СУ “Климент Охридски” – Физически Факултет

РЕФЕРАТ

Адаптивни и рекурсивни процедури

за Монте Карло интегриране

Пламен Станиславов Стаменов

София

2002

РЕФЕРАТ – *Адаптивни и рекурсивни процедури…*

Пламен Стаменов

1. Увод

Числените методи черпят вдъхновение от най-различни източници. Както е известно “Сплайн” е било изначално наименованието на еластичните дървени пръчки, които са използвали чертожниците за да чертаят криви. “Симулирано отгряване” е названието на друг числен метод, на основата на термодинамична аналогия. Кой, не би се засмял при мисълта за славата на Монте Карло.

Като оставим на страна насмешливата страна на въпроса, методите Монте Карло, както ще стане ясно по-късно са съвсем сериозни числени процедури. Те са дори, понякога, единствената алтернатива (например за пространства с гигантски размерности). Обхватът на различните процедури за интегриране е драстично различен. Съществуват както и строго специализирани, оптимизирани и бързи процедури изпълняващи конкретни задачи, като симулации на магнитни или ядрени структури; така и общи и сравнително “тромави” процедури за общо многомерно интегриране.

2. Елементарно Монте Карло интегриране

Да предположим, че сме избрали N равномерно разпределени точки в многомерния обем V. Нека ги наречем x_1, x_2, x_N. Основната теорема на Монте Карло интегрирането дава оценка на интеграла на многомерна функция f над многомерния обем,

$$\int f dV \approx V < f > \pm V \sqrt{\frac{< f^2 > - < f >^2}{N}}$$

Тук ъгловите скоби означават средна стойност по всички N случайно избрани точки,

$$< f > \equiv \frac{1}{N} \sum_{i=1}^{N} f(x_i) \qquad < f^2 > \equiv \frac{1}{N} \sum_{i=1}^{N} f^2(x_i)$$

Знакът “плюс или минус” в горния израз означава едно стандартно отклонение, като оценка на грешката на интеграла, а далеч не твърда граница, още повече, че няма никаква гаранция, че

грешката е разпределена гаусово; ето защо тази величина следва да се разглежда само като груба оценка на възможната грешка.

Нека предположим, че искаме да интегрираме функцията *g* върху област *W*, която не е лесена за покриване от случайни числа. Например, *W* може да има много сложна форма. Това не е проблем. Просто следва да се намери област *V*, която съдържа W, но е надмножество на *V*. Ние, естествено, бихме искали двете области да са възможно най-близки. Това е един не лош начин за подобряване на скоростта, от който алгоритмите интегриращи по Монте Карло, и разбира се, най-вече хората които ги пишат се възползват.

Общите процедури за интегриране са доста усложнени и пълни с детайли, но сравнително прости като идея. Да предположим, че изкаме да намерим центъра на масите на тяло с сложна форма. Първото което следва да проверим е дали интегралите, които следва да пресметнем не могат да се сведат до интеграли с по-малка размерност и да бъдат пресметнати с някоя от класическите квадратурни процедури. Дори и да не можем да сведем интеграла до едномерни, то някоя “хитра” смяна на промеливите може да ни е от полза.

Ето и една примерна процедура, която извършва интегрирането на обема на сечението на тор с две равнини – една доста неприятна област в анаитично отношение:

```
#include "nrutil.h"
n=... Set to the number of sample points desired.
den=... Set to the constant value of the density.
sw=swx=swy=swz=0.0; Zero the various sums to be accumulated.
varw=varx=vary=varz=0.0;
vol=3.0*7.0*2.0; Volume of the sampled region.
for(j=1;j<=n;j++) {
x=1.0+3.0*ran2(&idum); Pick a point randomly in the sampled region.
y=(-3.0)+7.0*ran2(&idum);
z=(-1.0)+2.0*ran2(&idum);
if (z*z+SQR(sqrt(x*x+y*y)-3.0) < 1.0) { Is it in the torus?
sw += den; If so, add to the various cumulants.
swx += x*den;
swy += y*den;
swz += z*den;
varw += SQR(den);
varx += SQR(x*den);
vary += SQR(y*den);
varz += SQR(z*den);
}
}
w=vol*sw/n; The values of the integrals (7.6.5),
x=vol*swx/n;
y=vol*swy/n;
z=vol*swz/n;
dw=vol*sqrt((varw/n-SQR(sw/n))/n); and their corresponding error estimates.
dx=vol*sqrt((varx/n-SQR(swx/n))/n);
dy=vol*sqrt((vary/n-SQR(swy/n))/n);
dz=vol*sqrt((varz/n-SQR(swz/n))/n);
```

