**Project 1: Collision Probability Calculation for a Circular Cell**

2015-22938 김동훈

1. Collision Probability Method for Circular Pin Cell   
   길이 방향으로 무한히 길고, 축이 같은 실린더로 이루어진 Circular Pin 문제에 Collision Probability Method를 적용하여 다음과 같은 관계식의 각 항을 구해본다.  
   (각 영역에서의 average Flux)   
   i : Region Index.  
   : 경계면에서 단위 incoming current가 들어올 때 i영역에 생기는 Flux  
   (**partial flux due to incoming neutrons**)  
   : 경계면에서 들어오는 current   
   : i영역의 source (문제에서 input으로 줄 것이다.)  
   : k영역에 단위 source가 있을 때 i영역에서의 Flux  
   (**partial flux in region i due to internal source in region k**)  
   1. **Absorption Blackness**  and **Total Absorption Blackness**    
         
      Total Absorption Blackness는 각 영역의 Absorption Blackness의 총합으로 정의한다.  
         
      (단위 incoming current당  의 전체 removal rate(우변)는 단위 incoming current당 Absorption rate(좌변)와 같다.)
   2. 와 albedo  
        
         
      여기서 는 **albedo**이고 정의는 albedo의 두 가지 정의 중 와 중 후자이다.  즉 albedo가 0일때의 값이다. (albedo가 0일 때 Black, 0과 1사이일 때 Gray, 1일 때 Reflective B.C. 이다.)  
      만 따로 때서 **Boundary multiplication factor**라고 정의하고 라고 쓰면 라고 간단히 표현할 수 있다.
   3.   
      : 에 unit source density가 있을 때 경계면에 도달하는 중성자수  
      : 에 하나의 중성자가 있을 때 경계면에 도달하는 중성자수(**Escape Probability**)  
      위 정의에 의해 가 성립한다.  
      여기서 알베도가 주어졌을 때 경계면에서 반사되서 되돌아오는 중성자의 수는 이다. 따라서 k에 unit source density가 있을 때 i영역에 돌아오는 중성자의 Flux는 이다.  
      
   4. Relation between partial flux due to incoming neutrons() and    
        
      
   5. 를 구하는 방법  
      영역 k에 unit source density가 있다고 가정하면 아래와 같은 Balance Equation을 얻을 수 있다.   
          
      의 에 k에 unit source density가 있을 때 j의 flux인 를 대입하고,  에 를 대입하고, 모든 j영역에 대해 합하면 위의 식이 된다.   
       이를 행렬의 형태로 정리하면 아래와 같다.  
       for k=1,…,n  
      이제 는 LU factorization후 forward substitution, backward substitution을 통해 구한다. 이는 뒤에 코드와 함께 설명한다.
   6. 를 구하는 방법  
      경계에서 하나의 중성자가 들어오는 상황을 가정하면 아래와 같은 Balance Equation을 얻을 수 있다.  
        
      의 에 하나의 중성자가 경계면으로 들어올 때 j의 flux인 를 대입한다.  
      의 정의가 경계면에 cosine current가 들어올 때 region i에서 first collision 할 확률인데 지금 경계면에서 하나의 중성자가 들어오는 상황이므로 가 들어가는 항을 모든 j영역에 대해 합하면 으로 쓸 수 있고 따라서 위의 식이 된다.   
      이를 행렬의 형태로 쓰면 아래와 같다.  
        
      우변에 나오는  는 Collision Probability Kernel로 구할 수 있다.  
        
      i영역에서 충돌하지 않고 경계면에 도착할 확률은 전체 확률 1에서 i영역에서 다른 영역에서 반응할 확률을 모두 더해서 뺀 것과 같다.  
         
      위의 식은 first collision escape probability와 first collision absorption blackness의 관계식에 위의 first collision escape probability와 collision probability사이의 식을 대입한 뒤 collision probability를 collision probability kernel로 표현한 식이다.  
        
      따라서 마)의 n개의 Linear System과 바)의 하나의 Linear System을 풀면 n개의  와 를 알 수 있다.
   7.  (Total number of neutrons reaching boundary first time)   
      
   8. Total number of neutrons escaping and incoming

 ,   
위 두 값은 분자가 초항, 공비가 인 무한등비급수의 합을 생각하면 얻을 수 있다.

* 1. Collision Probability Kernel() in Annular Geometry  
     이제 Annular Geometry에서의 Collision Probability Kernel를 구하면 문제를 해결에 필요한 모든 계수를 계산 할 수 있다.
     1. 수평거리 t 만큼 이동할 때 충돌하지 않을 확률  
          
        안으로, 거리 R만큼 간 상황을 생각해보자.  
        안으로 들어갈 비율:    
        모든 가능한 에서 수평거리에 도달할 확률  
          
        안으로 들어갈 비율:   
        로 들어가서 t에 도달할 확률  
          
        







* + 1. Bickley Function의 성질  
       정의:   
       미분:    
       적분:, 
    2. Collision Probability Kernel in Annual Geometry  
         
        ,    
          
         
          
         
         
          
          
         
       Self Collision을 포함한 일반적인 형태는 다음과 같다.  
       









