Metody Obliczeniowe w Nauce i Technice

Laboratorium 7 – raport

Mateusz Kocot

1. Kwadratury

Najpierw zaimplementowano klasę abstrakcyjną *Integral*, z której będą dziedziczyć klasy odpowiedzialne za poszczególne metody numeryczne obliczania całek.

```
class Integral
{
protected:
    constexpr static double EPS = 1e-8;
public:
    virtual double calculate(double start, double end, int n,
                             std::function<double(double)> f) = 0;
};
```

Stałą *EPS* wprowadziłem ze względu na niedokładność typu zmiennopozycyjnego *double*. Metodą, która będzie w klasach dziedziczących odpowiedzialna za całkowanie, jest metoda wirtualna *calculate*. Przyjmuje ona końce przedziału, na którym będzie liczona całka: *start*, *end*, liczbę przedziałów (a w przypadku Monte Carlo – liczbę losowanych próbek): *n* oraz funkcję *f*. Funkcja ta zwraca wartość obliczonej całki oznaczonej.

Poniżej przedstawiono klasy dziedziczące z *Integral*:

a) *IntegralRectangular* – klasa odpowiedzialna za metodę prostokątów,
b) *IntegralTrapezoidal* – klasa odpowiedzialna za metodą trapezów,
c) *IntegralSimpson* – klasa odpowiedzialna za metodę Simpsona,
d) *IntegralMonteCarlo* – klasa odpowiedzialna za metody Monte Carlo.

```
class IntegralRectangular : public Integral
{
public:
    double calculate(double start, double end, int n,
                     std::function<double(double)> f);
};

class IntegralTrapezoidal : public Integral
{
public:
    double calculate(double start, double end, int n,
                     std::function<double(double)> f);
};
```

```
class IntegralSimpson : public Integral
{
public:
    double calculate(double start, double end, int n,
                     std::function<double(double)> f);
};

class IntegralMonteCarlo : public Integral
{
public:
    double calculate(double start, double end, int n,
                     std::function<double(double)> f);
    double findPi(int n);
};
```

W tym punkcie zajmę się tylko pierwszymi trzema metodami – kwadraturami.

Najpierw zaimplementowano najprostszą metodę – metodę prostokątów.

```
double IntegralRectangular::calculate(double start, double end, int n,
                                      std::function<double(double)> f)
{
    double width = (end - start) / n;
    double result = 0;
    for (double x = start + width / 2; x < end - EPS; x += width)
    {
        result += f(x);
    }
    result *= width;
    return result;
}
```

Funkcja ta na wzór definicji całki Riemanna dzieli wejściowy przedział na *n* mniejszych przedziałów. Na tych mniejszych przedziałach funkcja *f* przybliżana jest za pomocą prostokątów. W celu zwiększenia dokładności, do obliczenia pól prostokątów wykorzystywane są punkty lezące na środkach przedziałów. W związku z tym, ostatecznie nie są obliczane wartości funkcji w punktach *start* i *end*.

Następnie zaimplementowano metodę trapezów.

```
double IntegralTrapezoidal::calculate(double start, double end, int n,
                                      std::function<double(double)> f)
{
    double width = (end - start) / n;
    double result = (f(start) + f(end)) / 2.0;
    double x = start + width;
    while (x < end - EPS)
    {
        result += f(x);
        x += width;
    }
    result *= width;
    return result;
}
```

Podobnie jak poprzednio, wejściowy przedział dzielony jest na *n* mniejszych przedziałów. Tym razem jednak, funkcja *f* na tych przedziałach przybliżana jest za pomocą trapezów. Do obliczania pól trapezów wykorzystywane są punkty skrajne przedziałów. W związku z tym, metoda ta liczy wartości funkcji *f* w punktach *start* i *end*.

