Experimentos usando Pi

https://github.com/guilhermeivo/computing-projects/

guilhermeivo

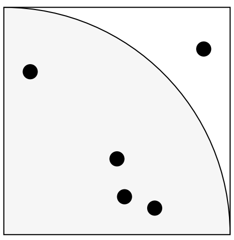
14 de março de 2025

Queremos implementar algoritmos para calcular o valor de π usando métodos computacionais.

Pi usando Monte Carlo

O método de Monte Carlo é uma técnica de amostragem estatística, teve seu desencadeamento após o desenvolvimento do *ENIAC*[1], primeiro computador eletrônico, desenvolvido durante a Segunda Guerra Mundial, para lidar com grandes cálculos. O método baseia-se em números "aleatórios" para resolver os seus problemas que é aplicada com sucesso em diversos problemas contemporâneos [3].

`monteCarlo()` O problema consiste em montar um quadrado de comprimento 1, e $\frac{1}{4}$ de uma circunferência inscrita nesse quadrado e de forma aleatório disparar pontos dentro do espaço.



$$A = \pi \cdot R^2 \implies A = \pi$$

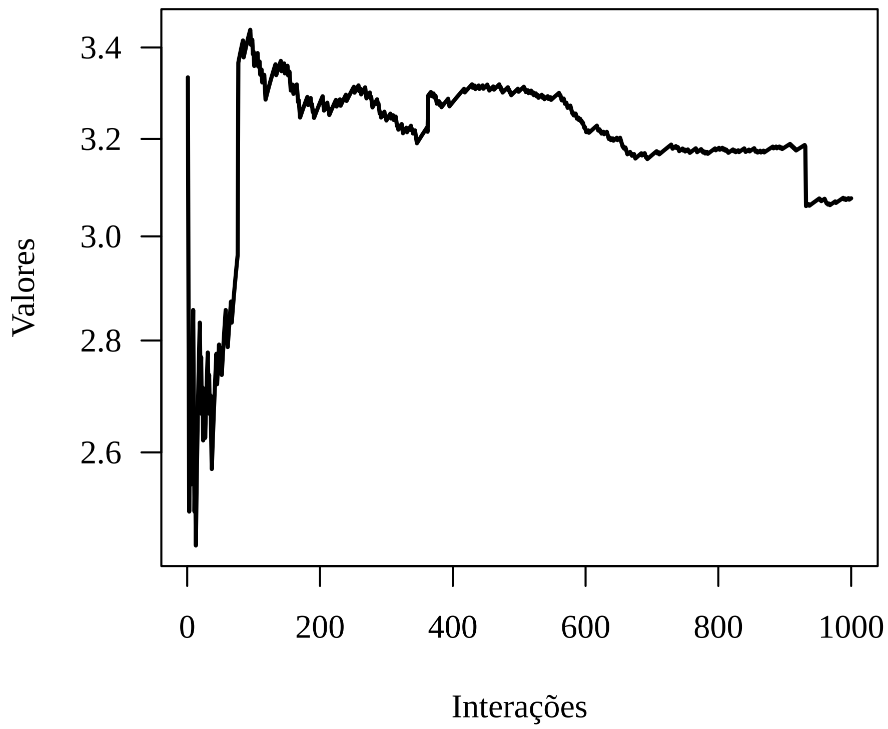
$$A_c = \frac{\pi}{4} \approx 4 \cdot \frac{\text{dentro}}{\text{total}} = \pi$$

```
long double monteCarlo(unsigned long long n) {
    unsigned long long ins;
    ins = 0;
    long double x, y, z, pi;

    srand(time(NULL));
    unsigned long long i;
    for (i = 0; i < n; i++) {
```

[1]O ENIAC produziu 2,037 dígitos de π em 70 horas em 1949 usando a *Fórmula de Machin*[2, p. 592 e 627]

```
        x = (long double) rand() / (long double) RAND_MAX;
        y = (long double) rand() / (long double) RAND_MAX;

        z = x * x + y * y;
        if (z <= 1) {
            ins++;
        }
    }

    pi = (long double) 4 * ins;
    pi = (long double) pi / (long double) n;

    return pi;
}
```

