ПЛАВАЮЩИЕ ЧИСЛА

Представления чисел с плавающей точкой и операции над ними.
Точность вычислений. Поиск корней уравнений

К. Владимиров, Intel, 2022
mail-to: konstantin.vladimirov@gmail.com

Мотивирующий пример: определитель

- Что такое определитель?
- Допустим все коэффициенты целые. Может ли определитель не быть целым?
- Как вы подсчитаете определитель для матрицы $\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$?
- А что насчёт $\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$?
- Как вы запрограммируете такое вычисление?
- Какая будет алгоритмическая сложность предлагаемого решения?
- Как вы будете тестировать ваш алгоритм?

Problem DP – свойства определителя

- На входе: желаемый определитель и размер квадратной целочисленной матрицы
- На выходе: матрица $N \cdot N$ с заданным определителем
- Пример
  - Вход: `-5 2`
  - Выход: `2 3 4 2 1`
- Полезные свойства
  - Определитель треугольной матрицы равен произведению диагональных элементов
  - Сложение строки со строкой умноженной на коэффициент не меняет определитель

Вычисление определителя

- Разложение по строке (столбцу) это рекурсивное вычисление определителя меньшей матрицы и умножение на элемент строки с правильным знаком

$$det\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = a_{11} \cdot det\begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} - a_{12} \cdot det\begin{pmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{pmatrix} + \cdots$$

- Оцените алгоритмическую сложность этого метода?

Метод Гаусса-Жордана

- Основан на свойстве определителя, что прибавление столбца или вычитание столбца ничего не меняет

$$det\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = det\begin{pmatrix} a_{11} & a_{12} \\ 0 & a_{22} - a_{12}\frac{a_{21}}{a_{11}} \end{pmatrix}$$

- У треугольной матрицы определитель равен произведению элементов на главной диагонали
- Напишите это алгоритмом используя промежуточный тип float
- Это показывает, что даже для матриц с целочисленными значениями, выгодно производить промежуточные вычисления в вещественных числах

Внезапная проблема

- Что если первый элемент слишком маленький?
- Примените ваш алгоритм к $\begin{pmatrix} 0.00001 & 100.00 \\ 100.00 & 100.00 \end{pmatrix}$
- Попробуйте увеличивать и уменьшать первый элемент и наблюдать размер ошибки
- Добро пожаловать в мир плавающей арифметики

Алгоритм D -- full pivoting

```
// Input: M -- matrix NxN

for (current = 0; current < N; ++current) {
  // max from (N - current)x(N - current) submatrix
  (max, col, row) = max_submatrix_element(M, current);
  swap_columns(M, current, col);
  swap_rows(M, current, row);
  pivot = element(M, current, current);
  if (pivot == 0)
    exit(-1); // elimination not possible
  eliminate(M, current, pivot);
}

// Output: product of main diagonal
```

Problem DT – определитель матрицы

- На входе матрица в обычном представлении (можете тестировать на результатах проблемы DP)
- На выходе её определитель, подсчитанный методом Гаусса-Жордана с полным пивотингом

Немного о работе компьютера

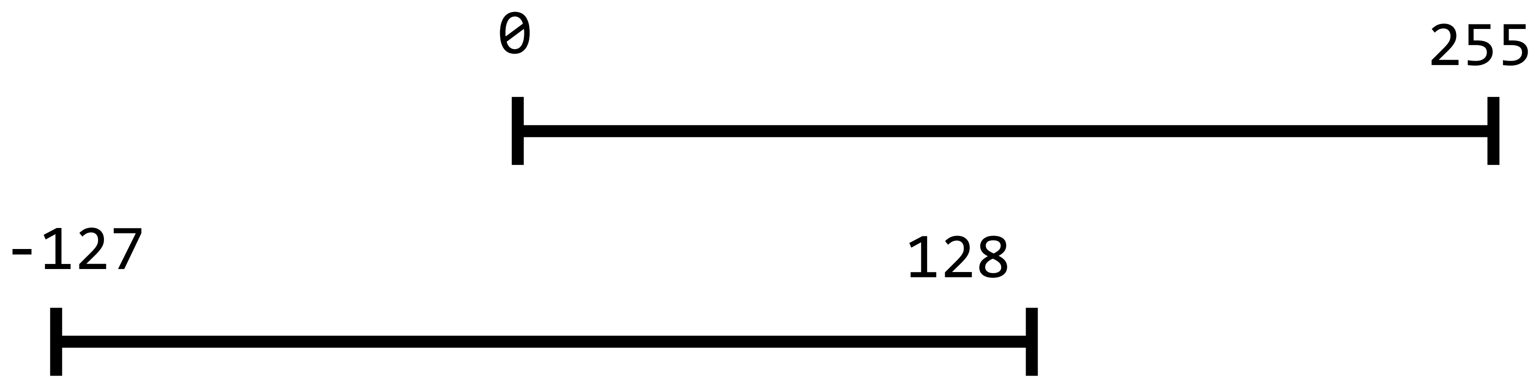
- Память позволяет хранить ограниченное число бит
- Каждая операция в компьютере производится над ограниченным числом бит

```
00000000 <_add>:
int add(int x, int y) {
  return x + y;
   0: 8b 44 24 08     mov  eax, DWORD PTR [esp+0x8]
   4: 03 44 24 04     add  eax, DWORD PTR [esp+0x4]
```

