Московский Авиационный Институт
(национальный исследовательский университет)

Курсовая работа по курсу
«Численные методы»

Метод Монте-Карло для вычисления кратных интегралов

Студент:
Макаров Никита, 8о-306Б

Руководитель:
Ревизников Д.Л.

Москва
2015

Содержание

1 Введение

Методами Монте-Карло называют численные методы решения математических задач при помощи моделирования случайных величин. Однако, решать методами Монте-Карло можно любые математические задачи, а не только задачи вероятностного происхождения, связанные со случайными величинами.

Важнейшим приемом построения методов Монте-Карло является сведение задачи к расчету математических ожиданий. Так как математические ожидания чаще всего представляют собой обычные интегралы, то центральное положение в теории метода Монте-Карло занимают методы вычисления интегралов.

Преимущества недетерминированных методов особенно ярко проявляются при решении задач большой размерности, когда применение традиционных детерминированных методов затруднено или совсем невозможно.

Границы между простым и сложным, возможным и невозможным существуют всегда, но с развитием вычислительной техники сдвигаются вдаль. До появления электронных вычислительных машин (ЭВМ) методы Монте-Карло не могли стать универсальными численными методами, ибо моделирование случайных величин вручную - весьма трудоемкий процесс. Развитию методов Монте-Карло способствовало бурное развитие ЭВМ. Алгоритмы Монте-Карло сравнительно легко программируются и позволяют производить расчеты во многих задачах, недоступных для классических численных методов. Так как совершенствование ЭВМ продолжается, есть все основания ожидать дальнейшего развития методов Монте-Карло и дальнейшего расширения области их применения.

2 Идея метода Монте-Карло

3 Погрешность метода Монте-Карло

4 Вычисление интегралов усреднением подынтегральной функции

5 Тестирование разработанного ПО

6 Исходный код программы

```
classdef MonteCarlo
    % Integration by Monte Carlo method

    methods(Static)

        %% randInRange: Returns random value in range [a,b].
        function [x] = randInRange(a,b)
            n = length(a);
            x = zeros(1,n);
            for i = 1:n
                x(i) = a(i) + (b(i) - a(i)) .* rand();
            end
        end

        %% checkPoint: Checks, is point x in G area, or not.
        function [inArea] = checkPoint(x,G)
            n = length(G); % old version: n = length(x) !!!
            inArea = 1;
            for i = 1:n
                inArea = inArea && G{i}(x);
            end
        end

        %% ndIntegral: Computing n-dimensional definite integral
         % at G area which no more than
         % n-dimensional parallelepiped with properties a and b.
        function [I,c] = ndIntegral(f,a,b,G,N,t_beta)
            %t_beta = 3; % beta = 0.997
            fSum = 0; % sum of computed values f(x)
            fSumSquared = 0; % squared sum of f(x)
            n = 0; % amount of points found in G

            % generating N random vectors x
            for i = 1:N
                x = MonteCarlo.randInRange(a,b);
                % check conditions, x must be in [a,b]
                inArea = MonteCarlo.checkPoint(x,G); % bool value
                % adding f(x)
                if (inArea)
                    fSum = fSum + f(x);
                    fSumSquared = fSumSquared + f(x)^2;
                    n = n + 1;
                end
            end

            c = inf;

            % computing n-dimensional volume of figure
            V = prod(b-a);

            % computing integral value
            I = V * fSum / N;
```

```

        if (nargin == 6)
            fAvg = fSum / n;
            fSquaredAvg = fSumSquared / n;

            Omega = n / N;

            % computing standard deviation
            S1 = sqrt(fSquaredAvg - fAvg^2);
            S2 = sqrt(Omega * (1 - Omega));

            % computing error
            error = V*t_beta*(Omega*S1/sqrt(n) + abs(fAvg)*S2/sqrt(N));
            c = V*t_beta*(sqrt(Omega)*S1 + fAvg*S2);

            disp('Error of integration:');
            disp(error);
        end
    end

    %% TESTS

    %% test1d: computing 1-dimensional definite integral
    function [I] = test1d(N0, maxError, t_beta)
        f = @(x) 1/sqrt(2*pi) * exp(-(x^2)/2);

        % conditions of G area
        x1Cond = @(x) (0<=x(1) && x(1)<=3);
        G = {x1Cond};

        % limitations for every element in x
        a(1) = 0; b(1) = 3;

        [I,c] = MonteCarlo.ndIntegral(f,a,b,G,N0,t_beta);
        disp('First approximation of integral value:');
        disp(I);

        % computing min necessary N
        N = ceil((c/maxError)^2);

        if (N > N0)
            disp('Minimal necessary N:');
            disp(N);

            % computing more correct integral value
            [I] = MonteCarlo.ndIntegral(f,a,b,G,N,t_beta);
            disp('Integral value:');
            disp(I);
        end
    end

    %% test2d: computing 2-dimensional definite integral
    function [I] = test2d(N0, maxError, t_beta)
        f = @(x) x(1) + x(2);