* + 1.  and Gauss Quadrature and Gauss-Jacobi Quadrature  
        ,  ,   
            
       
    2.  and Gauss-Jacobi Quadrature  
          
          
          
         
          
          
         
       

1. 코드구성

|  |
| --- |
| void CollisionProbability::cpman(string inputName) //cp로 문제 하나를 푸는 전체과정을 총괄하는 함수  {  inName = inputName;  readInput();  gaussJacobiQuad(ngauss, x\_i, weight);  calCPKernels();  solveSystem();  } |

한 문제를 푸는 함수를 각각의 함수로 구성하였다.  
  
  
  
  
  
  
  
  
  
다음은 주어지는 input파일을 읽어 저장하는 함수이다.

|  |
| --- |
| void CollisionProbability::readInput()  {  //입력파일이름은 Col..Proba.. 클래스에 string inName 으로 저장되어있다.  string inputName;  inputName = inName;  inputName.append(".txt");  ifstream fin(inputName);  string str;  int i; // for counter  fin >> str; // input\_set\_# 와 빈줄 한줄을 넘어간다.  fin >> str;  nr = atof(str.c\_str());  /\*c\_str is needed to convert string to const char\* previously (the function requires it)\*/  fin >> str;  ngauss = atof(str.c\_str());  fin >> str; // !nr...을 넘어간다.  allocateVaiable();  // 클래스 자체에서 메모리 주소만 정의되어있던 포인터 변수들을 nr, 즉 크기에 맞게 동적할당해준다.  // 소멸자에서 반드시 이에 대응하는 delete(deallocate 함수가 있어야 한다.)  for (i = 0; i < nr; i++) {  fin >> str;  rad[i] = atof(str.c\_str());  }  fin >> str; // !radius 를 넘어간다.  for (i = 0; i < nr; i++) {  fin >> str;  sigt[i] = atof(str.c\_str());  }  fin >> str; // !sigt 를 넘어간다.  for (i = 0; i < nr; i++) {  fin >> str;  sigs[i] = atof(str.c\_str());  }  fin >> str; // !sigs 를 넘어간다.  for (i = 0; i < nr; i++) {  fin >> str;  qi[i] = atof(str.c\_str());  }  fin >> str; // !qi 를 넘어간다.  fin >> str;  albedo = atof(str.c\_str());  fin >> str;  jext = atof(str.c\_str());  fin.close();  } |

아래의 함수는 input을 읽는 중간에 들어가는 함수로 주어진 인풋의 nr값에 맞게 변수들을 동적할당하는 함수이다.

|  |
| --- |
| Void CollisionProbability::allocateVaiable() |
| { |
| int i, j; |
| rad = new real[nr]; |
| memset(rad, 0, sizeof(real)\*(nr)); |
| sigt = new real[nr]; |
| memset(sigt, 0, sizeof(real)\*(nr)); |
| sigs = new real[nr]; |
| memset(sigs, 0, sizeof(real)\*(nr)); |
|  |
| sigr = new real[nr]; |
| memset(sigs, 0, sizeof(real)\*(nr)); |
|  |
|  |
| qi = new real[nr]; |
| memset(qi, 0, sizeof(real)\*(nr)); |
| x\_i = new real[nr]; |
| memset(x\_i, 0, sizeof(real)\*(nr)); |
| weight = new real[nr]; |
| memset(weight, 0, sizeof(real)\*(nr)); |
|  |
| delta = new real[nr]; |
| memset(delta, 0, sizeof(real)\*(nr)); |
|  |
| Sij = new real\*[nr + 1]; |
| for (i = 0; i < (nr + 1); i++) { |
| Sij[i] = new real[nr + 1]; |
| memset(Sij[i], 0, sizeof(real)\*(nr + 1)); |
| } |
|  |
| Sijk = new real\*\*[nr + 1]; |
| for (i = 0; i < (nr + 1); i++) { |
| Sijk[i] = new real\*[nr + 1]; |
| for (j = 0; j < (nr + 1); j++) { |
| Sijk[i][j] = new real[nr]; |
| memset(Sijk[i][j], 0, sizeof(real)\*(nr)); |
| } |
| }//// |
|  |
| Pij = new real\*[nr]; |
| for (i = 0; i < nr; i++) { |
| Pij[i] = new real[nr]; |
| memset(Pij[i], 0, sizeof(real)\*nr); |
| } |
|  |
| vol = new real[nr]; |
| memset(vol, 0, sizeof(real)\*(nr)); |
| ci = new real[nr]; |
| memset(ci, 0, sizeof(real)\*(nr)); |
|  |
| rho = new real\*[nr + 1]; |
| for (i = 0; i < (nr + 1); i++) { |
| rho[i] = new real[ngauss]; |
| memset(rho[i], 0, sizeof(real)\*(ngauss)); |
| } |
| gamma = 0.; |
| /// |
| removal = new real[nr]; |
| memset(sigs, 0, sizeof(real)\*(nr)); |
|  |
| } |