- Естественный способ трактовать ограниченное число бит: как натуральное число

От натуральных к целым

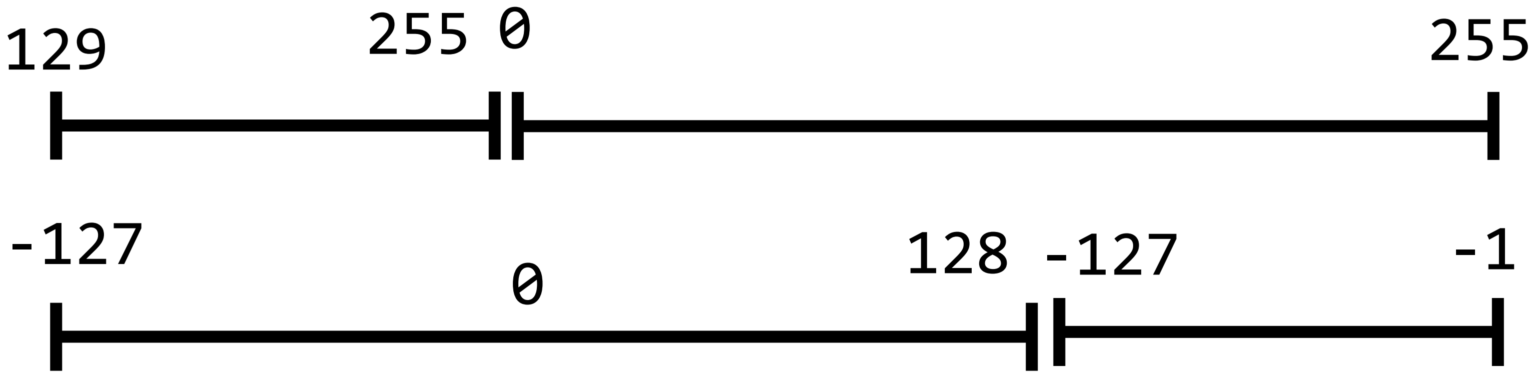
- Естественный способ думать об ограниченных целых числах: как о сдвинутых вправо натуральных



- Но поскольку диапазон всё равно ограничен… Например, сколько будет в беззнаковых числах `0u - 1u`?

От натуральных к целым

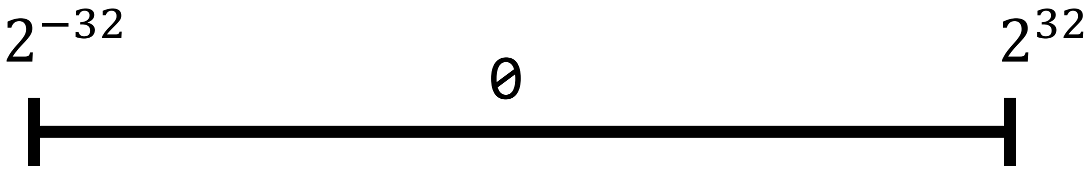
- Естественный способ думать об ограниченных целых числах: как о сдвинутых вправо натуральных



- Целые числа обычно закодированы как верхний диапазон натуральных в обратном порядке (поэтому у них всегда выставлен верхний "знаковый" бит)

Обсуждение

- Как закодировать вещественные числа?
- Первая идея кодировка с фиксированной точностью



- Эта идея иногда находит применения, но в целом она очень плоха:
  - маленькие диапазоны чисел (64-битное плавающее число не больше чем 2^{32})
  - низкая точность: размер шага не больше, чем 2^{-32} это слишком крупный шаг для многих практических применений
- Можно генерировать вещественное число бит за битом