Niewątpliwie najdokładniejszą z trzech metod jest metoda Simpsona. Jej implementacja znajduje się poniżej.

```
double IntegralSimpson::calculate(double start, double end, int n,
                                  std::function<double(double)> f)
{
    double width = (end - start) / n;
    double result = f(start) + f(end);
    double t = start + width / 2;
    double y = start + width;
    while(y < end - EPS)
    {
        result += 4 * f(t) + 2 * f(y);
        t += width;
        y += width;
    }
    result += 4 * f(t);
    result /= 6;
    result *= width;
    return result;
}
```

W tym przypadku znowu wejściowy przedział dzielony jest na *n* mniejszych przedziałów. Na tych przedziałach funkcja *f* przybliżana jest za pomocą paraboli według następującego wzoru.

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{x_{i+1} - x_i}{6}\left(f(x_i) + 4f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_{i+1})\right),$$

Uwzględniane są zatem zarówno punkty na krańcach przedziałów, jak i punkty leżące na środkach przedziałów. W związku z tym, funkcja ta liczy wartości funkcji *f* w punktach *start* i *end*.

W ramach testów, wykorzystam wyżej zaimplementowane metody do obliczenia czterech całek, które przedstawiono poniżej. Ich dokładne wartości policzono za pomocą WolframAlpha (https://www.wolframalpha.com/). Wartości te będę rozpatrywał z dokładnością 6 miejsc po przecinku.

$$\int_1^3 f_1(x)dx = \int_1^2 e^{-x^2}dx \cong 0.139383$$

$$\int_1^3 f_2(x)dx = \int_1^3 \sqrt{1+x^3}dx \cong 6.229969$$

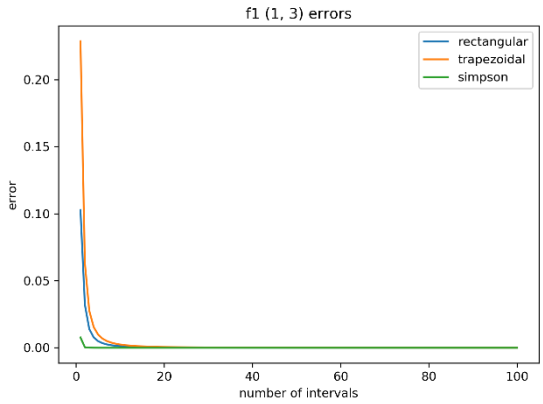
$$\int_1^3 f_3(x)dx = \int_1^3 \frac{\sin(2x)}{x}dx \cong -0.253232$$

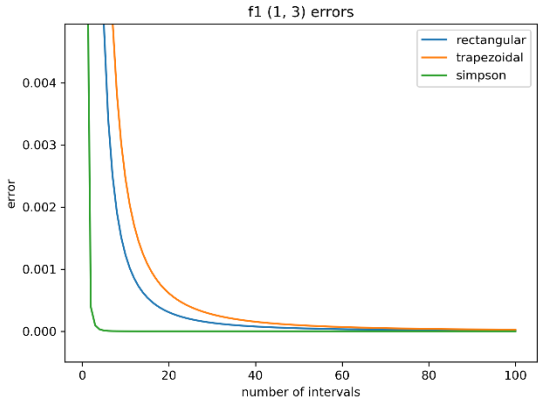
$$\int_1^3 f_4(x)dx = \int_1^3 \sin(x^2)\,dx \cong 0.463294$$

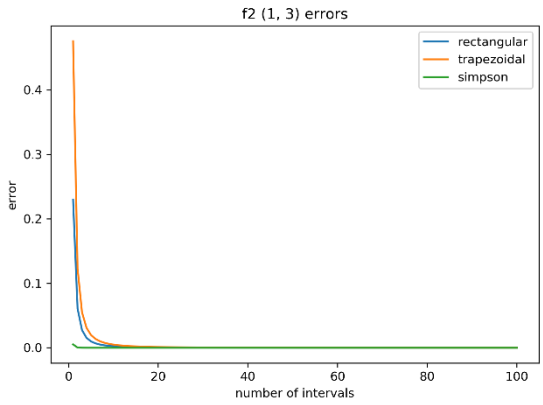
Funkcje te zaimplementowano w postaci wektora wyrażeń lambda:

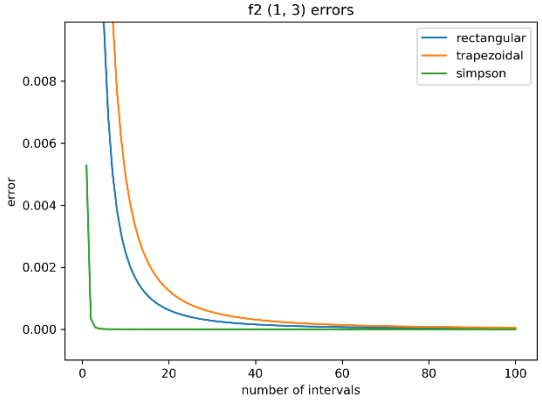
```
std::vector< std::function<double(double)> > funs =
    {[](double x) { return exp(-x*x); },            // f1(x) = e^(-x^2)
     [](double x) { return x*x*x; },                // f2(x) = x^3
     [](double x) { return 1 / x; },                // f3(x) = 1 / x
     [](double x) { return sin(4 * x) + x*x; }};    // f4(x) = sin(4x) + x^2
```

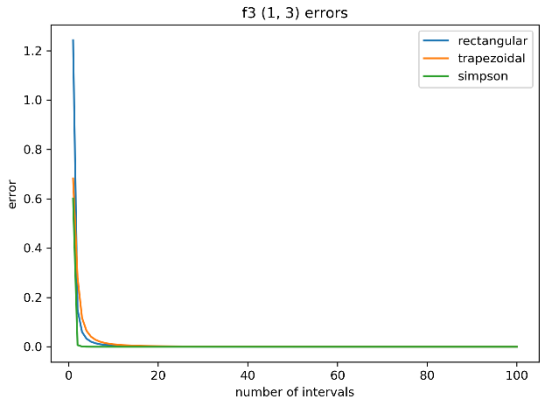
Poniżej przedstawię wykresy błędów w zależności od liczby przedziałów dla obliczanych metodami numerycznymi całek testowych. Wykresy po prawej stronie mają zmniejszony zakres współrzędnej *y*.