아래의 함수는 가우스 자코비 쿼드러쳐를 계산하는 함수이다. 이 값들은 문제에서 주어지는 gausspoint에 따라 정해지는 값이므로 문제를 풀기 전에 input을 읽은 후 미리 계산해둔다.

|  |
| --- |
| void CollisionProbability::gaussJacobiQuad(int inGaussNum, real \* x\_in, real \*weight\_in) |
| { //Gauss Jacobi Quadrature |
|  |
| switch (inGaussNum) |
| { |
| case 1: |
| x\_in[0] = 0.6666666667; |
|  |
| weight\_in[0] = 0.5000000000; |
|  |
| break; |
| case 2: |
| x\_in[0] = 0.3550510257; |
| x\_in[1] = 0.8449489743; |
|  |
| weight\_in[0] = 0.1819586183; |
| weight\_in[1] = 0.3180413817; |
|  |
| break; |
| case 3: |
| x\_in[0] = 0.2123405382; |
| x\_in[1] = 0.5905331356; |
| x\_in[2] = 0.9114120405; |
|  |
| weight\_in[0] = 0.0698269799; |
| weight\_in[1] = 0.2292411064; |
| weight\_in[2] = 0.2009319137; |
|  |
| break; |
| case 4: |
| x\_in[0] = 0.1397598643; |
| x\_in[1] = 0.4164095676; |
| x\_in[2] = 0.7231569864; |
| x\_in[3] = 0.9428958039; |
|  |
| weight\_in[0] = 0.0311809710; |
| weight\_in[1] = 0.1298475476; |
| weight\_in[2] = 0.2034645680; |
| weight\_in[3] = 0.1355069134; |
|  |
| break; |
| case 5: |
| x\_in[0] = 0.0985350858; |
| x\_in[1] = 0.3045357266; |
| x\_in[2] = 0.5620251898; |
| x\_in[3] = 0.8019865821; |
| x\_in[4] = 0.9601901429; |
|  |
| weight\_in[0] = 0.0157479145; |
| weight\_in[1] = 0.0739088701; |
| weight\_in[2] = 0.1463869871; |
| weight\_in[3] = 0.1671746381; |
| weight\_in[4] = 0.0967815902; |
|  |
| break; |
| default: |
| cout << "check the ngauss in the input file is proper." << endl; |
| break; |
| } |
| } |

아래의 함수는 각위치와 가우스포인트에 따른 를 계산하여, 를 계산하고,  를 계산하는 함수이다.

|  |
| --- |
| void CollisionProbability::calCPKernels() |
| { |
| int i, j, k,m; |
| real temp, sum; |
| real y; |
|  |
| delta[0] = rad[0]; |
|  |
| for (i = 0; i < nr; i++) { |
| sigr[i] = sigt[i] - sigs[i ]; |
| } |
| for (i = 1; i < nr; i++) { |
| delta[i] = rad[i] - rad[i - 1]; |
| } |
| vol[0] = rad[0] \* rad[0] \* \_pi; |
| for (i = 1; i < nr; i++) { |
| vol[i] = (rad[i] \* rad[i] - rad[i - 1] \* rad[i - 1])\*\_pi; |
| } |
| for (i = 0; i < nr; i++) { |
| ci[i] = (sigs[i] / sigt[i]); |
| } |
|  |
| for (k = 0; k < nr; k++) { |
| // calculate rho |
| for (m = 0; m < ngauss; m++) { |
| for (i = k; i < nr; i++) { |
| if (i == k) { |
| y = rad[k] - x\_i[m] \* x\_i[m] \* delta[k]; |
| rho[i + 1][m] = (sqrt(rad[i] \* rad[i] - y\*y))\*sigt[i];// +rho[i][m]; |
| } |
| else { |
| y = rad[k] - x\_i[m] \* x\_i[m] \* delta[k]; |
| rho[i + 1][m] = (sqrt(rad[i] \* rad[i] - y\*y) - sqrt(rad[i - 1] \* rad[i - 1] - y\*y))\*sigt[i] + rho[i][m]; |
| } |
| } |
| } |
|  |
| //cal Sijk |
| for (i = k + 1; i < (nr + 1); i++) { |
| for (j = i; j < (nr + 1); j++) { |
| for (m = 0; m < ngauss; m++) { |
| Sijk[i][j][k] += 2 \* delta[k] \* (weight[m])\*(calBickeley(rho[j][m]+ rho[i][m]) - calBickeley(rho[j][m]- rho[i][m])); |
| } |
| Sij[i][j] += Sijk[i][j][k]; |
| } |
| } |
| } |
|  |
| for (i = 1; i < (nr + 1); i++) { |
| for (j = 0; j < i; j++) { |
| Sij[i][j] = Sij[j][i]; |
| } |
| } |
|  |
| for (i = 0; i < nr; i++) { |
| for (j = 0; j < nr; j++) { |
| //calPij(i, j); |
| if (i == j) sum = vol[i] \* sigt[i]; |
| else |
| { |
| sum = 0.; |
| } |
| Pij[i][j] = sum + 2 \* (Sij[i + 1][j + 1] + Sij[i][j] - Sij[i][j + 1] - Sij[i + 1][j]); |
| } |
| } |
| } |