Общия извод при употребата на подбни процедури е: опитайте се да извадите “най-големия” член който е аналитично интегрируем и при това резултата може да се обърне. Критерия за най-добър образ е това което остане от функцията да бъде възможно най-близко до константа. Разбире се ако се стигне до крайноя случай в който вие сте успели да направите интеграла точно константа, то вие по същество се свършили цяляа работа. Тогава защо ви е Монте Карло? Е оказва се, че такива крайни случай са рядкост и по-често ни се отдава да сведем функцията до нещо близко но малко по-голямо от търсената (обикновено многомерна) област. Тази техника се среща в литературата под названието *reduction of varience*.

Главния недостатък на простото Монте Карло интегриране, че точността нараства само като корен квадратен от N, броя на изолзваните случайни точки. Ако изискванията ви за точност не са високи, или пък разолагате с добър компютър, то тази техника е силно препоръчителна, поради своята простота, общност и относителна сигурност. В следващите два параграфа ще дискутираме техники за “преминаване на бариерата корен от N” и постигане, поне в някои случаи по-виска точност с по-малко пресмятания на функцията.

3. Квази-случайни поредици

Току що видяхме, че избирайки N точки равномерно разпределени в n-мерно пространство води до грешка на Монте Карло метода, която намалява като 1/sqrt(N). С най-общи думи, всяка нова точка внася линеен принос в сумата, която по-късно става среда стойност на функцията, и също така дава линеен принос в сумата на квадратите, която ще стане грешката. Излизайки “изпод” квадратния корен стигаме до степента $N^{-1/2}$.

Това, че коренната зависимост изглежда логична, не заначи, че тя е неизбежна. Един прост контрапример е следния: взимаме точките в които оценяваме функцията да лежат върху декартова мрежа и изчисляваме във всяка точка точно по веднъж (без зачение в какъв ред). По този начин нашата Монте Карло схема се превръща в детерминистична квадратурна процедура, чиято сходимост за достатъчно гладки и непрекъснати функции е дори по-добра.

Проблема с подобна мрежа е, че не може предварително да се реши колко гъста да бъде построена. Започвайки веднъж, обаче сме длъжни да извършим всички пресмятания на функцията;

защото в противен случай ще допуснем непростимо груба грешка в оценката на интеграла. Човек може да се запита, дали няма някоя междинна схема, по която да се подбират точки "по случаен начин", при все това те да са разпръснати по някакъв самоизбягващ се начин, избягвайки по този начин проблема с клъстеризацията на равномерно разпределените по линейна конгруентна схема числа.

Подобен въпрос възниква и за задачи различни от Монте Карло интегрирането. Може да ни се прииска да претърсим n-мерното пространство за точка кадето някакво локално свойство се изпълнява (например локален екстремум). Разбира се, за да има някакъв смисул подобна задача, то следва да имаме достатъчна непрекъснатост на съответното свойство, така че съответното условие да се изпълнява в някаква n-мерна околност. Възможно е, обаче да не знаем *a priory*, къде и в каква околност се изпълнява конкретното условие, и следователно да искаме да подбираме точки докато условието се изпълни. Въпросът е дали има по-добър начин да правим това от некорелираните псевдо-случайни числа?