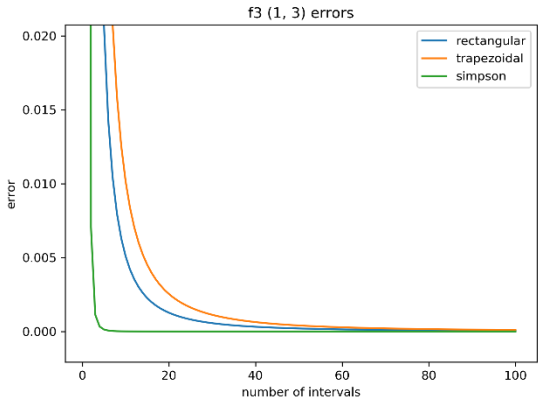


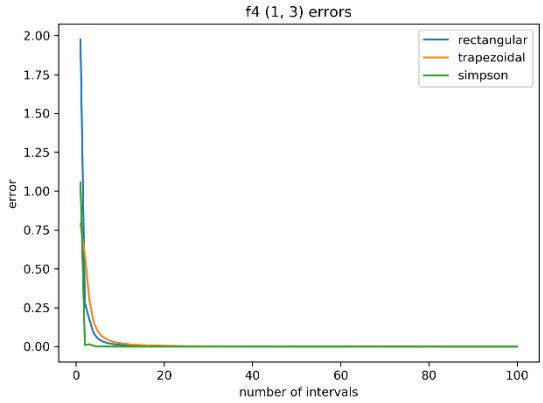


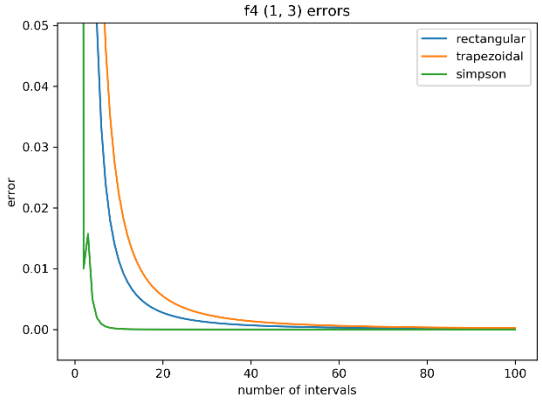












Wszystkie wykresy (po odpowiedniej stronie) są do siebie bardzo podobne. Niemałym zaskoczeniem może być fakt, że praktycznie dla każdej funkcji i każdej liczby przedziałów, metoda prostokątów jest dokładniejsza od metody trapezów. Dla f_3 i f_4 jedynie dla małej liczby przedziałów, metoda trapezowa okazuje się lepsza od metody prostokątów. Istnieją jednakże wzory na błędy obu metod. Dla wielomianów stopnia co najwyżej pierwszego, są one dokładne. W pozostałych przypadkach wzory na błędy dla przedziału $[a, b]$ wyglądają następująco:

- Metoda prostokątów (dla pewnego $\xi \in [a, b]$):

$$\frac{1}{24}(b-a)^3 f''(\xi)$$

- Metoda trapezów (dla pewnego $\xi \in [a, b]$):

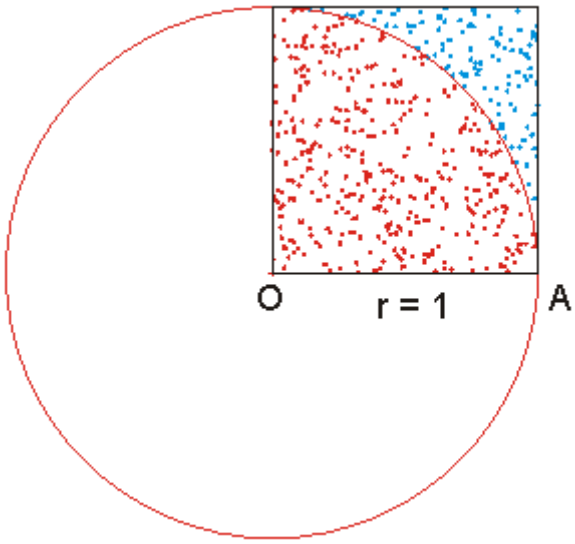
$$\frac{1}{12}(b-a)^3 f''(\xi)$$

Widać więc, że błąd metody trapezów jest większy (dwukrotnie) od błędu metody prostokątów.

Metoda Simpsona dla każdej funkcji i każdej liczby przedziałów liczy całkę z mniejszym błędem niż metody prostokątów i trapezów. Szacowanie funkcji parabolą jest w końcu dużo dokładniejsze od szacowania wielokątami. Oprócz tego, metoda ta będzie zawsze dokładna dla wielomianu stopnia co najwyżej trzeciego, mimo że wykorzystywana jest parabola, tj. wielomian stopnia drugiego. Sytuacja tutaj wygląda jednak podobnie jak w przypadku np. metody prostokątów, w której funkcję przybliżamy wielomianami stopnia zerowego, a jednak jest dokładna dla wielomianów stopnia pierwszego.

2. Metody Monte Carlo

W przeciwieństwie do wyżej opisanych metod, metody Monte Carlo bazują na losowaniu próbek. Prostym przykładem wykorzystania Monte Carlo jest przybliżenie wartości π (rys. 1).



