Чтобы вычислить π-*p* с одинарной точностью (7 десятичных цифр) для $p \approx \pi$, число π нужно знать с двойной точностью (16 цифр)

Взаимное поглощение старших цифр называется “cancellation”

Как вычислить отклики Харриса без `float`?

Что если нет `float`? (на DSP)

Варианты:

- Вычислять в целых числах
- Симулировать floating-point

Если G_X и G_Y целочисленные, достаточно 64-битовых целых

`int16_t` для G_X и G_Y
`int32_t` для $<G_X \cdot G_Y>$
`int64_t` для $A_{IJ} \cdot A_{KL}$

ВАЖНО: Следите за насыщением!

Значения градиентов могут не уместиться в 16 бит для Sobel 7x7

Сумма $<G_X \cdot G_Y>$ может не влезть в 32-бита из-за суммирования <...>

Пояснение к трюку с насыщением:

Хотя G_X, G_Y формально может не уместиться в 16 бит для Sobel 7x7, на практике можно обрезать такие пиковые значения, заменить их на INT16_MIN или INT16_MAX

Далее, если магнитуда градиентов велика, пиковые значения $<G_X \cdot G_Y>$ могут не уместиться в 32-бит, если окно суммирования <...> велико

Но на практике, значения G_X, G_Y обычно далеки от пиковых, и мы можем формально обрезать сумму $<G_X \cdot G_Y>$ по границам диапазона для 32-битовых целых чисел

Как вычислить $k \cdot trace(A)^2$

Если вычислять в `int64_t`, как умножить $R = trace(A)^2$ на $k \approx 0.04$

Представим k в следующем виде с 16-бит мантиссой m максимальной такой что $|m| < 2^{15}$

$$k \approx m \cdot 2^e$$

Тогда $k \cdot R$ примерно равно:

```
int64_t kR = m * (R >> -e);
```

Заметим, что $-e \geq 15$, поскольку $k < 1$

Трюк с умножением на целое плюс сдвиг, это стандартный приём для вычислений на DSP

Типичный пример: конверсия цвета

$$Y = R \cdot C_R + G \cdot C_G + B \cdot C_B$$

DSP обычно поддерживают такие операции в “железе”:

- умножение 16-битовых целых с 32-битовым результатом
- операция MAC сразу прибавит результат к аккумулятору
- деление 32-бит целого на 2^{-e} с корректным округлением

Harris через симуляцию floating-point на DSP

Можно симулировать floating-point целыми числами, здесь мантисса это целое, 16- или 32-битовое:

$$\text{value} = \text{mantissa} \cdot 2^{\text{exponent}}$$

Для алгоритма Harris/Stephens, достаточно *половинной* точности, если симулировать floating-point с 16-битовой мантиссой

Здесь “достаточно” означает: для прохождения OpenVX тестов, с допуском ± 10% по числу и силе найденных артефактов

Не все DSP поддерживают такую симуляцию достаточно быстрыми базовыми операциями

Нужна быстрая арифметика над 32-битовыми целыми числами:

- Умножение 16-битовых с 32-битовым результатом
- Сложение и вычитание, арифметические сдвиги

Как правило: симуляция будет не быстрой, и простой алгоритм как Harris часто выгоднее огрубить и вычислять в целых числах

Форматы и арифметика

Целые числа с насыщением

Используйте `int16_t`

- Обычно поддерживается DSP
- Достаточен для простых Sobel
- Вдвое быстрее int32_t на CPU

Осторожно с насыщением:

```
int32_t t;
int32_t r; // результат
int16_t s; // с насыщением
t = std::min(r, INT16_MAX);
s = std::max(t, INT16_MIN);
```

Поддерживается Intel SSE/AVX

```
__m128i s; // 128 бит = 8x16
s = _mm_adds_epi16(s, a[i]);
```

Также:

Внимательно с округлением:

```
using std::rint;
using std::ldexp;
int16_t x, y, z, n, e;
z = ((int32_t)x + y    ) >> 1;
n = ((int32_t)x + y + 1) >> 1;
e = rint(ldexp((float)x+y,-1));
```

Здесь:

z - среднее, округлённое к 0
n - округлено к ближайшему
e - к ближайшему чётному

Fixed-point арифметика на DSP

Популярные форматы

- Q7.8 внутри `int16_t`
- Q15: внутри `int32_t`

Тип Q7.8 часто используется для небольших вещественных чисел, подразумевая 8 бит после точки:

$$\text{value} = \text{целое} \cdot 2^{-8}$$

DSP обычно поддерживают Q7.8

Легко симулировать Q7.8 на CPU, например для тестирования DSP (побитовый эталонный результат)

Q7.8 иногда используют на CPU, из-за слабой поддержки float16_t (чисел “половинной” точности)