Figura 1: Convergência para o método de *Monte Carlo*

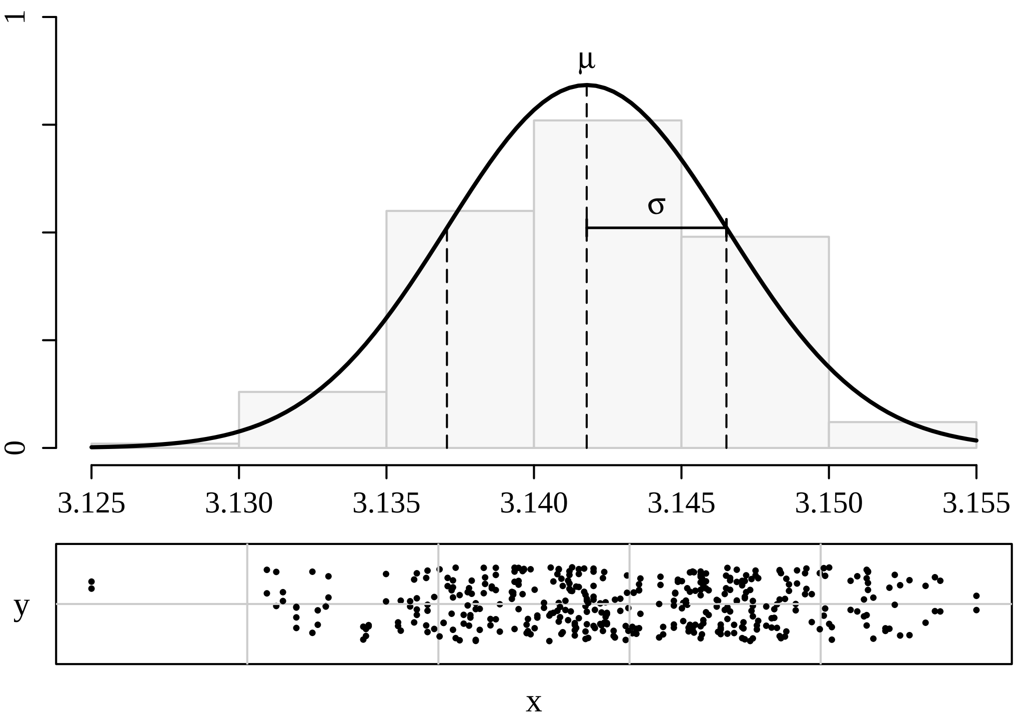


Fonte: Elaborado pelo próprio autor

Observando que os valores coletados na Figura 1 convergem com flutuações nos valores, pois os valores são obtidos de forma aleatória. Então, podemos interpretar estatisticamente este resultado realizando uma hipótese de normalidade na amostra.

Temos, um histograma:

Figura 2: Histograma dos valores obtidos no método de *Monte Carlo*, com curva de densidade, aplicado à coleta de π



Fonte: Elaborado pelo próprio autor

Devido às propriedades do gráfico, temos:

- O ponto $x = \mu$ é um ponto de máximo absoluto, sendo o único ponto crítico do gráfico, sendo representado pela linha pontilhada central da Figura 2
- Possui dois pontos de inflexão $x = \mu - \sigma$ e $x = \mu - \sigma$, sendo representado pelas linhas pontilhadas nas extremidades do gráfico.
- A área entre $\mu \pm \sigma$ representa 68% da área total do gráfico, ou seja, 68% dos valores de π estão presentes nesse intervalo.

Sendo: $mean(var)$, média aritmética dos valores de π obtidos.

$$var = \{x_1,\ x_2,\ \dots,\ x_n\}$$

$$\mu = v\bar{a}r = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

$sd(var)$, desvio padrão dos valores obtidos, ou seja, medida de dispersão com as mesmas unidades que a amostra.

$$\sigma = sd = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - v\bar{a}r)^2}$$

$sd(var)/sqrt(m)$, desvio padrão da média ($v\bar{a}r$), no limite para m grande. O desvio padrão da média (ξ) caracteriza as flutuações dos valores $v\bar{a}r_i$ em torna da média população μ.

$$\xi = \frac{\sigma}{\sqrt{m}}$$

E realizando testes estatísticos nos dados para determinar se os dados afastam ou não da distribuição normal.

```
shapiro.test(var)
```

```
[1] Shapiro-Wilk normality test

data:  var
W = 0.9987, p-value = 0.6886
```

Observando que o $valor - p$ retornado pelo teste é maior que 0,05, fortalecendo a hipótese que os dados são normalmente distribuídos.