Rys. 1. Rysunek obrazujący wykorzystanie metod Monte Carlo do przybliżenia wartości liczby π. Źródło: *http://www.moebius-informatik.ch/estimating-pi-with-the-monte-carlo-approach/*

Weźmy ćwiartkę koła jednostkowego o polu $\frac{\pi}{4}$. Następnie będziemy losować *n* punktów w kwadracie $[0,1] \times [0,1]$. Stosunek liczby punktów wylosowanych w środku do liczby wszystkich punktów pozwala przybliżyć pole ćwiartki. Wobec tego:

$$\frac{n_{środek}}{n} \approx \frac{\pi}{4} \Rightarrow \pi \approx 4\frac{n_{środek}}{n}$$

Zaimplementowano funkcję przybliżającą π metodą Monte Carlo jako metodę klasy *IntegralMonteCarlo*.

```
double IntegralMonteCarlo::findPi(int n)
{
    std::default_random_engine gen;
    std::uniform_real_distribution<double> distr(0, 1);
    int n_in = 0;
    for (int i = 0; i < n; i++)
    {
        double x = distr(gen);
        double y = distr(gen);
        double dist = sqrt(x*x + y*y);
        if (dist <= 1)
        {
            n_in++;
        }
    }
    return 4.0 * n_in / n;
}
```

Wykorzystane zostały funkcje *std::default_random_engine* oraz *std::unfiorm_real_distribution* z biblioteki *random*.

Otrzymane przybliżenia π dla różnej liczby próbek przedstawiono w tab. 1.

Liczba próbek	Otrzymane przybliżenie π
10	2.8
100	3.2
1000	3.192
10000	3.14032
100000	3.1396
1000000	3.14211

Tab. 1. Przybliżenia π uzyskane metodą Monte Carlo dla różnych liczb próbek

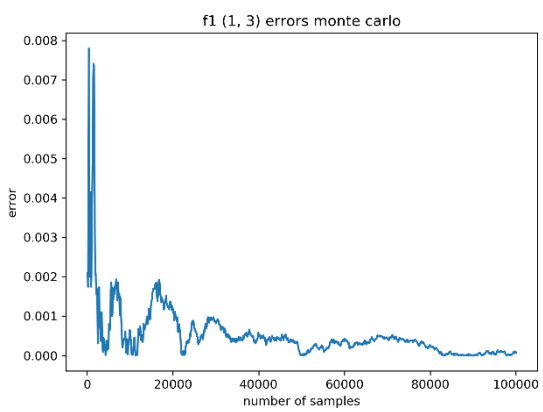
Dla porównania, prawdziwa wartość $\pi \cong 3.141593$.

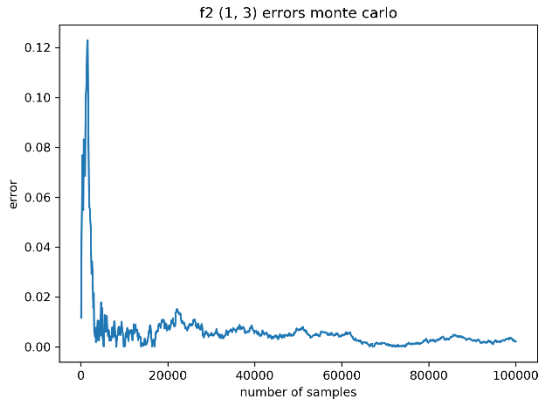
Dopiero 1000000 próbek pozwoliło uzyskać dokładność na poziomie 2 miejsc po przecinku. Jednakże, przybliżenie dla 100 próbek jest już w miarę dokładnie. Metoda ta będzie więc dobra do szacowania określonych wartości wtedy, gdy nie będzie wymagana duża dokładność.