Отговорът на горния въпрос е "Да". Поредици от n-торки, които запълват пространството по-равномерно, отколкото некорелираните случайни числа се наричат "квази-случайни" поредици. Този термин е донякъде безсмислен, тъй-като в квази-случайните числа няма нищо случайно. Всъщност те целенасочено се строят така че да не са случайни, и по точно казано, точките да се избягват максимално една друга.

Принципно прост пример са редиците на *Halton*. За едномерно пространство, поредното j-то число H_j от подобна поредица се получава по следния начин.

- Записваме j като число при основа на бройната система b, където b енякакво прсто число. (Нпример $j = 17$, при основа $b = 3$ е 122.)
- Обръщаме цифрите на числото и слагаме десетична точкана b-тата позиция напред.
- За да се получат поредици от този вид се взимат първите n прости числа.

Не е трудно да се проумее как работи подобен алгоритъм работи. Всеки път броя на цифрите нараства с единица и ние получаваме все по-фина и по фина мрежа. По този начин получаваме точки които максимално се избчгват една друга.

Други начини за генериране на квази-случайни числа са предложени от Faure, Sobol, Nienderreiter и други. Сега накратко ще дискутираме варианта предложан от Антонов и Салеев. Поредиците *Sobol* се генерират числа между нула и единица, директно като двоични дроби с дължина *w* бита, с помощта на специални двоични функции наречени *дирекционни числа*. В оригиналния метод, *j*-тото число се получава, като се налага логическо XOR на съответното дирекционно число и *j*. Когато j расте, всички базисни функции си сменят ролята, като всяка от тях се използва не по-често от веднъж на 2^{k-1} стъпки.

Приносът на Антонов и Салеев е да покажат, че вместо да се използват битовете на цялото число *j*, за да се избират базисните функции, могат да се използват направо битовете на Грей-кодовете на *j* – *G*(*j*). Т. е. Могат да се строят кодовете на Грей с последователно увеличаваща се дължина.

Стигнахме до мястото където следва да кажем няколко думи за това как се генерират базисните функции V_i. Извода се базира на базата на простите двоични полиноми от ред *q*.

$$P = x^q + a_1 x^{q-1} + a_2 x^{q-2} + \ldots + a_{q-1} + 1$$

Поредиците от цели числа M_i се дефинират чрез рекурентното съотношение:

$$M_i = 2a_1 M_{i-1} \oplus 2^2 a_2 M_{i-2} \oplus \cdots$$
$$\oplus 2^{q-1} M_{i-q+1} a_{q-1} \oplus (2^q M_{i-q} \oplus M_{i-q})$$

В горното равенство с $\oplus$ се отбелязва операцията XOR. Началните точки на рецурсивните съотношения могат да са произволни цели числа, които са нечетни и по-малки от съответните степени на 2. Тогава азисните функции се дават от:

$$V_i = M_i / 2^i$$

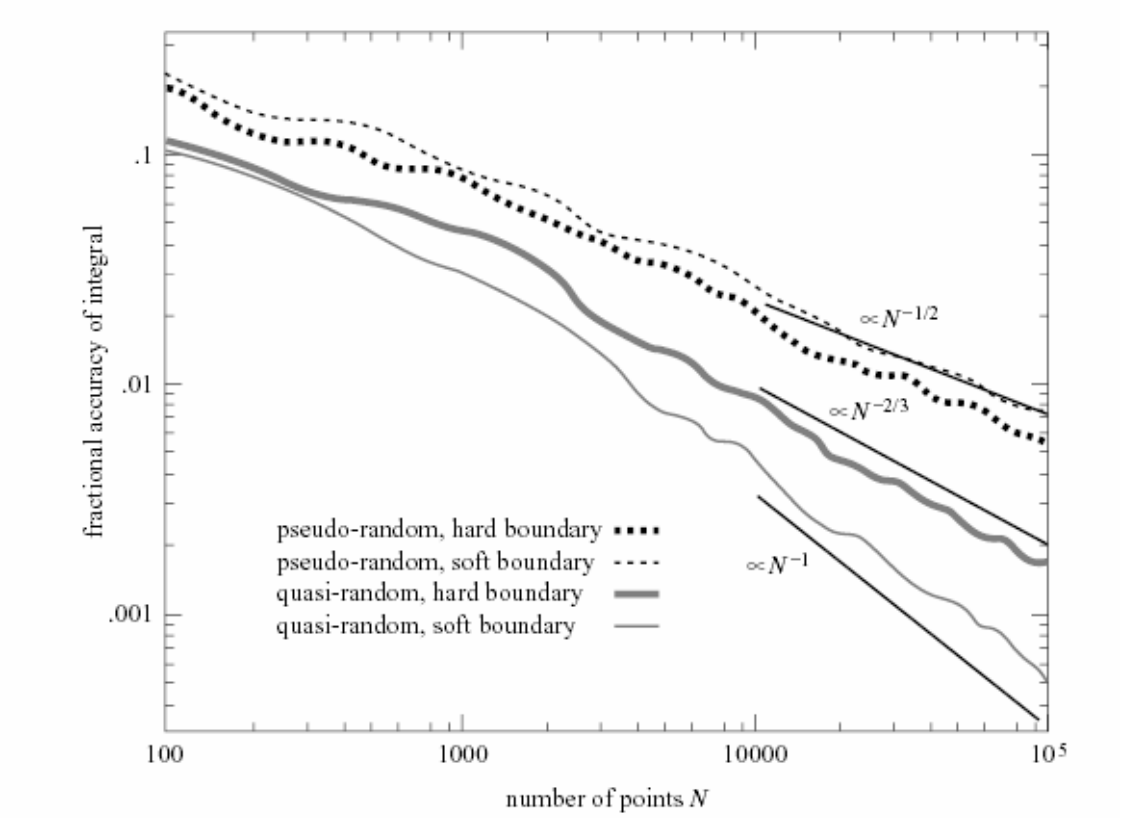
където i принадлежи на 1,…, *w*.

Нека хвърлим светлина върху една примерна процедура загенериране на подобни поредици.

```
#include "nrutil.h"
#define MAXBIT 30
#define MAXDIM 6
void sobseq(int *n, float x[])
{
int j,k,l;
unsigned long i,im,ipp;
```

```
static float fac;
static unsigned long in,ix[MAXDIM+1],*iu[MAXBIT+1];
static unsigned long mdeg[MAXDIM+1]={0,1,2,3,3,4,4};
static unsigned long ip[MAXDIM+1]={0,0,1,1,2,1,4};
static unsigned long iv[MAXDIM*MAXBIT+1]={
0,1,1,1,1,1,1,3,1,3,3,1,1,5,7,7,3,3,5,15,11,5,15,13,9};
if (*n < 0) { Initialize, don't return a vector.
for (k=1;k<=MAXDIM;k++) ix[k]=0;
in=0;
if (iv[1] != 1) return;
fac=1.0/(1L << MAXBIT);
for (j=1,k=0;j<=MAXBIT;j++,k+=MAXDIM) iu[j] = &iv[k];
To allow both 1D and 2D addressing.
for (k=1;k<=MAXDIM;k++) {
for (j=1;j<=mdeg[k];j++) iu[j][k] <<= (MAXBIT-j);
Stored values only require normalization.
for (j=mdeg[k]+1;j<=MAXBIT;j++) { Use the recurrence to get other values.
ipp=ip[k];
i=iu[j-mdeg[k]][k];
i ^= (i >> mdeg[k]);
for (l=mdeg[k]-1;l>=1;l--) {
if (ipp & 1) i ^= iu[j-l][k];
ipp >>= 1;
}
iu[j][k]=i;
}
}
} else { Calculate the next vector in the sequence.
im=in++;
for (j=1;j<=MAXBIT;j++) { Find the rightmost zero bit.
if (!(im & 1)) break;
im >>= 1;
}
if (j > MAXBIT) nrerror("MAXBIT too small in sobseq");
im=(j-1)*MAXDIM;
for (k=1;k<=IMIN(*n,MAXDIM);k++) { XOR the appropriate direction number
into each component of the
vector and convert to a floating
number.
ix[k] ^= iv[im+k];
x[k]=ix[k]*fac;
}
}
}
```