위의 함수에 나오는 calBickeley함수는 아래와 같은데 숙제에서 주어진 포트란 함수를 C++형태로 바꾸어서 만들었다.

|  |
| --- |
| real CollisionProbability::calBickeley(real x\_in) |
| { |
| // FUNCTION KI3(XX) |
| // C / HEBE, 02 Jan 1996 ---- - By Rudi Stamm'ler \* \* |
| // C ACCURACY 1.0E-5, UP TO XX = 9.0. |
|  |
| double KI3; |
| double X = fabs(x\_in); |
| int i; |
|  |
| if (X > 0.99999) |
| { |
| if (X > 9.0) { |
| KI3 = 0.0; |
| } |
| else |
| { |
| i = floor(2.5\*(X - 0.99998)) + 1; |
|  |
| switch (i) |
| { |
| // % C \*\* RANGE 1.0 - 1.4 \*\* |
| case 1: |
| KI3 = ((-0.05337485\*X + 0.3203223)\*X - 0.7538355)\*X + 0.7247294; |
| break; |
| //% C \*\* RANGE 1.4 - 1.8 \*\* |
| case 2: |
| KI3 = ((-0.03146833\*X + 0.2295280)\*X - 0.6279752)\*X + 0.6663720; |
| break; |
| //% C \*\* RANGE 1.8 - 2.2 \*\* |
| case 3: |
| KI3 = ((-0.01906198\*X + 0.1631667)\*X - 0.5094124)\*X + 0.5956163; |
| break; |
| //% C \*\* RANGE 2.2 - 2.6 \*\* |
| case 4: |
| KI3 = ((-0.01174752\*X + 0.1152418)\*X - 0.4046007)\*X + 0.5191031; |
| break; |
| //% C \*\* RANGE 2.6 - 3.0 \*\* |
| case 5: |
| KI3 = ((-0.007328415\*X + 0.08097913)\*X - 0.3159648)\*X + 0.4425954; |
| break; |
| //% C \*\* RANGE 3.0 - 3.4 \*\* |
| case 6: |
| KI3 = ((-0.004617254\*X + 0.05669960)\*X - 0.2434341)\*X + 0.3703178; |
| break; |
| //% C \*\* RANGE 3.4 - 3.8 \*\* |
| case 7: |
| KI3 = (0.007923547\*X - 0.07158569)\*X + 0.1684022; |
| break; |
| //% C \*\* RANGE 3.8 - 4.2 \*\* |
| case 8: |
| KI3 = (0.005095111\*X - 0.05016344)\*X + 0.1278307; |
| break; |
| //% C \*\* RANGE 4.2 - 4.6 \*\* |
| case 9: |
| KI3 = (0.003286040\*X - 0.03501524)\*X + 0.09611422; |
| break; |
| //% C \*\* RANGE 4.6 - 5.0 \*\* |
| case 10: |
| KI3 = (0.002126242\*X - 0.02437465)\*X + 0.07170491; |
| break; |
| //% C \*\* RANGE 5.0 - 5.8 \*\* |
| case 11: |
| KI3 = (0.001123687\*X - 0.01425519)\*X + 0.04616317; |
| break; |
| //% C \*\* RANGE 5.8 - 6.6 \*\* |
| case 12: |
| KI3 = (0.001123687\*X - 0.01425519)\*X + 0.04616317; |
| break; |
| //% C \*\* RANGE 5.8 - 6.6 \*\* |
| case 13: |
| KI3 = (4.762937E-4\*X - 6.810124E-3)\*X + 0.02475115; |
| break; |
| //% C \*\* RANGE 6.6 - 7.4 \*\* |
| case 14: |
| KI3 = (4.762937E-4\*X - 6.810124E-3)\*X + 0.02475115; |
| break; |
| //% C \*\* RANGE 6.6 - 7.4 \*\* |
| case 15: |
| KI3 = (2.031843E-4\*X - 3.232035E-3)\*X + .01302864; |
| break; |
| //% C \*\* RANGE 7.4 - 8.2 \*\* |
| case 16: |
| KI3 = (2.031843E-4\*X - 3.232035E-3)\*X + .01302864; |
| break; |
| //% C \*\* RANGE 7.4 - 8.2 \*\* |
| case 17: |
| KI3 = (8.701440E-5\*X - 1.524126E-3)\*X + 6.749972E-3; |
| break; |
| //% C \*\* RANGE 8.2 - 9.0 \*\* |
| case 18: |
| KI3 = (8.701440E-5\*X - 1.524126E-3)\*X + 6.749972E-3; |