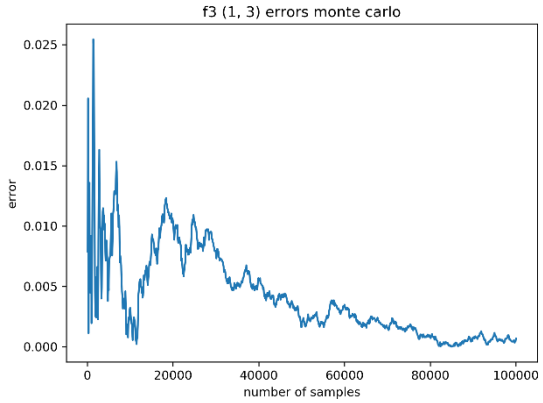
Metody Monte Carlo można wykorzystać także do przybliżenia całki oznaczonej. Tym razem, liczymy wartość funkcji w *n* losowych punktach. Następnie sumę tych wartości dzielimy przez *n* oraz mnożymy przez szerokość przedziału otrzymując oczekiwane przybliżenie. W klasie *IntegralMonteCarlo* zaimplementowano metodę calculate.

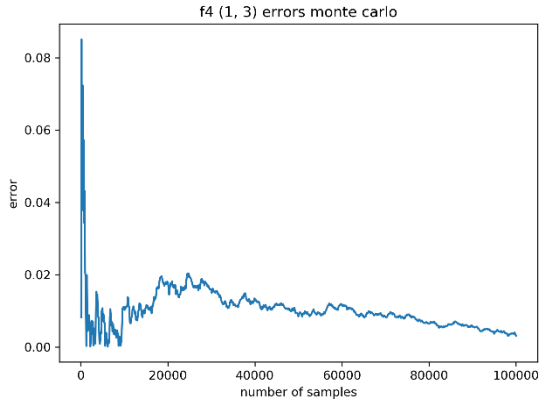
```
double IntegralMonteCarlo::calculate(double start, double end, int n,
                                     std::function<double(double)> f)
{
    double result = 0;
    std::default_random_engine gen;
    std::uniform_real_distribution<double> distr(start, end);
    for (int i = 0; i < n; i++)
    {
        result += f(distr(gen));
    }
    result *= (end - start);
    result /= n;
    return result;
}
```

Wykresy błędów od liczby próbek (od 100 do 100000, co 100) dla wcześniej wprowadzonych całek testowych pokazano poniżej.







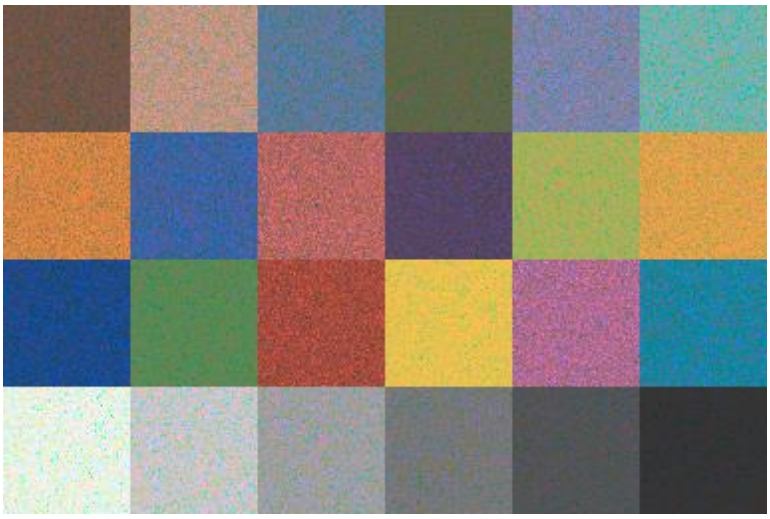


Na wykresach widać, że generalnie jest tendencja spadkowa. Metoda ta polega jednak na losowości, więc raz otrzymane dobre przybliżenie może dla być gorsze nawet dla większej liczby próbek, co widać najlepiej na wykresie dla f_3. Mimo wszystko, błędy te nawet dla małych liczb próbek są nieznaczne w porównaniu do prawdziwych wartości przedstawionych w punkcie 1. Wobec tego, tak jak w przypadku przybliżania π, metody Monte Carlo warto stosować do przybliżenia wartości całek, gdy nie zależy nam na tym, żeby wynik był bardzo dokładny. W ogólności jednak, dla całek testowych, które wybrałem, wydajniejsze są trzy wcześniej przedstawione kwadratury. Dla 100 przedziałów, całki przybliżone zostały tam bardzo dokładnie. Tutaj natomiast, dla 100, 1000, a nawet 10000 próbek, błąd potrafi być zauważalny.