Além disso, tem-se:

```
library(nortest)
ad.test(var)
```

```
[1]   Anderson-Darling normality test

data:  var
A = 0.2726, p-value = 0.6684
```

```
ks.test(var, "pnorm", mean(var), sd(var))
```

```
[1] Asymptotic one-sample Kolmogorov-Smirnov test

data:  var
D = 0.015002, p-value = 0.978
alternative hypothesis: two-sided
```

O $valor - p$ em todos os testes anteriores foram superiores que 0,05, fortalecendo a nossa hipótese inicial de que a normalidade não é rejeitada, como demonstrado na Figura 2. Então, apresentando erros de medição aleatórios.

Desenvolvendo um Caso Geral

Para facilitar o desenvolvimento dos algoritmos para calcular arctg vamos realizar um caso geral, onde iremos adotar o valor para x sendo igual a $1/a$, pois em todas as fórmulas utilizadas é usado arctg do tipo arctg $1/a$. Então, podemos desenvolver a série da função arctg da seguinte maneira:

$$\begin{aligned}
A \cdot \operatorname{arctg} x =& A \cdot \sum_{k=0}^{n} \left((-1)^k \, \frac{x^{2k+1}}{2k+1} \right) + A \cdot R_n(x) \Rightarrow \\
\Rightarrow A \cdot \operatorname{arctg}\left(\frac{1}{a}\right) =& A \cdot \sum_{k=0}^{n} \left((-1)^k \, \frac{\left(\frac{1}{a}\right)^{2k+1}}{2k+1} \right) + A \cdot R_n\left(\frac{1}{a}\right) \\
=& A \cdot \underbrace{\sum_{k=0}^{n} \left((-1)^k \, \frac{1}{a^{2k+1}} \cdot \frac{1}{2k+1} \right)}_{P_n\left(\frac{1}{a}\right)} + A \cdot R_n\left(\frac{1}{a}\right)
\end{aligned}$$

$$\begin{aligned}
P_n\left(\frac{1}{a}\right) =& \frac{1}{a} - \frac{1}{3a^3} + \frac{1}{5a^5} + \frac{1}{7a^7} - \frac{1}{9a^9} + \frac{1}{11a^{11}} + \cdots + \\
& + \frac{1}{(2n+1) \cdot a^{2n+1}} - \frac{1}{(2n+3) \cdot a^{2n+3}}
\end{aligned}$$

$$\begin{aligned}
& \frac{1}{(2n+1) \cdot a^{2n+1}} - \frac{1}{(2n+3) \cdot a^{2n+3}} = \\
=& \frac{(2n+3) \cdot a^{2n+3} - (2n+1) \cdot a^{2n+1}}{(2n+1) \cdot a^{2n+1} \cdot (2n+3) \cdot a^{2n+3}} = \\
=& \frac{(2n+3) \cdot a^2 - (2n+1)}{(2n+1) \cdot (2n+3) \cdot a^{2n+3}}, \text{ usando que } n = 2k \text{, temos,} \\
=& \frac{(4k+3) \cdot a^2 - (4k+1)}{(4k+1) \cdot (4k+3) \cdot a^{4k+3}}
\end{aligned}$$

$$\begin{aligned}
P_n\left(\frac{1}{a}\right) =& \sum_{k=0}^{n} \left(\frac{(4k+3) \cdot a^2 - (4k+1)}{(4k+1) \cdot (4k+3) \cdot a^{4k+3}} \right) \\
=& \sum_{k=0}^{n} \left(\frac{(i+3) \cdot a^2 - (i+1)}{(i+1) \cdot (i+3) \cdot a^{i+3}} \right), \; i{+}{=}4
\end{aligned}$$