```

```
        % conditions of G area
        x1Cond = @(x) (0<=x(1) && x(1)<=2);
        x2Cond = @(x) (x(1)^2<=x(2) && x(2)<=2*x(1));
        G = {x1Cond,x2Cond};

        % limitations for every element in x
        a(1) = 0; b(1) = 2;
        a(2) = 0; b(2) = 4;

        [I,c] = MonteCarlo.ndIntegral(f,a,b,G,N0,t_beta);
        disp('First approximation of integral value:');
        disp(I);

        % computing min necessary N
        N = ceil((c/maxError)^2);

        if (N > N0)
            disp('Minimal necessary N:');
            disp(N);

            % computing more correct integral value
            [I] = MonteCarlo.ndIntegral(f,a,b,G,N,t_beta);
            disp('Integral value:');
            disp(I);
        end
    end

    %% test3d: computing 3-dimensional definite integral
    function [I] = test3d(N0, maxError, t_beta)
        f = @(x) 10*x(1);

        % conditions of G area
        x1Cond = @(x) (0<=x(1) && x(1)<=1);
        x2Cond = @(x) (0<=x(2) && x(2)<=sqrt(1-x(1)^2));
        x3Cond = @(x) (0<=x(3) && x(3)<=((x(1)^2+x(2)^2)/2));
        G = {x1Cond,x2Cond,x3Cond};

        % limitations for every element in x
        a(1) = 0; b(1) = 1;
        a(2) = 0; b(2) = 1;
        a(3) = 0; b(3) = 1;

        [I,c] = MonteCarlo.ndIntegral(f,a,b,G,N0,t_beta);
        disp('First approximation of integral value:');
        disp(I);

        % computing min necessary N
        N = ceil((c/maxError)^2);

        if (N > N0)
            disp('Minimal necessary N:');
            disp(N);

            % computing more correct integral value
            [I] = MonteCarlo.ndIntegral(f,a,b,G,N,t_beta);
```

```
                disp('Integral value:');
                disp(I);
            end
        end

        %% test6d: computing 6-dimensional definite integral
        function [I] = test6d(N0, maxError, t_beta)
            % Solving the problem of the
            % mutual attraction of two material bodies.

            gravityConst = 6.67e-11; % gravitational constant
            m1 = 6e+24; % mass of the Earth
            m2 = 7.35e+22; % mass of the Moon
            r = 384467000; % distance between Earth and Moon
            p1 = 5520; % avg density of Earth
            p2 = 3346; % avg density of Moon
            R1 = 6367000; % Earth radius
            R2 = 1737000; % Moon radius

            dist = @(x) sqrt((x(1)-(r+x(4)))^2 + ...
                             (x(2)-x(5))^2 + ...
                             (x(3)-x(6))^2);

            fx = @(x) (x(1)-(r+x(4))) / ((dist(x))^3);
            fy = @(x) (x(2)-x(5)) / ((dist(x))^3);
            fz = @(x) (x(3)-x(6)) / ((dist(x))^3);

            % conditions of G area
            x1Cond = @(x) (x(1)^2+x(2)^2+x(3)^2<=R1^2);
            x2Cond = @(x) (x(4)^2+x(5)^2+x(6)^2<=R2^2);
            G = {x1Cond,x2Cond};

            % limitations for every element in x
            a(1) = -R1; b(1) = R1;
            a(2) = -R1; b(2) = R1;
            a(3) = -R1; b(3) = R1;
            a(4) = -R2; b(4) = R2;
            a(5) = -R2; b(5) = R2;
            a(6) = -R2; b(6) = R2;

            [FxI,c] = MonteCarlo.ndIntegral(fx,a,b,G,N0,t_beta);
            Fx = gravityConst * p1 * p2 * FxI;
            Nx = ceil((c/maxError)^2)

            [FyI,c] = MonteCarlo.ndIntegral(fy,a,b,G,N0,t_beta);
            Fy = gravityConst * p1 * p2 * FyI;
            Ny = ceil((c/maxError)^2)

            [FzI,c] = MonteCarlo.ndIntegral(fz,a,b,G,N0,t_beta);
            Fz = gravityConst * p1 * p2 * FzI;
            Nz = ceil((c/maxError)^2)

            I = sqrt(Fx^2 + Fy^2 + Fz^2);
            disp('Integral value:');
            disp(I);
```

```

            F_correct = gravityConst * m1 * m2 / r^2;

            diff = abs(I - F_correct);
            disp('Difference with correct answer:');
            disp(diff);
        end

    end

end
```