И все пак, колко добра е в същост *Sobol* поредицата? За Монте Карло интегриране на гладка функция в пространство с размерност n, отговорът е, че частичната грешка ще намалява с N, броят на необходимите точки с $(\ln N)^n/N$, или почти толкова бързо като $1/N$.



Нека сега си предсгавим, че искаме да интегрираме в *N*-мерното пространство възможно *най-икономично*, като имаме огганичение от *M* точки. Например бихме могли да поискаме да изтестваме удароустойчивостта на коли от даден модел, като функция на 4 пройзводствени параметъра, имайки на разположение само три коли за трошене. (Идеята не е в това дали горната идея е добра, тя не е, а в това да се направи, каквото е възможно за да извлецхе максималната информация от случая.)

За тази цел се използва техниката на “латинския хиперкуб”. Идеята е да се раздели пространството на съответия брой размерности, и всеки размер да се раздели съответно на *M* части, като по този начин цялото пространство се разделя на M^N на брой клетки. (Размерите на клеткте по рзличните направления могат да са, както едакви, така и различни според вкуса на читателя.) С 3 коли, и 4 параметъра за оценка, например, разделянето е на: 3 x 3 x 3 x 3 = 81 клетки.

По-нататък, избираме M на брой клетки, които да съдържат точките за оценка, по следния начин: По случаен начин изираме една от M^N на брой клетки, за начална. После зачеркваме всички клетки по съответните редове, колони и т. н., оставяйки $(M\text{-}1)_N$ кандидати. По случаен начин изираме нова случайна точка и така докато остане само една точка.

Резултатът от подобна процедура е, че всеки от параметрите е тестван в всяка своя подобласт. Ако системата се командва предимно от 1 свой параметър, то той ще бъде открит по този метод. Ако обаче имаме силно взаимодействие между параметрите, особена полза няма да има. Метода следва да се използва с особено внимание!

4. Адаптивни и рекурсивни Монте Карло процедури

Идеята за употребата на *importance sampling* вече веднъж бе дискутирана в втора точка. Нека сега подходим по малко по-формален начин. Нека предположим, че подинтегралната функция може да се запише като пройзведение от една почти константна функция h, по друга положително дефинитна функция g. Тогава интеграла по многомерния обем V е следния:

$$\int f dV = \int (f/g) g dV = \int h g dV$$

Можем да разгледаме горната операция, като смяна на променливите. По директния подход към проблема е, да се обобщи направо основната теорема, за случая на неравномерно разпределение на точките за пресмятане на функцията. Нека предположим, че точките x_i се избират от обема V, така че

$$\int p dV = 1$$

Ето и обобщената главна теорема на Монте Карло методите, която дава оценка ан интеграла на коя да е функция f, използвайки N точки

$$I \equiv \int f dV = \int \frac{f}{p} p dV \approx \langle \frac{f}{p} \rangle \pm \sqrt{\frac{\langle f^2/p^2 \rangle - \langle f/p \rangle^2}{N}}$$

За горния израз, разбира се, важат същите забележки относно грешките както и в простата теорема на Монте Карло.