| break; |
| //% C \*\* RANGE 8.2 - 9.0 \*\* |
| case 19: |
| KI3 = (3.742673E-5\*X - 7.157367E-4)\*X + 3.454768E-3; |
| break; |
| case 20: |
| KI3 = (3.742673E-5\*X - 7.157367E-4)\*X + 3.454768E-3; |
| break; |
| case 21: |
| KI3 = (3.742673E-5\*X - 7.157367E-4)\*X + 3.454768E-3; |
| break; |
| } |
| } |
| } |
| else { |
| i = floor(20.0\*X + 1.00001); |
| switch (i) |
| { |
| // % C \*\* RANGE 0.00 - 0.05 \*\* |
| case 1: |
| KI3 = (0.7266088\*X - 0.9990226)\*X + 0.7853961; |
| break; |
| //% C \*\* RANGE 0.05 - 0.10 \*\* |
| case 2: |
| KI3 = (0.6466375\*X - 0.9912340)\*X + 0.7852024; |
| break; |
| //% C \*\* RANGE 0.10 - 0.15 \*\* |
| case 3: |
| KI3 = (0.5856605\*X - 0.9791293)\*X + 0.7845986; |
| break; |
| //% C \*\* RANGE 0.15 - 0.20 \*\* |
| case 4: |
| KI3 = (0.5346648\*X - 0.9638914)\*X + 0.7834577; |
| break; |
| //% C \*\* RANGE 0.20 - 0.25 \*\* |
| case 5: |
| KI3 = (0.4907827\*X - 0.9463843)\*X + 0.7817094; |
| break; |
| //% C \*\* RANGE 0.25 - 0.30 \*\* |
| case 6: |
| KI3 = (0.4521752\*X - 0.9271152)\*X + 0.7793031; |
| break; |
| //% C \*\* RANGE 0.30 - 0.35 \*\* |
| case 7: |
| KI3 = (0.4177388\*X - 0.9064822)\*X + 0.7762107; |
| break; |
| //% C \*\* RANGE 0.35 - 0.40 \*\* |
| case 8: |
| KI3 = (0.3869945\*X - 0.8849865)\*X + 0.7724519; |
| break; |
| //% C \*\* RANGE 0.40 - 0.45 \*\* |
| case 9: |
| KI3 = (0.3590753\*X - 0.8626685)\*X + 0.7679903; |
| break; |
| //% C \*\* RANGE 0.45 - 0.50 \*\* |
| case 10: |
| KI3 = (0.3338676\*X - 0.8400133)\*X + 0.7628988; |
| break; |
| //% C \*\* RANGE 0.50 - 0.60 \*\* |
| case 11: |
| KI3 = (0.2998569\*X - 0.8054172)\*X + 0.7540982; |
| break; |
| //% C \*\* RANGE 0.60 - 0.70 \*\* |
| case 12: |
| KI3 = (0.2998569\*X - 0.8054172)\*X + 0.7540982; |
| break; |
| //% C \*\* RANGE 0.60 - 0.70 \*\* |
| case 13: |
| KI3 = (0.2609154\*X - 0.7587821)\*X + 0.7401279; |
| break; |
| //% C \*\* RANGE 0.70 - 0.80 \*\* |
| case 14: |
| KI3 = (0.2609154\*X - 0.7587821)\*X + 0.7401279; |
| break; |
| //% C \*\* RANGE 0.70 - 0.80 \*\* |
| case 15: |
| KI3 = (0.2278226\*X - 0.7125290)\*X + 0.7239594; |
| break; |
| //% C \*\* RANGE 0.80 - 0.90 \*\* |
| case 16: |
| KI3 = (0.2278226\*X - 0.7125290)\*X + 0.7239594; |
| break; |
| //% C \*\* RANGE 0.80 - 0.90 \*\* |
| case 17: |
| KI3 = (0.1994999\*X - 0.6672761)\*X + 0.7058777; |
| break; |
| //% C \*\* RANGE 0.90 - 1.00 \*\* |
| case 18: |
| KI3 = (0.1994999\*X - 0.6672761)\*X + 0.7058777; |
| break; |
| //% C \*\* RANGE 0.90 - 1.00 \*\* |
| case 19: |
| KI3 = (0.1751248\*X - 0.6234536)\*X + 0.6861762; |
| break; |
| case 20: |
| KI3 = (0.1751248\*X - 0.6234536)\*X + 0.6861762; |
| break; |
| } |
| } |
|  |
| return KI3; |
| } |