Плавающая точность

- Для научных вычислений принято приближать вещественные числа рациональными, используя идею плавающей точки
- Например мы договариваемся, что у нас есть 8 значащих разрядов. Тогда с плавающей точкой возможны числа: `1024561`, `10245.61`, `10.24561`, `1.024561`
- Это было осознано довольно рано и после ряда попыток разной успешности, стандартизовано в 1985 году (см. $[IEEE]$)
- Сейчас это фактический стандарт, от которого крайне редко отступают

Представление с плавающей точкой

- Представление $\pm 1.frac * 2^{exp-127}$ при этом `(exp > 0) && (exp < 255)`

s	exponent								fractional part of mantissa																						
0	0	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31								23																							0

- Например на рисунке представлено какое-то число
- `exp – 127 = 124 – 127 = -3`
- `mantissa = +1.01b =` $1 + \frac{1}{4}$ `= 1.25f`
- `Вместе: 1.25 *` 2^{-3} `= 0.15625f`

*значения `0` и 255 зарезервированы на особые случаи

Представление с плавающей точкой

- Представление $\pm 1.frac * 2^{exp-127}$ при этом `(exp > 0) && (exp < 255)`

s	exponent								fractional part of mantissa																						
1	1	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31								23																							0

- Например на рисунке представлено какое-то число
- `exp – 127 = 129 – 127 = 2`
- `mantissa = -1.011b =` $-\left(1+\frac{1}{4}+\frac{1}{8}\right)$ `= -1.375f`
- Вместе: `-1.375f *` 2^2 `= -5.5f`

Обсуждение

- У такого формата есть один недостаток: нельзя представить `0`
- Найдите самое близкое к нулю число

s	exponent								fractional part of mantissa																						
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31								23																							0

Обсуждение

- У такого формата есть один недостаток: нельзя представить `0`
- Найдите самое близкое к нулю число

s	exponent								fractional part of mantissa																						
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31								23																							0

- Имеем 2^{-126} это довольно большое число и это точно не ноль

Денормализованные числа

- Специальное значение `exp = 0` соответствует денормализованным числам
- Нормализованные: $\pm 1.frac * 2^{exp-127}$ при этом `exp > 0`
- Денормализованные: $\pm 0.frac * 2^{-126}$ при этом `exp == 0`

s	exponent								fractional part of mantissa																						
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31								23																							0

- Благодаря этому число, состоящее изо всех нулей это ноль, что интуитивно правильно.
- Контринтуитивно здесь то, что возможен `-0.0`, отличающийся от `+0.0` при побитовом сравнении

Бесконечности

- Экспонента, состоящая изо всех единиц ($e = 255$) отображает $\pm\infty$

s	exponent							fractional part of mantissa																						
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31								23																							0

s	exponent							fractional part of mantissa																						
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31								23																							0

Концепция not-a-number

- Число, представляющее собой неопределённость (например результат деления нуля на ноль) называется NaN

s	exponent								fractional part of mantissa																						
0	1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
31								23																							0

- NaN это exp=255 и любая ненулевая fraction
- Первый бит fraction отличает qNaN от sNaN, но это не часть семантики языка
- Сравнение чего угодно с NaN даёт false, в т. ч. NaN не равен сам себе (и поэтому он не число)

Problem EX – работа с битами чисел

- Вам дано число с плавающей точкой типа float (мантисса 23 нижних бита)
- Вы должны получить из него другое число с плавающей точкой, инвертировав все нечётные биты мантиссы (нулевой бит считается чётным) и напечатать с точностью до пятого знака после запятой

Упражнения

- Запишите в формате floating-point число `1.0f`
- Число `0.1f` не представимо в формате floating-point точно. Постройте лучшее возможное приближение. Насколько оно хорошо?
- Число $\frac{1}{3}$ тоже нельзя точно представить. Какое будет лучшее приближение этого числа?
- Охарактеризуйте все числа, которые можно представить точно. Например что вы скажете о числе `0.0361328125f`?
- Какое расстояние между самым большим по модулю нормализованным числом и следующим за ним?