Сложение Q7.8 чисел *x+y* равно сумме их целочисленных величин (но надо следить за насыщением)

Это гораздо быстрее на CPU, чем конвертировать в 32-бит `float`

GPU могут вычислять в float16_t быстро, а перспективные - вдвое быстрее чем в формате float32_t

Симуляция floating-point на DSP

Часто достаточно 16-бит мантиссы

$$value = mantissa \cdot 2^{exponent}$$

Легко симулировать на DSP, если DSP поддерживает подсчёт битов и соответствующие сдвиги

Легко симулировать на CPU для тестирования алгоритма на DSP

Тонкости симуляции (см. пример)

- округление к чётному
- следи за насыщением
- 0 и NaN особые точки
- помни о бесконечном

```
...
  // main case
  if (value!=0 && isfinite(value)) {
    int exp;
    double val, rnd;
    val = frexp(value, &exp);
    val = ldexp(val, 15);
    exp = exp - 15;
    rnd = rint(val);
    // rounding might increase abs(rnd)
    if (rnd<INT16_MIN || rnd>INT16_MAX) {
      exp = exp + 1;
      val = val / 2;
      rnd = rint(val);
    }
    // cast to 16-bits is exact here
    exponent = (int16_t) exp;
    mantissa = (int16_t) rnd;
  }
...
```

Float-16 половинной точности

Стандартизован: IEEE-754-2008

- Нет стандартного типа C/C++
- Есть __fp16 на GCC для ARM
- Не путать с float16 для OpenCL

Назначение формата: экономия памяти (например deep learning)

Стандарт не определяет операции над 16-битовыми числами, только формат и конвертацию в/из 32-бит

Для будущих GPU анонсированы вычисления вдвое быстрее `float` (NVidia Pascal 2016)

Как закодировано внутри 16 бит:

1	5	11 бит

Здесь:

- 1 бит: знак (+ или -)
- 5 бит: порядок
- 11 бит: мантисса

Старший 11й бит мантиссы скрыт: ввиду нормализации, он всегда равен 1, и нет смысла его хранить

Экзотика: де-нормализованные, -0, бесконечность, NaN

Стандартные `float` и `double`

32-битовый `float`

1	8	24 бит

Точность около 7 десятичных цифр
Диапазон от 10^{+38} до 10^{-38}

Производительность (desktop):

- 20-200 гига-флопс на CPU
- ½ - 5 тера-флопс на GPU

Основной для GPU: достаточно точен для практически всех задач обработки сигналов/изображений

Казус: на старых GPU быстрее целых для индексов массивов

64-битовый `double`

1	11	53 бит

Точность до 16 десятичных цифр
Диапазон от 10^{+308} до 10^{-308}

Производительность (desktop):

- 1/2 от скорости float на CPU
- 1/10 от скорости float на GPU

Основной для CPU: достаточен для почти всех математических вычислений (моделирования)

Доступны и более точные числа (это иногда нужно для расчётов)

Зачем нужна стандартизация

IEEE-754 от 1985 и 2008 годов

- Стандартное кодирование
- Операции воспроизводимы однозначно (+, -, *, /, и sqrt)
- Корректное округление
- ±0, NaN, бесконечность

Ограничения

- Трансцендентные функции можно округлять не совсем корректно (±последний бит)

Дополнительно

- Десятичные числа (2008)
- Половинная точность, 16 бит
- Четверная точность, 128 бит

Пример нестандартного типа: 80-бит расширенный double в процессорах AMD/Intel x86/87

Что плохо с не стандартными?

Предсказуемый результат часто важнее точного; поэтому например Java строго оговаривает результат с точностью до последнего бита (ключевое слово strictfp, StrictMath)

Будущие библиотеки функций (sin, exp, …) видимо смогут обеспечить корректное округление результата с приемлемым быстродействием

Float-128 и другие форматы

128-битовый для точных расчётов, обычно на FORTRAN (тип: real*16)

Нет стандартного имени типа для языка C/C++, но доступен в GCC как __float128 (_Quad в Intel C++)

Реализация очень медленная, порядка в 100 раз хуже `double` (для моего ноутбука на Intel/x86)

Если точности `double` не хватает, попробуйте контроль ошибок округления (делали ниже), который замедляет “всего лишь” в 10x раз

Другие форматы:

- C# `decimal` для бухгалтерии
- Java BigDecimal (и BigInteger)
- GNU MP для multiple precision
- MPFR: корректное округление
- Double-`double` и подобные

Прочая экзотика:

- Интервалы

Кстати, избегайте интервалов: обычные методы вычислений обычно не работают если наивно заменить числа на интервалы