generalArctg()

```
long double generalArctg(unsigned long int n, unsigned long int a ) {
```

```c
    unsigned long int pow, aux_pow, numerator, quotient;
    long double res;

    pow = a * a * a;
    aux_pow = a * a * a * a;
    res = 0;

    int i;
    for (i = 0; i < n; i+=4) {
        numerator = (unsigned long int) (i + 3) * (a * a) - (i +
           1);
        quotient = (unsigned long int) (i + 1) * (i + 3) * pow;
        res += (long double) numerator / quotient;

        pow *= aux_pow;
    }

    return res;
}

```

$$\left|R_n\left(\frac{1}{a}\right)\right| \leq \frac{\left(\frac{1}{a}\right)^{2n+3}}{2n+3}$$

$$A \cdot \frac{1}{a^{2n+3}} \cdot \frac{1}{2n+3} < A \cdot \frac{1}{a^{2n+3}} \cdot \frac{1}{2 \cdot (n+1)} \overset{?}{<} \epsilon$$
$$\frac{1}{\epsilon} \cdot \frac{A}{2a} < a^{2 \cdot (n+1)} \cdot (n+1)$$

Como $\epsilon = 10^{-d}$, tem-se: $10^d \cdot \dfrac{A}{2a} < a^{2 \cdot (n+1)} \cdot (n+1)$, sendo n então a quantidade de iterações necessárias.

generalErrorHandle()

```c
unsigned long int generalErrorHandle(unsigned long int d,
    unsigned long int A, unsigned long int a) {
    unsigned long int n, err, small, pow, aux_pow;

    n = 0;
    err = 1;

    aux_pow = a * a;
    pow = aux_pow;
```

```

    int i;
    for (i = 0; i < d; i++) {
        err *= 10;
    }

    small = err * A / (2 * a);

    while (small >= (unsigned long int) pow * (n + 1)) {
        pow *= aux_pow;
        n++;
    }

    return n;
}

```

Porém utilizando essas funções para $\pi/4 = \text{arctg}\,(1)$ o maior erro que conseguimos alcançar, devido às limitações dos tipos utilizados (`unsigned long int` e `long double`) nas funções, é $\epsilon = 10^{-9}$. Então é necessário utilizar funções para cálculos de precisão arbitrária, sendo, nesse caso, utilizado a biblioteca *GNU MPFR*, uma biblioteca *C* para cálculos de ponto flutuante de precisão múltipla com arredondamento correto, é baseado na biblioteca *GNU Multiple Precision Arithmetic* (*GMP*).

Usando *MPFR* e *GMP*

`eralArctg_mpfr()` Substituindo os tipos tradicionais pelos utilizados nas bibliotecas, temos:

```
void generalArctg_mpfr(const mpz_t n, unsigned long int a, fp_t
   res) {
    fp_set_ui(res, 0);

    mpz_t i, i_1, i_3, aux_pow_4, aux_pow_2, pow, a_gmp,
        numerator, quotient;
    mpz_inits(i, i_1, i_3, aux_pow_4, aux_pow_2, pow, a_gmp,
        numerator, quotient, NULL);

    mpz_set_ui(a_gmp, a);
    mpz_pow_ui(aux_pow_4, a_gmp, 4); // aux = a^4
    mpz_pow_ui(aux_pow_2, a_gmp, 2); // aux = a^4
    mpz_pow_ui(pow, a_gmp, 3); // pow = a^3

    fp_t frac, numerator_mpfr, quotient_mpfr;
```

```
    fp_inits(frac, numerator_mpfr, quotient_mpfr, NULL);

    for (mpz_set_ui(i, 0); mpz_cmp(i, n) < 0; mpz_add_ui(i, i, 4)) { // for (int i = 0; i < n; i+=4) {...}
        mpz_add_ui(i_1, i, 1); // i + 1
        mpz_add_ui(i_3, i, 3); // i + 3

        // numerator
        mpz_mul(numerator, i_3, aux_pow_2); // (i + 3) * a^2
        mpz_sub(numerator, numerator, i_1); // (i + 3) * a^2 - (i + 1)

        // quocient
        mpz_mul(quotient, i_1, i_3); // (i + 1) * (i + 3)
        mpz_mul(quotient, quotient, pow); // (i + 1) * (i + 3) * a^{i+3}

        fp_set_z(numerator_mpfr, numerator);
        fp_set_z(quotient_mpfr, quotient);

        fp_div(frac, numerator_mpfr, quotient_mpfr); // frac = numerator / quotient
        fp_add(res, res, frac); // res += frac

        mpz_mul(pow, pow, aux_pow_4); // pow *= a^4
    }
    mpz_clears(i, i_1, i_3, aux_pow_4, aux_pow_2, pow, a_gmp, NULL);

    fp_clears(frac, numerator_mpfr, quotient_mpfr, NULL);
}