Быстрый приближённый логарифм

- В следующей таблице приведены значения чисел и их логарифмов по основанию 2

Число	**Представление**	**Представление минус `0x3f800000`**	**Логарифм**
`1.0f`	`0x3f800000`	`0x00000000`	`0.0f`
`2.0f`	`0x40000000`	`0x00800000`	`1.0f`
`4.0f`	`0x40800000`	`0x01000000`	`2.0f`

- Из этого следует интересная формула
- `log(x) ≃ ([x as bits] - 0x3f800000) / 0x00800000`

Обсуждение

- Разница между приведением и трактовкой как биты очень велика

```
float x = 1.0f;
unsigned i = (unsigned) x; // i == 1
```

- Это приведение (cast)

```
float x2 = 1.0f;
unsigned i2 = *(unsigned *) &x2; // i == 0x3f800000
```

- Это трактовка как биты (reinterpretation)

Strict aliasing

- Правилами языка запрещён алиасинг двумя разными типами на один объект

```
float x2 = 1.0f;
unsigned *pi2 = (unsigned *) &x2; // aliasing
```

- Здесь одна ячейка в памяти имеет два имени для доступа и это очень плохо
- Можно выкрутится чем-то вроде:

```
unsigned as_uint(float f) {
  unsigned u; memcpy(&u, &f, sizeof(unsigned)); return u;
}

unsigned i2 = as_uint(x2);
```

- Здесь у нас не будет strict aliasing violation, т. к. нас интересует значение

Быстрые приближения

- Можно реализовать быстрый логарифм на базе формулы

```
log(x) ≃ (float) (as_uint(x) - 0x3f800000) / (float) 0x00800000
```

- Давайте потратим некоторое время на анализ того как это вообще может работать

Быстрые приближения

- Можно реализовать быстрый логарифм на базе формулы

`log(x) ≃ (float) ([x as bits] - 0x3f800000) / (float) 0x00800000`

- Реализуйте также быстрое возведение двойки в данную степень

2^x `≃ [((unsigned) (x * (float)0x00800000) + 0x3f800000) as float]`

- Подобная же магия возможна и для квадратного корня

$\sqrt{x}$ `≃ [(([x as bits] >> 1) + (0x3f800000 >> 1)) as float]`

- Сравните с функциями логарифма, возведения в степень и извлечения корня стандартной библиотеки. Имеет ли на вашей системе эта магия реальный смысл?

Концепция ulp

- "Unit in the last place" это расстояние между двумя последовательными числами с плавающей точкой
- Например посчитаем `ulp(1.0f)`

```
float d0, d1;
d0 = 1.0f;
d1 = nextafterf(d0, d0 + 1.0f);
printf("%.8f", d1 - d0); // на экране 0.00000012
```

- Все вычисления над парой чисел `z = x (op) y` должны быть округлены в пределах `0.5*ulp(z)`

Важность округления

- Результат, полученный после арифметической операции внутри ulp, должен быть округлён к ближайшему представимому значению
- Округлять можно вверх, вниз, к нулю и к ближайшему
- Для выставления метода округления используется функция `fesetround`

```
float a = 1.0, b = 3.0;
```

```
fesetround(FE_UPWARD);     assert(as_uint(a / b) == 0x3eaaaaab);
fesetround(FE_DOWNWARD);   assert(as_uint(a / b) == 0x3eaaaaaa);
fesetround(FE_TONEAREST);  assert(as_uint(a / b) == 0x3eaaaaab);
fesetround(FE_TOWARDZERO); assert(as_uint(a / b) == 0x3eaaaaaa);
```

Problem RP – верхняя и нижняя границы

- Пользователь вводит числитель и знаменатель дроби
- На выходе верхняя и нижняя аппроксимации при представлении в формате `float` как два шестнадцатиричных числа: экспонента и дробная часть мантиссы
- Если возможно точное представление, они должны совпадать

```
input:

1 3

output:

0x7d 0x2aaaaa 0x7d 0x2aaaab
```

см. также https://www.h-schmidt.net/FloatConverter/IEEE754.html

Разная точность

- В языке C поддержаны три уровня точности: float, double, long double
- float это обычные 32-битные плавающие числа
- double это 64-битные плавающие числа
- long double часто совпадает с double, но иногда (в таких компиляторах как gcc) оно реализуется через extended-precision 80-битные числа

Тип	**bits in exponent**	**bits in fraction**	**significant decimal digits**
float	8	23	7-8
double	11	52	15-16
long double*	16	64	20-21

*не всегда, см. текст слайда

Упражнение

- Первые знаки числа π в двоичной записи

```
11.00100100001111110110101010001000100001011010001100001000110
10011000100110001100110001010001011100000001101110000011100110
10001001010010000001
```