Тонкости арифметики

Суммирование целых на DSP

Тернарная операция

$s = x \cdot y + z$

Название
- MAC обычно для целых (DSP)
- FMA обычно для floating-point

Обычно быстрее, чем операции $t=x \cdot y$ затем $s=t+z$ по отдельности

Точнее, если аккумуляторы s и z 32-битовые для 16-битовых x и y

Рекомендуется как основной метод для фильтрации (e.g.: Sobel), и/или скалярных произведений

Пример: G_X и G_Y для Sobel 7x7

Хотя формально результат может не уместиться в диапазон для 16-битовых целых, фактически обычно умещается

Поэтому, можно вычислять в 16-битовых с “насыщением”: вычислить результат в 32-бит, и записать в 16-бит, обрезав по границам диапазона для 16-бит

Плохая новость: не все DSP поддерживают быстрый MAC

Суммирование с плавающей точкой и FMA

Библиотечная функция

```
float s = std::fma(x,y,z); // x·y+z
```

Основное применение: фильтры и/или скалярные произведения

Насколько FMA точнее? Пусть:

```
float p = a · b;
float e = fma(a·b - p);
```

Тогда a·b = p + e *точно* (если нет underflow при вычислении p=a·b)

FMA может быть быстрее, чем отдельно t=a·b и s=t+z на GPU

Забавный казус: на старых GPU, оказывается выгоднее вычислять index массива с плавающей точкой

```
//   index = i*step + offset
int index = mad24(i,step,offset);
```

На новых GPU производительность целочисленной арифметики лучше, и этот трюк не столь эффективен

Ориентировочно, FMA одинарной точности для CPU и новых GPU может быть вдвое быстрее, чем x·y+z в целых 32-битовых числах

Нарушение ассоциативности и дистрибутивности

Операции с плавающей точкой

$s = \mathtt{fl}(a + b)$
$p = \mathtt{fl}(a \cdot b)$

Здесь fl(...) округляет точное a+b до ближайшего числа, представимого как число с плавающей точкой

Операции с плавающей точкой не всегда ассоциативные, то есть:

$$(a + b) + c \neq a + (b + c)$$
$$(a \cdot b) \cdot c \neq a \cdot (b \cdot c)$$

Также нет дистрибутивности

$$(a + b) \cdot c \neq ab + bc$$

Тонкости округления к ближнему:

Если ближайших два, выбираем у которого последний бит мантиссы равен нулю (округление к чётному)

Округление к чётному отличается от “школьного”, например: округляя 2.5 до чётного целого получим 2, а не 3 как при школьном округлении

Моды округления (к целому):

```
y = rint(x);    // к чётному
y = trunc(x);   // к нулю
y = floor(x);   // к -∞
y = ceil(x);    // к +∞
```

Суммирование - главный источник неточности

Теряются младшие разряды

Пример: в десятичных числах половинной точности (3 цифры), второе слагаемое полностью теряется при округлении

$$\mathtt{fl}(1.23 + 0.00456) = 1.23$$

Решение: аккумулятор большей точности - например 32-битовая сумма для 16-битовых данных

float16_t $a_1, \ldots, a_N$;
float32_t $S = a_1 + \ldots + a_N$;
float16_t s = округлить(S);

Если слагаемые упорядочены, как:

$$|a_1| \geq \ldots \geq |a_N|$$

Тогда лучше складывать от меньших к большим, как:

$$a_N + \ldots + a_1$$

Если упорядочение не известно, промежуточный результат можно накапливать с “guard” цифрами, обычно с удвоенной точностью

См. далее про double-double, etc

Cancellation и лемма Sterbenz

Cancellation это потеря старших значащих цифр при вычитании

Пример, десятичные с 3 цифрами:

π - e ≈ 3.14 - 2.72 = 0.42

Заметим, что:

- Лишь 2 значащих цифры в 0.42
- Однако, этот результат точный

Лемма (Sterbenz): Разность s = a-b точная, если a и b различаются не более, чем в 2 раза

Cancellation *выявляет* неточность, накопленную ранее внутри a и b

Лемма Sterbenz верна лишь если поддерживаются суб-нормальные

s	exp	mantissa

При минимальном значении exp, mantissa не нормализована, и в ней нет скрытого бита 1

Такие очень маленькие числа называют subnormal или denormal

IEEE-754 подерживает denormal

Но иногда для скорости, результат сбрасывают в 0 вместо аккуратного вычисления denormal (flush-to-zero)

Уточнённое суммирование

Суммируем ошибку

```
float s=0, e=0;
for (n=0; n<N; n++) {
    e += ошибка(s + a[n]);
    s += a[n];
}
```

Используйте s+e вместо double, это может быть быстрее на GPU

Как вычислить ошибку a+b, при условии что |a|≥|b| (Dekker)

```
s = a + b;    // примерно
t = s - a;    // точно
e = t - b;    // точно
```

Для вычислений на CPU, и на GPU с быстрым `double`, подобный трюк можно использовать для уточнения суммы чисел двойной точности