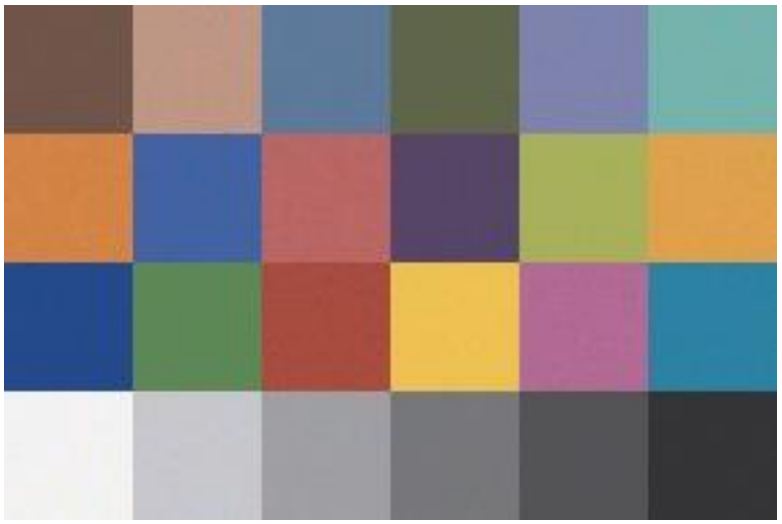
3. Zadanie praktyczne

Pobrano plik *mcintegration.cpp* (https://www.scratchapixel.com/code.php?id=31&origin=/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice)

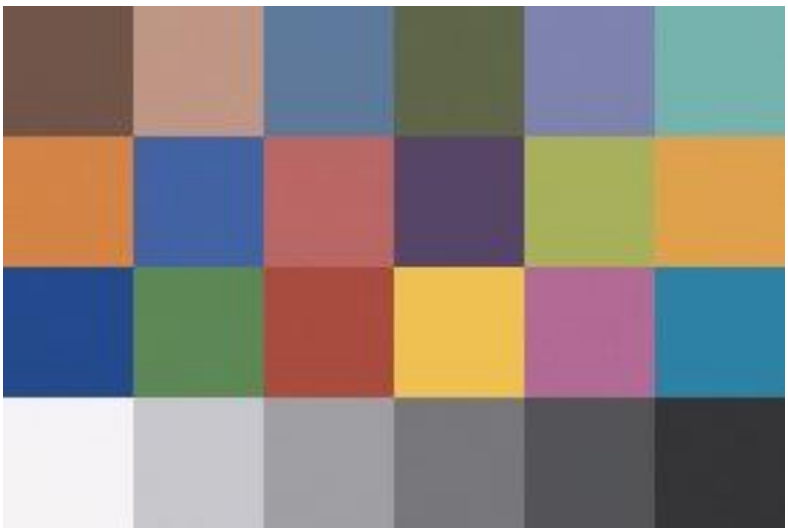
Dokonano w nim kilka zmian. Po pierwsze, można teraz wprowadzić maksymalną liczbę przebiegów, po których program się wyłączy. Oprócz tego, rezultat zapisywany jest po 1, 10, 50, 100 i 200 przebiegach, co odpowiada odpowiednio 32, 320, 1600, 3200 oraz 6400 losowanym próbkom. Wyniki prób odtworzenia tablicy McBetha (ang. *McBeth chart*) przedstawiono poniżej.