- Запишите наиболее близкое к π представление с одинарной и с двойной точностью

Основные правила

- Работа с плавающими числами имеет свою специфику. В частности следует
  - Избегать сравнения на равенство
  - Быть очень аккуратным с ошибками сложения
  - Учитывать конечный размер плавающих чисел
  - Иметь в виду, что операции над числами не всегда возвращают числа
- Далее каждый из этих пунктов будет разобран детально

Избегайте сравнивать на равенство

- Следующий код не слишком сложен и сравнение должно выполняться, но увы

```
d1 = 10.0;
d2 = sqrt(d1);
d3 = d2 * d2;

if (d1 == d3) {
  // сюда мы можем и не попасть (зависит от округления)
}
```

- Правильный способ сравнивать: в пределах некоего ε

```
if (fabs(d1 - d3) < tolerance) {
```

- Хорошая ли идея выбирать `tolerance == ulp(result)`?

Аккуратнее с ошибками сложения

- В следующем примере мы пытаемся вычислить как можно более точную производную, измельчая шаг до предела `double` диапазона

```
double h, cosval;

for (i = 1; i < 20; ++i) {
  h = pow(10.0, -i);
  cosval = (sin(1.0 + h) - sin(1.0)) / h;
  printf("%d:\t%.15lf\n", i, cosval); // всё лучше и лучше?
}

cosval = cos(1.0);
printf("True result: %.15lf\n", cosval);
```

- Результаты при этом могут быть неожиданны (показать demo)

Помните о конечности диапазона

- Даже числа одинарной точности предоставляют гигантские диапазоны. Но конечные
- Это заметно при сложении с очень большими числами

```
f = 16777216.0f; // 2^24
nextf = f + 1.0f; // побитово не отличается от f
```

- И при сложении с очень малыми

```
fone = 1.0f;
feps = 0.00000005f;
fenext = fone + feps; // побитово не отличается от fone
```

- И тут встаёт вопрос: а с насколько маленькими можно складывать?

Ваш результат это не всегда число

- Следующий код позволяет получить бесконечность за конечное время

```
double d = 1.79e+308;
double infd = 2.0 * d;

printf("d: %le\tinfd: %le\n", d, infd);
```

- Дальше над результатом будут работать другие правила (например умножение на `0` даст не `0`, а NaN)

Сложение внутри диапазона

- По определению FLT_EPSILON (и её аналог DBL_EPSILON) это минимальная константа, такая, что

`1.0f + FLT_EPSILON != 1.0f`

- Обычно FLT_EPSILON это число около `1e-5`
- Эта константа указывает сколько порядков внутри диапазона. То есть числа порядка `1e-22` безопасно складывать с числами где-то от `1e-27` и до `1e-17`. Но это не совсем линейно. Проведите эксперименты самостоятельно!
- Сам диапазон задаётся константами FLT_MIN (DBL_MIN) и FLT_MAX (DBL_MAX) где имеется в виду минимум и максимум по модулю

Логарифмы для больших чисел

- Сложение важно, потому что часто для больших чисел им можно заменить умножение. Например пусть хочется точно подсчитать $\frac{200!}{190! \cdot 10!}$
- Определим функцию*

```
double log_fact(int n) {
  double sum = 0.0;
  for (int i = 2; i <= n; ++i) sum += log((double)i);
  return sum;
}
```

- Теперь можно подсчитать непосредственно

```
x = exp(log_fact(200) – log_fact(190) – log_fact(10));
```

*функция log_fact несколько наивна. На практике lgamma даёт лучшую точность

Обсуждение

- Допустим вы хотите найти корень уравнения

$x^2 * \sin(x) - 5x + 7 = 0$

- Вы знаете, что он лежит где-то в диапазоне от -3 до 3
- Как вы его найдёте?

Обсуждение

- Допустим вы хотите найти корень уравнения

$x^2 * \sin(x) - 5x + 7 = 0$

- Вы знаете, что он лежит где-то в диапазоне от -3 до 3
- Как вы его найдёте?
- В данном случае нам повезло: $f(-3) > 0$ и $f(3) < 0$
- Для решения можно воспользоваться дихотомией: на каждом шаге делить отрезок пополам и если там значение совпадает по знаку с левым, то брать правый интервал и наоборот
- Это очень похоже на бинарный поиск в сортированном массиве

Problem DH: дихотомия уравнений

- Дано уравнение

$$x^2 * \sin(x) - 5x + 7 = 0$$

- Найдите делением пополам корень, лежащий в интервале от `-3` до `3`
- Попробуйте теперь найти один из семи корней, лежщих в интервале от `-13.5` до `13`
- Творческая задача: сможете ли вы написать программу, которая находит все семь? Как вы сделаете чтобы она работала в общем случае?
- Проверить численный ответ можно здесь: https://www.wolframalpha.com/input/?i=x%5E2*sin(x)+-+5x+%2B+7+%3D+0