아래의 함수는 행렬연산을 위한 변수선언부분, 행렬과 백터를 정의하여 문제를 푸는 부분, 결과값을 파일로 출력하는 세가지 부분으로 구성되어 있다. N+1개의 linear system이 행렬 A는 공유하므로 LU factorization은 한번만 해서 저장해두고, forward substitution과 backward substitution은 하나의 함수로 만들어서 쓸 수 있도록 했다.

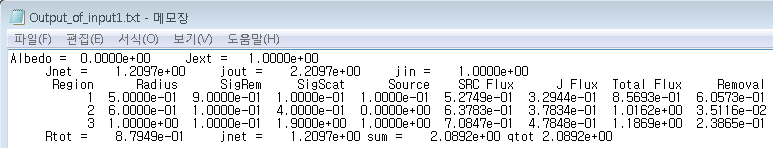
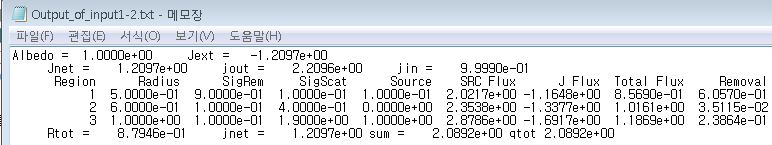
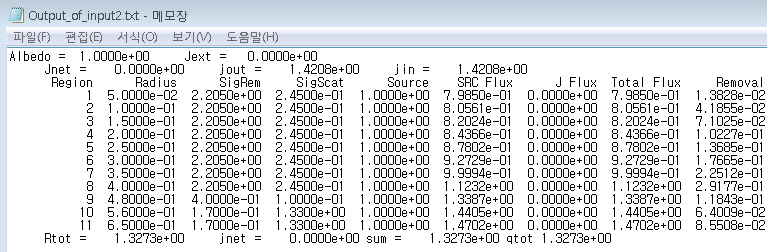
|  |
| --- |
| void CollisionProbability::solveSystem() |
| { |
|  |
| int i, j, k; |
|  |
| b\_y = new real[nr]; |
| memset(b\_y, 0, sizeof(real)\*(nr)); |
|  |
| b\_x = new real[nr]; |
| memset(b\_x, 0, sizeof(real)\*(nr)); |
|  |
| Yi = new real[nr]; |
| memset(Yi, 0, sizeof(real)\*(nr)); |
|  |
| Xiktmp = new real[nr]; |
| memset(Xiktmp, 0, sizeof(real)\*(nr)); |
|  |
| Ya = new real[nr]; |
| memset(Ya, 0, sizeof(real)\*(nr)); |
|  |
| xk = new real[nr]; |
| memset(xk, 0, sizeof(real)\*(nr)); |
|  |
| sourceFlux = new real[nr]; |
| memset(sourceFlux, 0, sizeof(real)\*(nr)); |
|  |
| J\_Flux = new real[nr]; |
| memset(J\_Flux, 0, sizeof(real)\*(nr)); |
|  |
| Flux = new real[nr]; |
| memset(Flux, 0, sizeof(real)\*(nr)); |
|  |
| A = new real\*[nr]; |
| for (i = 0; i < nr; i++) { |
| A[i] = new real[nr]; |
| memset(A[i], 0, sizeof(real)\*nr); |
| } |
|  |
| Xik = new real\*[nr]; |
| for (i = 0; i < nr; i++) { |
| Xik[i] = new real[nr]; |
| memset(Xik[i], 0, sizeof(real)\*nr); |
| } |
|  |
| Xika = new real\*[nr]; |
| for (i = 0; i < nr; i++) { |
| Xika[i] = new real[nr]; |
| memset(Xika[i], 0, sizeof(real)\*nr); |
| } |
|  |
|  |
| real totalx; |
| real Jout, Jin; |
|  |
| real deno; |
| real temp; |
|  |
| // construct matrix A |
|  |
| for (i = 0; i < nr; i++) { |
| for (j = 0; j < nr; j++) { |
| A[i][j] = -Pij[i][j] \* ci[j]; |
| } |
| A[i][i] = A[i][i] + sigt[i] \* vol[i]; |
| } |
|  |
| // construct vector b\_y for Yi |
| for (i = 0; i < nr; i++) { |
| temp = sigt[i] \* vol[i]; |
| for (j = 0; j < nr; j++) { |
| temp = temp -Pij[i][j]; |
| } |
| b\_y[i] = 4.0 / (2\*\_pi\*rad[nr-1])\*temp; |
| } |
|  |
| // LU Factorization |
| luFactorization(A); |
|  |
| //calculate Yi |
| solveLU(A, Yi, b\_y); |
|  |
|  |
| // construct vector b\_x for Xik |
|  |
| for (k = 0; k < nr; k++) { |
| for (i = 0; i < nr; i++) { |
| b\_x[i] = Pij[k][i] / sigt[k]; |
| } |
| //calculate Xiktmp |
| solveLU(A, Xiktmp, b\_x); |
|  |
| for (i = 0; i < nr; i++) { |
| Xik[i][k] = Xiktmp[i]; |
| } |
| } |
|  |
| //cal gamma |
| for (i = 0; i < nr; i++) { |
| gamma += (sigt[i] - sigs[i])\*Yi[i]\*vol[i]; |
| } |
|  |
| deno = 1 - (1 - gamma)\*albedo; |
| for (i = 0; i < nr; i++) { |
| Ya[i] = Yi[i] / deno; |
| } |
|  |
| for (i = 0; i < nr; i++) { |
| xk[i] = \_pi\*rad[nr - 1] / 2 \* vol[i] \* Yi[i] ; |
| } |
|  |
| totalx = 0; |
| for (i = 0; i < nr; i++) { |
| totalx += qi[i] \* xk[i] ; |
| } |
|  |
| for (k = 0; k < nr; k++) { |
|  |
| for (i = 0; i < nr; i++) { |
| Xika[i][k] = Xik[i][k]+albedo\*xk[k] \*Ya[i] ; |
| } |
| } |
|  |
| Jout = (totalx + (1 - gamma)\*jext) / (1 - albedo\*(1 - gamma)); |
| Jin = (albedo\*totalx + jext) / (1 - albedo\*(1 - gamma)); |
|  |
| for (i = 0; i < nr; i++) { |
|  |
| for (k = 0; k < nr; k++) { |
| sourceFlux[i] += Xika[i][k] \* qi[k]; |
| } |
| J\_Flux[i] = Ya[i] \* jext ; |
| Flux[i] = J\_Flux[i] + sourceFlux[i] ; |
| } |
|  |
|  |
|  |
| real Rtot=0; |
| real Jnet; |
| Jnet = Jout - Jin; |
|  |
| real qtot = 0; |
|  |
| for (i = 0; i < nr; i++) { |
| removal[i] = sigr[i] \* vol[i] \* Flux[i] ; |
| Rtot += removal[i] ; |
| qtot += vol[i] \* qi[i]; |
| } |
|  |
| string outName; |
| outName = "Output\_of\_"; |
| outName.append(inName); |
| outName.append(".txt"); |
|  |
| ofstream fout1(outName); |
|  |
| //fout1.setf(ios::fixed); |
| fout1.setf(ios::scientific); |
| fout1 << setprecision(4); |
| fout1 << "Albedo = " << albedo << " Jext = " << jext << endl; |
| fout1 << " Jnet =" <<setw(14)<< Jnet<< " jout =" <<setw(14)<< Jout << " jin =" << setw(14) << Jin << endl; |
| fout1 << " Region Radius SigRem SigScat Source SRC Flux J Flux Total Flux Removal" << endl; |
| for (i = 0; i < nr; i++) |
| fout1 << setw(12)<<(i+1)<<setw(12)<< rad[i] << setw(12) << sigr[i] << setw(12) << sigs[i] << setw(12) << qi[i]<< setw(12)<<sourceFlux[i] << setw(12) << J\_Flux[i] << setw(12) << Flux[i] <<setw(12) <<removal[i] << endl; |
| fout1 << " Rtot =" << setw(14) << Rtot << " jnet =" << setw(14) << Jnet << " sum =" << setw(14) << qtot << " qtot "<< qtot <<endl; |
|  |
| fout1.close(); |
| } |