Има и още един съвсем различен подход към проблема. Така наречения метод *stratified sampling*. Нека още малко да разширим означенията като с <<*f*>>, ще означаваме точната средна стойност на функцията върху обема V (това е, разбира се нищо друго освен интеграла на функцията

разделен на обема на областта.), а с <*f*> да означим простата Монте Карло оценка на средната стойност, или:

$$\langle\langle f\rangle\rangle \equiv \frac{1}{V}\int f dV \;\; \langle f\rangle \equiv \frac{1}{N}\sum_{N} f(x_i)$$

Нека сега разделим областта на два независими и равни обема и ги означим с *a* и *b*, и да изберем *N*/2 точки във всеки от обемите. Не е трудно да се съорази, че изборът направен по този начин дава грешка, не по-голяма от стандартната. Но при това все още не сме изолзвали възможността да изберем различен брой точки в различните обеми. Това ни позволява да смъкнем още оценката на грешката. По-общия поглед върху проблема е да се разгледаме обема V разделен на повече от две области и да америм оптималното разпределение на "опипващите точки" в обемите. В пространства с много висока размерност, обаче, този подход не е особено полезен, тъй-като води до така наречената K^d екслозия (*K* е броя на сегментите ан които разделяме, а *d* е размерността на пространството). Този факт налага използването на смесени стратегии, пример на които ще приведем по-долу.

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define ALPH 1.5
#define NDMX 50
#define MXDIM 10
#define TINY 1.0e-30
extern long idum; For random number initialization in main.
void vegas(float regn[], int ndim, float (*fxn)(float [], float), int init,
unsigned long ncall, int itmx, int nprn, float *tgral, float *sd,
float *chi2a)
{
float ran2(long *idum);
void rebin(float rc, int nd, float r[], float xin[], float xi[]);
static int i,it,j,k,mds,nd,ndo,ng,npg,ia[MXDIM+1],kg[MXDIM+1];
static float calls,dv2g,dxg,f,f2,f2b,fb,rc,ti,tsi,wgt,xjac,xn,xnd,xo;
static float d[NDMX+1][MXDIM+1],di[NDMX+1][MXDIM+1],dt[MXDIM+1],
dx[MXDIM+1], r[NDMX+1],x[MXDIM+1],xi[MXDIM+1][NDMX+1],xin[NDMX+1];
static double schi,si,swgt;
Best make everything static, allowing restarts.
if (init <= 0) { Normal entry. Enter here on a cold start.
mds=ndo=1; Change to mds=0 to disable strati_ed sampling,
i.e., use importance sampling only. for (j=1;j<=ndim;j++) xi[j][1]=1.0;
}
if (init <= 1) si=swgt=schi=0.0;
Enter here to inherit the grid from a previous call, but not its answers.
if (init <= 2) { Enter here to inherit the previous grid and its
answers. nd=NDMX;
ng=1;
if (mds) { Set up for strati_cation.
ng=(int)pow(ncall/2.0+0.25,1.0/ndim);
mds=1;
```