```

eralErrorHandle_gmp()

```
void generalErrorHandle_gmp(unsigned long int d, unsigned long int A, unsigned long int a, mpz_t n) {
    mpz_set_ui(n, 0);

    mpz_t err, aux_pow, pow, aux_cmp;
    mpz_inits(err, aux_pow, pow, aux_cmp, NULL);
    mpz_set_ui(err, 1);

    for (int i = 0; i < d; i++) {
        mpz_mul_ui(err, err, 10);
    }
```

```c

    mpz_mul_ui(err, err, A);
    mpz_div_ui(err, err, 2 * a);

    mpz_set_ui(aux_pow, a);
    mpz_set(pow, aux_pow);

    mpz_set(aux_cmp, pow);
    mpz_mul(aux_cmp, aux_cmp, n);
    while (mpz_cmp(err, aux_cmp) > 0) {  // err >= aux_cmp
        mpz_mul(pow, pow, aux_pow);
        mpz_set(aux_cmp, pow);
        mpz_add_ui(n, n, 1);
        mpz_mul(aux_cmp, aux_cmp, n);
    }

    mpz_clears(err, aux_pow, pow, aux_cmp, NULL);
}

```

Fórmula de *Machin* Original

`machin()` Função `machin` resolvendo a *Fórmula de Machin Original*:

$$\frac{\pi}{4} = 4 \cdot \operatorname{arctg} \frac{1}{5} - \operatorname{arctg} \frac{1}{239}$$

```c
void machin(unsigned long long err, fp_t pi) {
    fp_t arctg_5;
    generalArctgWithError_mpfr(arctg_5, err, 4, 5);

    fp_t arctg_239;
    generalArctgWithError_mpfr(arctg_239, err, 1, 239);

    fp_sub(pi, arctg_5, arctg_239);
    fp_mul_ui(pi, pi, 4);

    fp_clears(arctg_5, arctg_239, NULL);
}