LU factorization의 대상이되는 행렬의 꼴이 아래와 같다.   
피벗이 필요 없을 것으로 판단되어 피벗이 없는 lu factorization함수를 짰다.

|  |
| --- |
| void CollisionProbability::luFactorization(real \*\* matA) |
| { |
| int i, j, k, i\_temp, j\_temp; |
| real temp; |
|  |
| for (j = 0; j < nr-1 ; j++) { // 0:nr-2 -> 1:nr-1 |
| for (i = j+1; i < nr; i++) { // j:nr-1 -> j+1:nr |
| matA[i][j] = matA[i][j] / matA[j][j]; |
| } |
| for (k = j+1; k < nr; k++) { // j:nr-1 -> j+1:nr |
| for (i = j+1; i < nr; i++) { // j:nr-1 -> j+1:nr |
| matA[k][i] = matA[k][i] - matA[k][j] \* matA[j][i]; |
| } |
| } |
| } |
| } |

아래의 함수는 forward substitution과 backward substitution을 하는 함수이다.

|  |
| --- |
| void CollisionProbability::solveLU(real \*\* matA, real \* vec\_x, real \* vec\_b) |
| { |
| int i, j, k, i\_temp,j\_temp; |
| real temp; |
|  |
| // Forward substitution |
| vec\_x[0] = vec\_b[0]; |
| for (i = 1; i < nr;i++) { |
| temp = 0; |
| for (j = 0; j < i ;j++) { |
| temp = temp + matA[i][j]\*vec\_x[j]; |
| } |
| vec\_x[i] = (vec\_b[i] - temp); |
| } |
| // Backward substitution |
| vec\_x[nr-1] = vec\_x[nr-1] / matA[nr-1][nr-1]; |
| for (i = nr-2; i>=0 ; i--) { |
| temp = 0; |
| for (j= nr-1; j >i ; j--) { |
| temp = temp + matA[i][j]\*vec\_x[j]; |
| } |
| vec\_x[i] = (vec\_x[i] - temp) / matA[i][i]; |
| } |
| } |

1. 코드검증
   1. 주어진 두 문제의 결과
      1. 문제1-1  
         
      2. 문제1-2  
         
      3. 문제2  
           
         결과가 문제에서 주어진 결과와 정확히 일치한다.
   2. Gauss Quadrature와 Gauss-Jacobi Quadrature 각각의 사용시 해의 정확도 비교  
      문제가 좀더 큰 input2를 사용하여 정확도를 비교하였다.