Алгоритм SC – метод Риддерса

- Метод Риддерса обычно сходится несколько быстрее, чем дихотомия (математические подробности в $[Numrec]$)

```
typedef double (*func_t)(double x);

double fsgn(double x) { return signbit(x) ? -1.0 : 1.0; }

double secant(func_t f, double xleft, double xright) {
  assert(fsgn(f(xleft)) != fsgn(f(xright)));
  // далее в цикле:
  // xmid = (xleft + xright) / 2.0;
  // fl = f(xleft); fr = f(xright); fm = f(xmid);
  // xnew = xmid + (xmid-xleft) * fsgn(fl - fr) * fm / sqrt(fm * fm – fl * fr);
  // заменяем xleft = xnew или xright = xnew в зависимости от знака f(xnew)
  // проверяем условие выхода из цикла fabs(f(xnew)) < precision
  return xnew;
}
```

Problem EC – исследование сходимости

- Замерьте количество итераций методом Риддерса и дихотомией для уравнения

$$x^2 * \sin(x) - 5x + 7 = 0$$

- Попробуйте разные начальные интервалы
- Попробуйте float и double precision
- Подтверждают ли ваши результаты теоретическое превосходство метода?

Обсуждение

- Рассмотрим уравнение

$$x^2 + e^x - 0.827185 = 0$$

- У него два действительных корня, но довольно сложно выбрать два значения, в которых функция принимала бы разные знаки (попробуйте!)
- Что делать в этом случае?

Алгоритм N – метод Ньютона

```
struct func_deriv { double func; double der; };

// возвращает значение функции и производной в точке x
typedef struct func_deriv (*fder_t) (double x);

double newton(fder_t f, double x) {

  // реализуйте самостоятельно
```