Подробности можно нагуглить по ключевым словам double-`double` и floating-point exact transforms

Пример exact transform:

s + e = a + b *точно*

На практике, для точной суммы обычно применяют 6-стадийный алгоритм Деккера без ветвлений

Точный остаток деления

Точный остаток от q=a/b

```
float q = a / b;
float r = fma(a - q·b);
```

Тогда a = qb+r *точно* (если нет underflow при вычислении q·b)

Эта же магия работает и для квадратного корня, если sqrt(...) округляет результат корректно

```
float q = sqrt(a);
float r = fma(a - q·q);
```

Тогда $a = q^2+r$ *точно* (без underflow)

Вспомним также точное умножение

```
float p = a · b;
float e = fma(a·b - p);
```

Здесь p+e = a·b точно только если a·b вычисляется без underflow

Без underflow, разница ab - p точно представима в плавающих числах

В общем случае, можно вычислять интервал $[e^-, e^+]$ для остатка ab - p

Впрочем, для режима flush-to-zero интервал будет тривиален

Опции компилятора

Exact transforms чувствительны к оптимизациям компилятора:

- Лемма Sterbenz неверна если запрещены sub-normal числа (режим flush-to-zero)
- Формулу err = ((a+b)-a)-b могут заменить на просто ноль, если включена не-strict арифметика
- Функцию e=fma(a·b - p) могут заменить на t=a·b и e=t-p, что даст e=0 для p=fl(a·b)

Компилируйте:

cl /fp:strict /arch:AVX2 …

gcc -mfma ...

Заметим, что:

Intel поддерживает FMA только начиная с процессоров Haswell

Библиотечная функция std::fma() может быть очень медленной

GNU C/C++ заменит fma(a·b - p) если не указать опцию -mfma

Microsoft C/C++ предполагает ассоциативность без /fp:strict

Опция flush-to-zero может заметно ускорять floating-point вычисления

Литература и Web ресурсы

Процессоры: CPU, GPU, DSP

- https://msdn.microsoft.com/ru-ru/library/26td21ds.aspx
- https://software.intel.com/sites/landingpage/IntrinsicsGuide/

  Справочник по “intrinsics”, то есть расширениям для компилятора C/C++, чтобы полнее задействовать команды центрального процессора (CPU).

- https://khronos.org/opencl/

  Язык OpenCL для программирования графических процессоров (GPU). Не зависимый от конкретной платформы, и набирающий популярность.

- http://dspguide.com/

  Доступная книжка о программировании сигнальных процессоров (DSP), S. Smith, “The Scientist and Engineer's Guide to Digital Signal Processing”

Обработка изображений

- http://opencv.org/
- https://khronos.org/openvx/
- Robert Laganière. OpenCV 2 Computer Vision Application Programming Cookbook. 2011

  Наш основной пример: библиотека OpenCV и интерфейс OpenVX для компьютерного зрения, книжка про то как использовать OpenCV.

- Keith Jack. Video Demistified: A Handbook for the Digital Engineer.

  Толстая книжка про видео: интерфейсы, форматы, компрессия, и пр. Помогает лучше понять, что делают функции из OpenCV и OpenVX.

- https://en.wikipedia.org/wiki/Integrated_Performance_Primitives

  Высоко производительная библиотека для CPU, доступна бесплатно для исследования. OpenCV умеет вызывать IPP для ускорения расчётов.

Плавающая точка

- https://en.wikipedia.org/wiki/Floating_point
- https://en.wikipedia.org/wiki/Fixed-point_arithmetic

Краткий но вполне достаточный для общего понимания обзор форматов чисел с плавающей и фиксированной точкой, с историческим экскурсом.

- http://itlab.unn.ru/?dir=592
- Jean-Michel Müller, et al. Handbook of Floating-Point Arithmetic. 2010
- Jean-Michel Müller. Elementary Functions Algorithms and Implementation.

Курс лекций, PDF и видео, и две нетолстые книжки про внутреннюю кухню плавающей точки, как обеспечить корректность результатов.

- http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Знаменитая статья: “What Every Computer Scientist Should Know About Floating-Point Arithmetic”, 1991

Расширения

- https://en.wikipedia.org/wiki/Quadruple-precision_floating-point_format

  Другие форматы чисел с плавающей точкой, расширенной точности.

- https://en.wikipedia.org/wiki/GNU_MPFR

  Популярная библиотека для вычислений с произвольной точностью и корректным округлением результатов операций и функций.

- http://nsc.ru/interval/index.php

  Сайт “Интервальный анализ и его приложения”. Содержит коллекцию ссылок на ресурсы по интервалам, книжки, конференции, и software.

- https://sites.google.com/site/yevgenylatkin/

  Мой сайт про “twofold” арифметику со встроенным контролем ошибок.

Вопросы?