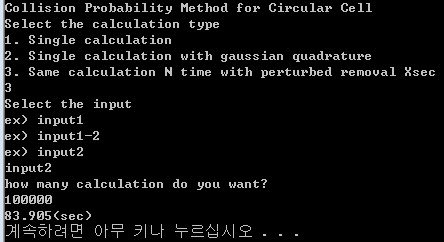
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Gauss Jacobi Quadrature | | | | |
|  | Flux | Relative error | | |
| Region | m=5 | 4 | 3 | 2 |
| 1 | 7.98500E-01 | -0.0006 | 0.0019 | 0.0462 |
| 2 | 8.05606E-01 | 0.0004 | 0.0001 | -0.0073 |
| 3 | 8.20240E-01 | -0.0001 | -0.0007 | -0.0013 |
| 4 | 8.43662E-01 | 0.0005 | -0.0008 | -0.0012 |
| 5 | 8.78016E-01 | -0.0003 | 0.0005 | -0.0025 |
| 6 | 9.27291E-01 | 0.0000 | 0.0002 | -0.0032 |
| 7 | 9.99937E-01 | 0.0001 | 0.0001 | -0.0051 |
| 8 | 1.12320E+00 | 0.0000 | 0.0000 | 0.0116 |
| 9 | 1.33872E+00 | 0.0007 | -0.0007 | -0.0112 |
| 10 | 1.44053E+00 | -0.0007 | -0.0007 | 0.0014 |
| 11 | 1.47022E+00 | -0.0007 | -0.0007 | -0.0054 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Gauss Quadrature | | | | |
|  | Flux | Relative error | | |
| Region | m=5(Gauss Jacobi) | 4 | 3 | 2 |
| 1 | 7.98500E-01 | -7.0686 | -13.8827 | -1.8284 |
| 2 | 8.05606E-01 | -2.3727 | -3.1139 | -1.6375 |
| 3 | 8.20240E-01 | -1.3001 | -1.5416 | -1.0202 |
| 4 | 8.43662E-01 | -0.8289 | -0.9146 | -0.6817 |
| 5 | 8.78016E-01 | -0.5494 | -0.5824 | -0.4503 |
| 6 | 9.27291E-01 | -0.3442 | -0.3894 | -0.2461 |
| 7 | 9.99937E-01 | -0.1474 | -0.3119 | 0.0103 |
| 8 | 1.12320E+00 | 0.4033 | -0.7176 | 0.6205 |
| 9 | 1.33872E+00 | 2.0422 | 4.6410 | 0.7821 |
| 10 | 1.44053E+00 | 1.2877 | 2.5185 | 0.5213 |
| 11 | 1.47022E+00 | 1.1855 | 2.8343 | 0.3721 |

Gauss Jacobi Quadrature를 사용한 경우 m(number of gauss point)가 클수록 상대오차상 줄어들었다. 또한 Gauss Quadrature에 비해 상대오차가 10000배가량 차이가 났다. Gauss Quadrature의 경우 오차가 너무 커서 m의 변화에 따른 상대오차의 변화가 오차에 비해 작아서 경향이 있다고 보기 어렵다.

* 1. 반복수행 시간

|  |
| --- |
| for (i = 0; i < n; i++) { |
| CollisionProbability cp; |
| cp.inName = name; |
| cp.cpman\_PerturbRemovalXsec(name); |
| }  void CollisionProbability::cpman\_PerturbRemovalXsec(string inputName)  {  {  inName = inputName;  readInput();  gaussJacobiQuad(ngauss, x\_i, weight);  manipulateRemovalXsec(0.03);  calCPKernels();  solveSystem();  }  }  void CollisionProbability::manipulateRemovalXsec(real rate)  {  srand(time(NULL));  int i;  real rn;  for (i = 0; i < nr; i++) {  rn = (-1.0+2.\*(rand()/RAND\_MAX))\*rate;  sigt[i] = (sigt[i] - sigs[i])\*(1 + rn) + sigs[i];  }  } |

calCPKernels(); 전에 removal Xsec에 3% perturbation을 주고 반복계산하도록 하였다.  


83.905초가 걸렸다.

1. Physics Examination
   1. 서로 다른 흡수 핵반응 단면적에 따른 결과  
      이 경우에 11개의 동축실린더형태로 이루어진 input2를 이용하였다. Spatial self shielding 효과를 확인하기 위하여 removal XS가 큰 중앙의 8개의 영역의 removal XS를 바꾸어가며 비교를 했다.  
        
        
      Spatail self shiending이 없다면 Removal rate의 개형이 XS의 분포와 같은 경향을 보여야 하지만 아래 그래프와 같이 removal XS가 커지는 8번째 영역에서 removal rate가 크고 안쪽의 기여는 줄어드는 것을 볼 수 있다.  
        
        
        
        
      Flux distribution은 위와 같고, normalize한 그래프는 아래와 같다.