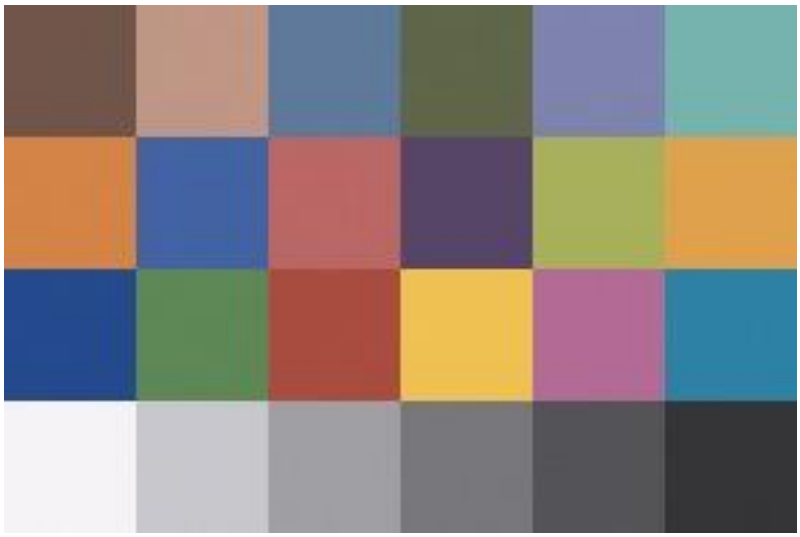
Wynik renderowania po 1 przebiegu (32 próbek)



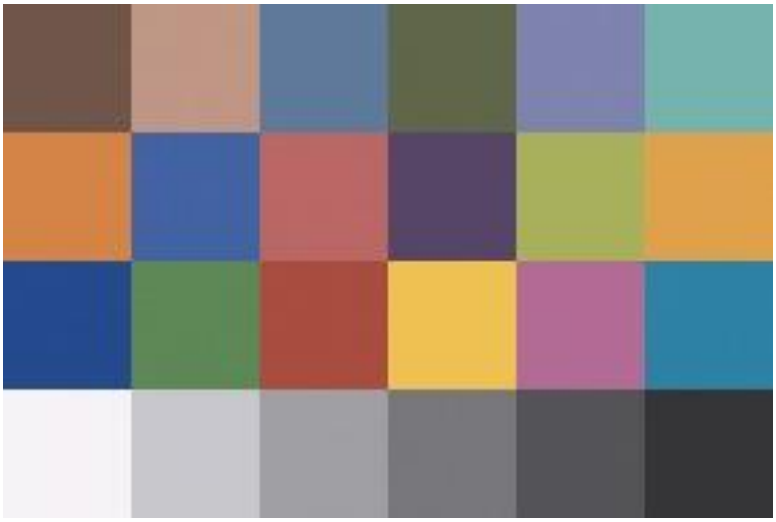
Wynik renderowania po 10 przebiegach (320 próbek)



Wynik renderowania po 50 przebiegach (1600 próbek)



Wynik renderowania po 100 przebiegach (3200 próbek)



Wynik renderowania po 200 przebiegach (6400 próbek)

Wynik po 1 przebiegu jest mocno zaszumiony. Wynik po 10 przebiegach, w zależności od potrzeb, mógłby już być akceptowany. Ciężko natomiast znaleźć już szum na tablicy po 50 przebiegach. Mimo że wynik po 200 przebiegach na pewno jest lepszy, to różnica nie wydaje się duża. Oczywiście to, czy warto renderować, dalej zależy od potrzeb. Więc jeśli wynik po danej ilości przebiegów nie będzie wystarczająco dobry, renderowanie można kontynuować. Jest to niewątpliwa zaleta renderowania progresywnego (ang. *progressive rendering*). Kolejną zaletą jest możliwość ustawienia limitu czasu i wyrenderowanie tak dobrego wyniku, na jaki nas czasowo stać.