```
if ((2*ng-NDMX) >= 0) {
mds = -1;
npg=ng/NDMX+1;
nd=ng/npg;
ng=npg*nd;
}
}
for (k=1,i=1;i<=ndim;i++) k *= ng;
npg=IMAX(ncall/k,2);
calls=(float)npg * (float)k;
dxg=1.0/ng;
for (dv2g=1,i=1;i<=ndim;i++) dv2g *= dxg;
dv2g=SQR(calls*dv2g)/npg/npg/(npg-1.0);
xnd=nd;
dxg *= xnd;
xjac=1.0/calls;
for (j=1;j<=ndim;j++) {
dx[j]=regn[j+ndim]-regn[j];
xjac *= dx[j];
}
if (nd != ndo) { Do binning if necessary.
for (i=1;i<=IMAX(nd,ndo);i++) r[i]=1.0;
for (j=1;j<=ndim;j++) rebin(ndo/xnd,nd,r,xin,xi[j]);
ndo=nd;
}
if (nprn >= 0) {
printf("%s: ndim= %3d ncall= %8.0f\n",
" Input parameters for vegas",ndim,calls);
printf("%28s it=%5d itmx=%5d\n"," ",it,itmx);
printf("%28s nprn=%3d ALPH=%5.2f\n"," ",nprn,ALPH);
printf("%28s mds=%3d nd=%4d\n"," ",mds,nd);
for (j=1;j<=ndim;j++) {
printf("%30s xl[%2d]= %11.4g xu[%2d]= %11.4g\n",
" ",j,regn[j],j,regn[j+ndim]);
}
}
}
for (it=1;it<=itmx;it++) {
Main iteration loop. Can enter here (init _ 3) to do an additional itmx iterations with
all other parameters unchanged.
ti=tsi=0.0;
for (j=1;j<=ndim;j++) {
kg[j]=1;
for (i=1;i<=nd;i++) d[i][j]=di[i][j]=0.0;
}
for (;;) {
fb=f2b=0.0;
for (k=1;k<=npg;k++) {
wgt=xjac;
for (j=1;j<=ndim;j++) {
xn=(kg[j]-ran2(&idum))*dxg+1.0;
ia[j]=IMAX(IMIN((int)(xn),NDMX),1);
if (ia[j] > 1) {
xo=xi[j][ia[j]]-xi[j][ia[j]-1];
rc=xi[j][ia[j]-1]+(xn-ia[j])*xo;
} else {
xo=xi[j][ia[j]];
rc=(xn-ia[j])*xo;
}
x[j]=regn[j]+rc*dx[j];
wgt *= xo*xnd;
}
f=wgt*(*fxn)(x,wgt);
f2=f*f;
fb += f;
f2b += f2;
for (j=1;j<=ndim;j++) {
di[ia[j]][j] += f;
if (mds >= 0) d[ia[j]][j] += f2;
```

```
}
}
f2b=sqrt(f2b*npg);
f2b=(f2b-fb)*(f2b+fb);
if (f2b <= 0.0) f2b=TINY;
ti += fb;
tsi += f2b;
if (mds < 0) { Use strati_ed sampling.
for (j=1;j<=ndim;j++) d[ia[j]][j] += f2b;
}
for (k=ndim;k>=1;k--) {
kg[k] %= ng;
if (++kg[k] != 1) break;
}
if (k < 1) break;
}
tsi *= dv2g; Compute _nal results for this iteration.
wgt=1.0/tsi;
si += wgt*ti;
schi += wgt*ti*ti;
swgt += wgt;
*tgral=si/swgt;
*chi2a=(schi-si*(*tgral))/(it-0.9999);
if (*chi2a < 0.0) *chi2a = 0.0;
*sd=sqrt(1.0/swgt);
tsi=sqrt(tsi);
if (nprn >= 0) {
printf("%s %3d : integral = %14.7g +/- %9.2g\n",
" iteration no.",it,ti,tsi);
printf("%s integral =%14.7g+/-%9.2g chi**2/IT n = %9.2g\n",
" all iterations: ",*tgral,*sd,*chi2a);
if (nprn) {
for (j=1;j<=ndim;j++) {
printf(" DATA FOR axis %2d\n",j);
printf("%6s%13s%11s%13s%11s%13s\n",
"X","delta i","X","delta i","X","delta i");
for (i=1+nprn/2;i<=nd;i += nprn+2) {
printf("%8.5f%12.4g%12.5f%12.4g%12.5f%12.4g\n",
xi[j][i],di[i][j],xi[j][i+1],
di[i+1][j],xi[j][i+2],di[i+2][j]);
}
}
}
}
for (j=1;j<=ndim;j++) { Re_ne the grid. Consult references to understand
the subtlety of this procedure. The re_nement
is damped, to avoid rapid, destabilizing
changes, and also compressed in range
by the exponent ALPH.
xo=d[1][j];
xn=d[2][j];
d[1][j]=(xo+xn)/2.0;
dt[j]=d[1][j];
for (i=2;i<nd;i++) {
rc=xo+xn;
xo=xn;
xn=d[i+1][j];
d[i][j] = (rc+xn)/3.0;
dt[j] += d[i][j];
}
d[nd][j]=(xo+xn)/2.0;
dt[j] += d[nd][j];
}
for (j=1;j<=ndim;j++) {
rc=0.0;
for (i=1;i<=nd;i++) {
if (d[i][j] < TINY) d[i][j]=TINY;
r[i]=pow((1.0-d[i][j]/dt[j])/
(log(dt[j])-log(d[i][j])),ALPH);
```