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

```
  return x;
}
```

Problem EN: решение методом Ньютона

- Даны два уравнения

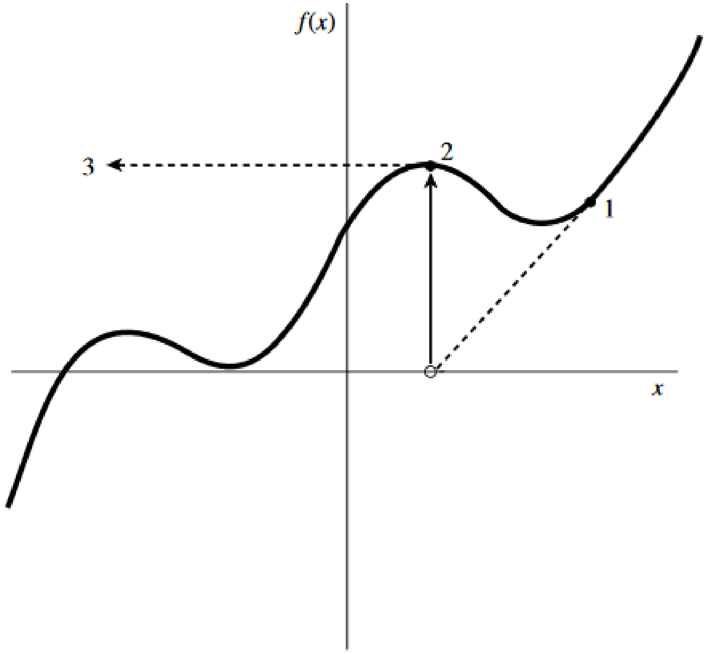
$$x^2 + e^x - 0.827185 = 0$$

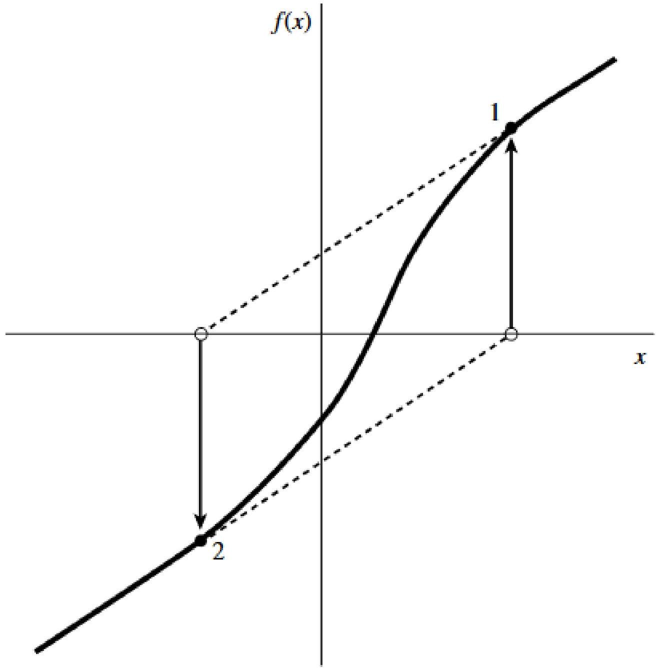
$$x^2 * \sin(x) - 5x + 7 = 0$$

- Решите оба используя алгоритм N
- Были ли у вас какие-нибудь сложности со вторым?

Проблемы со сходимостью

- У метода Ньютона есть проблемы со сходимостью (картинки из $[Numrec]$)





Фракталы

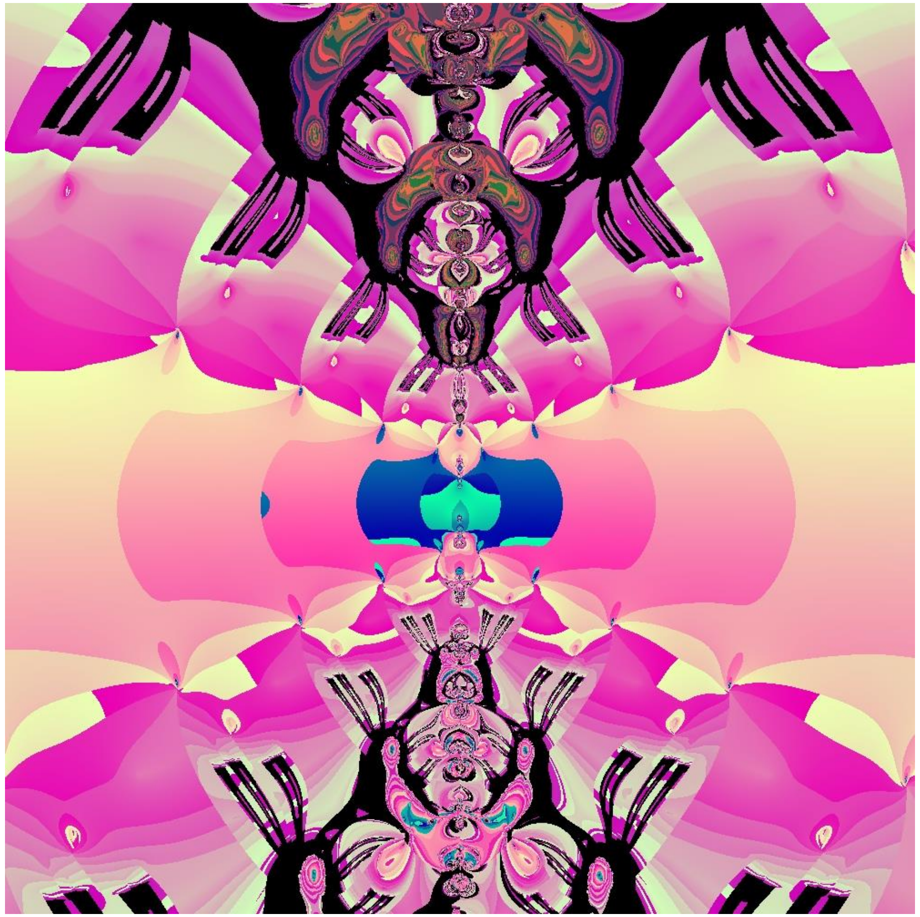
- Несмотря на трудности которые создают проблемы со сходимостью, они же порождают фрактальную структуру
- Например рассмотрим в комплексных числах уравнение