```

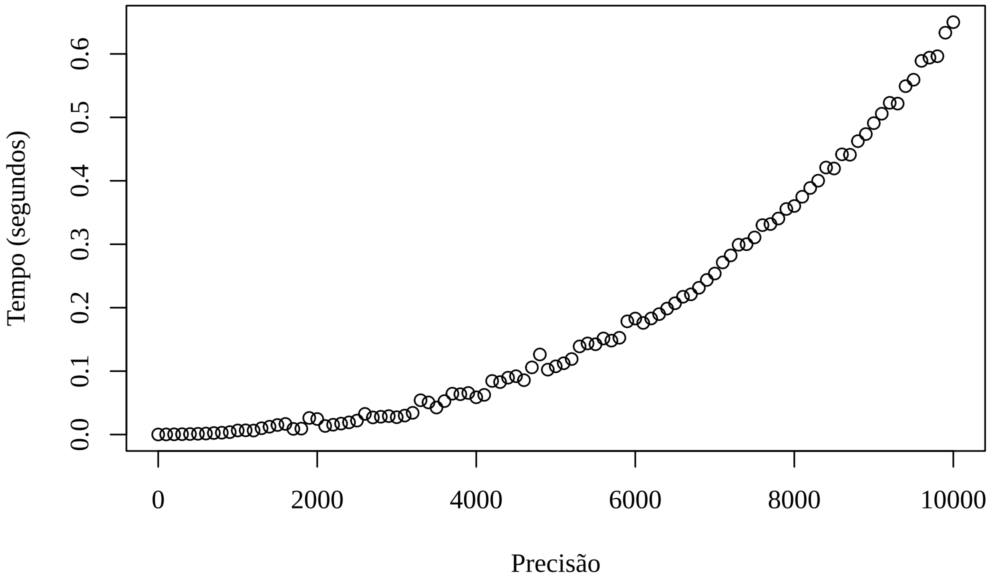
FÓRMULA DE *KIKUO TAKANO*

`takano()` Função `takano` resolvendo a *Fórmula de Kikuo Takano*:

$$\frac{\pi}{4} = 12 \cdot \text{arctg}\,\frac{1}{49} + 32 \cdot \frac{1}{57} - 5 \cdot \frac{1}{239} + 12 \cdot \text{arctg}\,\frac{1}{110.443} \qquad \text{(K. Takano, 1982)}$$

```
void takano(unsigned long long err, fp_t pi) {
    fp_t arctg_49;
    generalArctgWithError_mpfr(arctg_49, err, 12, 49);

    fp_t arctg_57;
    generalArctgWithError_mpfr(arctg_57, err, 32, 57);

    fp_t arctg_239;
    generalArctgWithError_mpfr(arctg_239, err, 5, 239);

    fp_t arctg_110443;
    generalArctgWithError_mpfr(arctg_110443, err, 12, 110443);

    fp_add(pi, arctg_49, arctg_57);
    fp_sub(pi, pi, arctg_239);
    fp_add(pi, pi, arctg_110443);
    fp_mul_ui(pi, pi, 4);

    fp_clears(arctg_49, arctg_57, arctg_239, arctg_110443, NULL);
}

```

Figura 3: Tempo de evolução com o aumento da precisão do valor de π usando fórmula de *Kikuo Takano*



Fonte: Elaborado pelo próprio autor

FÓRMULA DE *FREDRIK CARL MÜLERTZ STØRMER*

`stormer()` Função `stormer` resolvendo a *Fórmula de Fredrik Carl Mülertz Størmer*:

$$\frac{\pi}{4} = 44 \cdot \operatorname{arctg}\frac{1}{57} + 7 \cdot \frac{1}{239} - 12 \cdot \frac{1}{682} + 24 \cdot \operatorname{arctg}\frac{1}{12.943} \qquad \text{(F. C. M. Størmer, 1896)}$$

```
void stormer(unsigned long long err, fp_t pi) {
    fp_t arctg_57;
    generalArctgWithError_mpfr(arctg_57, err, 44, 57);

    fp_t arctg_239;
    generalArctgWithError_mpfr(arctg_239, err, 7, 239);

    fp_t arctg_682;
    generalArctgWithError_mpfr(arctg_682, err, 12, 682);

    fp_t arctg_12943;
    generalArctgWithError_mpfr(arctg_12943, err, 24, 12943);

    fp_add(pi, arctg_57, arctg_239);
    fp_sub(pi, pi, arctg_682);
    fp_add(pi, pi, arctg_12943);
```

```
    fp_mul_ui(pi, pi, 4);

    fp_clears(arctg_57, arctg_239, arctg_682, arctg_12943, NULL);
}
```

Referências

[1] ADVANCED MICRO DEVICES. **AOCC User Guide**. 5.0 ed. 57222, [*S. l.*]: [*S. n.*], 2024. Disponível em: `https://docs.amd.com/r/en-US/57222-AOCC-user-guide`. Acesso em: 14 de mar. de 2025.

[2] BERGGREN, L.; BORWEIN, J.; BORWEIN, P. **Pi**: a source book. Nova Iorque: Springer Science, 1997. ISBN 978-1-4757-2738-8.

[3] ULAM, S. **Special Issue**. 15, [*S. l.*]: Los Alamos Science, 1987.

APÊNDICES

A Valor de Pi

Sendo, π_n, n a quantidade de casas decimais, temos, usando fórmula de *Kikuo Takano*:

π_{100} = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510 58209 74944 59230 78164 06286 20899 86280 34825 34211 70679