```
rc += r[i];
}
rebin(rc/xnd,nd,r,xin,xi[j]);
}
}
}
void rebin(float rc, int nd, float r[], float xin[], float xi[])
vector r.
{
int i,k=0;
float dr=0.0,xn=0.0,xo=0.0;
for (i=1;i<nd;i++) {
while (rc > dr)
dr += r[++k];
if (k > 1) xo=xi[k-1];
xn=xi[k];
dr -= rc;
xin[i]=xn-(xn-xo)*dr/r[k];
}
for (i=1;i<nd;i++) xi[i]=xin[i];
xi[nd]=1.0;
}
```

5. Заключение

В настоящия реферат засегнахме няколко, коренно различни метода за интегриране по Монте Карло. Като примери бяха включени, няколко програмни фрагмента, нйакои от които далеч не прости. Това, което е общата черта на тези подходи е получаването на незаменими и точни оценки на базата на “случайни” извадки. Ефективността на процедурите, определено не заслужава асоциациите свързани с общото име на методите.

Литература

Hammersley, J.M., and Handscomb, D.C. 1964, Monte Carlo Methods (London: Methuen).

Shreider, Yu. A. (ed.) 1966, The Monte Carlo Method (Oxford: Pergamon).

Sobol’, I.M. 1974, The Monte Carlo Method (Chicago: University of Chicago Press).

Kalos, M.H., and Whitlock, P.A. 1986, Monte Carlo Methods (New York: Wiley).

Halton, J.H. 1960, Numerische Mathematik, vol. 2, pp. 84–90.

Bratley P., and Fox, B.L. 1988, ACM Transactions on Mathematical Software, vol. 14, pp. 88–100.

Lambert, J.P. 1988, in Numerical Mathematics – Singapore 1988, ISNM vol. 86, R.P. Agarwal, Y.M. Chow, and S.J. Wilson, eds. (Basel: Birkha¨ user), pp. 273–284.

Niederreiter, H. 1988, in Numerical Integration III, ISNM vol. 85, H. Brass and G. H¨ammerlin, eds. (Basel: Birkha¨ user), pp. 157–171.

Sobol’, I.M. 1967, USSR Computational Mathematics and Mathematical Physics, vol. 7, no. 4,

pp. 86–112.

Antonov, I.A., and Saleev, V.M 1979, USSR Computational Mathematics and Mathematical Physics, vol. 19, no. 1, pp. 252–256.

Dunn, O.J., and Clark, V.A. 1974, Applied Statistics: Analysis of Variance and Regression(New York, Wiley)

Hammersley, J.M. and Handscomb, D.C. 1964, Monte Carlo Methods(London: Methuen).

Kalos, M.H. and Whitlock, P.A. 1986, Monte Carlo Methods (New York: Wiley).

Bratley, P., Fox, B.L., and Schrage, E.L. 1983, A Guide to Simulation(New York: Springer-Verlag).

Lepage, G.P. 1978, Journal of Computational Physics, vol. 27, pp. 192–203.

Lepage, G.P. 1980, “VEGAS: An Adaptive Multidimensional Integration Program,” Publication CLNS-80/447, Cornell University.

Press, W.H., and Farrar, G.R. 1990, Computers in Physics, vol. 4, pp. 190–195.

Съдържание

1.....Увод 1

2.Елементарно Монте Карло интегриране 1

3.Квази-случайни поредици 3

4.Адаптивни и рекурсивни Монте Карло процедури 8

5.....Заключение 12

Литература 12

Съдържание 14