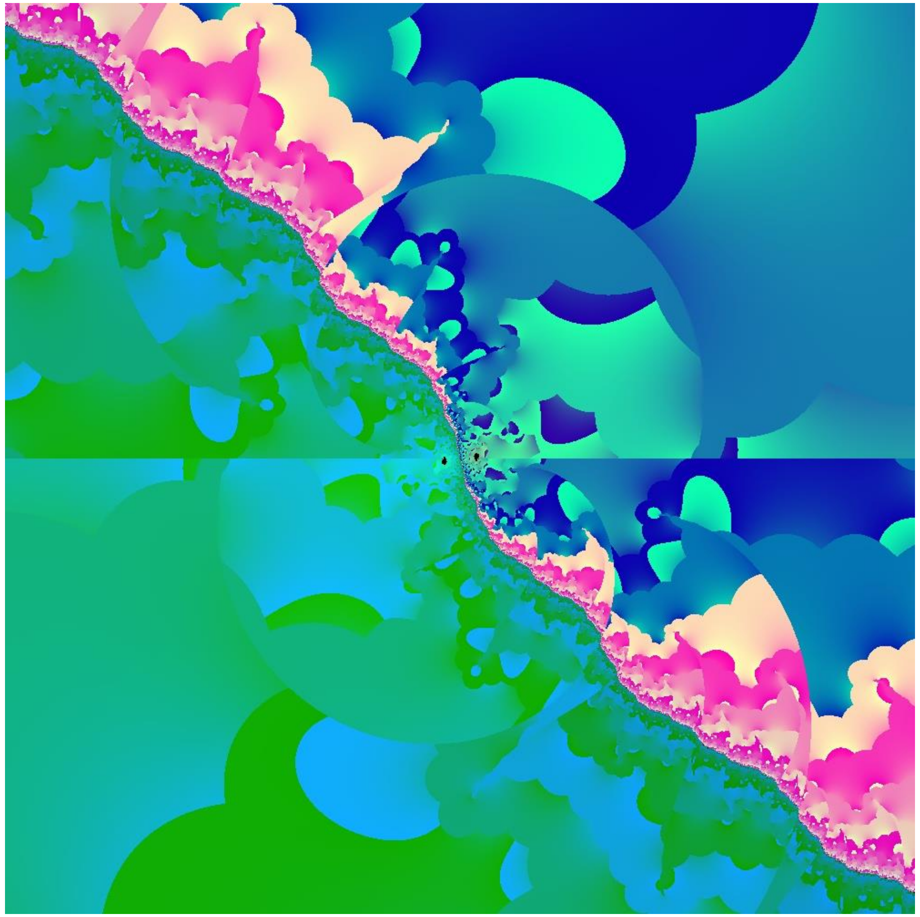
$$z^3 - 1 = 0$$

- Для него есть такие z_0 для которых метод Ньютона сходится и такие, для которых нет
- Из-за нестабильного поведения около локальных максимумов, область сходимости образует самоподобную кривую, то есть собственно фрактал
- Рисунки таких фракталов на комплексной плоскости могут быть крайне красивы

Подробнее: https://en.wikipedia.org/wiki/Newton_fractal

Примеры для более сложных функций





Источник: https://vk.com/fracgen

Problem RI – инверсный корень

- Метод Ньютона часто используется, чтобы вычислять функции
- Действительно, пусть дано a и надо найти $x = \frac{1}{\sqrt{a}}$
- Это всё равно, что решить уравнение $f(x) = \frac{1}{x^2} - a = 0$
- Напишите функцию

```
double inv_sqrt(double a);
```

- Не используйте при реализации стандартную функцию sqrt и деление, воспользуйтесь методом Ньютона
- Как вы будете тестировать вашу функцию?

Problem AN – улучшение приближений

- Возьмите быстрые приближения из Problem AQ и улучшите каждое из них дополнительным шагом метода Ньютона
- Существенно ли это улучшит точность по сравнению с библиотечными функциями?

Магический инверсный корень

- В качестве приближённого решения предыдущей проблемы будет работать следующая магическая процедура

```
float magic_inv_sqrt (float y) {
  double x2 = 0.5f * y;
  long i = *(long *) &y;
  i = 0x5f3759df - (i >> 1); // Magic!
  y = *(float *) &i;
  y  = y * (1.5f - (x2 * y * y)); // one additional Newton step
  return y;
}
```

- Оцените разницу в точности с вашим решением для Problem RI
- Догадываетесь ли вы как это работает?

Подробнее: https://en.wikipedia.org/wiki/Fast_inverse_square_root

Литература

- $[C11]$ ISO/IEC – "Information technology – Programming languages – C", 2011
- $[IEEE]$ ANSI/IEEE Std 754 – "Standard for Binary Floating-Point Arithmetic", 1985
- $[K\&R]$ Brian W. Kernighan, Dennis Ritchie – The C programming language, 1988
- $[Numrec]$ W. Press, S. Teukolsky – Numerical Recipes in C, 2nd edition, 1992
- $[TIPS]$ John D. Cook – Five Tips for Floating Point Programming, 2014
- $[TRICKS]$ James F. Blinn – Floating point tricks, 1997