GPU を使った並列計算の導入による数値計算の効率性向上についての検証 －数値積分を対象とした検証－

福井 昭吾

1　はじめに

GPU(Graphic Processing Unit) は、コンピュータの中でグラフィックに関する処理を担当する演算装置である。通常、コンピュータでの演算は CPU(Central Processing Unit) が担い、GPU はグラフィックの描画に関する処理のみを行う。しかし、近年における GPU の性能向上に伴い、グラフィック以外の処理における GPU の応用が進んでいる。実際に、人工知能・自動運転・金融計算などの様々な分野で GPU を使った計算が行われている。

GPU は、並列処理に特化した機構を持つ。近年の GPU はその中に数千程度のコアを内蔵している。そのため、大量のデータを分割し、GPU の各コアがこれらのデータを同時に処理することで、極めて効率的な計算が可能となる。一方、GPU と比較して、CPU はコア数が少ないものの複雑な処理をより素早く行うことができる。したがって、GPU を使った数値計算は、逐次的な処理を CPU が、並列的な処理を GPU がそれぞれ担当するという形態が一般的である。

本研究では、GPU を使った並列計算を導入することによってどの程度計算時間が短縮できるのかを、数値積分を題材に検証していく。任意の積分可能な関数 $f(x)$ について、その定積分の解析解が得られない場合がある。そのような場合に、その定積分を数値的に求めることを数値積分という。数値積分には多くの手法が存在するが、本研究では Gauss-Legendre 法を用いるものとし、数値計算ライブラリの一つである Math.NET Numerics の実装にしたがって数値積分の計算を行う。

検証には NVIDIA 社の GPU である Tesla K40 を用いる。また、GPU 上で計算を行うためのプログラミング言語として、同社による C/C++ 言語拡張である CUDA C/C++ を使う。CUDA C/C+ に関しては、NVIDIA (2016) による公式のマニュアルが存在する。また、Cheng et al. (2015) は、CUDA C/C++ によるプログラミングについて、最近の状況を踏まえつつ網羅的に説明している。

2　一変数関数の数値積分

積分可能な一変数関数 $f(x)$ の定積分 $\int_l^u f(x)dx$ は、Gauss-Legendre 法で次のように近似する[*1]。

$$\int_l^u f(x)dx \approx \frac{u-l}{2}\sum_{i=0}^{n-1} w_i f(x_i) \tag{1}$$

$$x_i = \frac{u+l}{2} + a_i\frac{u-l}{2}$$

$$w_i = \frac{2}{(1-a_i^2)\{P_n'(a_i)\}} \tag{2}$$

$$(i+1)P_{i+1}(x) = (2i+1)xP_i(x) - iP_{i-1}(x) \tag{3}$$

$$P_0(x) = 1$$

$$P_{-1}(x) = 0$$

ただし、w_i はウエイト、a_i は積分点、n は積分点の数である。また、積分点 a_i は多項式 $P_n(x)$ の根である。数値積分は解析解に対する近似に過ぎない。n を大きくすればその近似の精度を高めることができるが、その反面、計算時間は増加する。

w_i および a_i は積分点の数 n によって決まり、関数 $f(x)$ の形状や区間 $[l,u]$ には依存しない。したがって、所与の積分点数で式 (1) による数値積分を複数回行うならば、初めに n を設定して a_i および w_i を計算しておき、その後、この a_i と w_i を使い数値積分を繰り返すことで、a_i と w_i の計算にかかる時間を減らすことが可能となる。実際に、Math.NET Numerics では、$n = 2,\ldots,20,32,64,96,100,128,256,512,1024$ については、あらかじめ計算した a_i および w_i を用いて数値積分を計算している。

Gauss-Legendre 法に基づく数値計算では、$f(x_i)$ の導出を並列化することでさらに効率的な計算が見込める。つまり、式 (1) において、$f(x_0), f(x_1), \ldots, f(x_{n-1})$ の計算を並列に行うことができれば、これらを逐次的に計算するよりも短い時間で数値積分の結果を求められるだろう。数値積分におけるこのような並列化は極めて一般的である。例えば、数値計算ソフトウエアの一つである MATLAB では、一変数関数の数値積分について上記のような並列計算を行う実装が数種類公開されている。そのため、MATLAB とその実装を組み合わせて使うことで、並列計算を使った数値積分を容易に行うことができる。

一変数関数 $f(x)$ について区間 $I_k = (l_k, u_k)$ $(k = 1,\ldots,N)$ の数値積分を求めたいという状況を考えてみよう。例えば、$f(x)$ を確率密度関数として、点 $x_1,\ldots,x_{N+1}$ を境界とする各区間の相対度数を求めたいときなどに、この種の計算が必要となる。最も単純な解決法は、上記の方法を使って $\int_{I_k} f(x)dx$ の数値積分を区間 I_k ごとに逐次求めることだろう。ただし、GPU を使うならば、この計算は必ずしも効率的ではない。例えば、Gauss-Legendre

[*1] 数値積分の詳細については、例えば Monahan (2001), Press et al. (2002) 等を参照せよ。

法の積分点数を $n = 16$ とする場合、16 個の積分点について $f(x)$ を並列計算して $\int_{I_k} f(x)dx$ の数値積分を求めることを N 回を繰り返す。一般に GPU は数百から数千程度のコアを内蔵しているため、以上の方法だと一度の数値積分でごく一部のコアだけが使われるに過ぎない。

GPU を使って効率的な計算を行うならば、並列に行う計算の数はできるだけ多い方が望ましい。上記の状況では、数値積分に必要な $f(x)$ の値をすべての区間 I_k についてまとめて並列計算し、区間 I_k ごとの加重和を計算し数値積分を行うことで、さらに効率的な計算が可能となる。

3　二重積分の数値積分

次に、二変数関数の二重積分について、GPU による数値積分を考えてみよう。二重積分についても Gauss-Legendre 法による数値計算を考えることにする。Gauss-Legendre 法では、積分区間 $I^X \times I^Y = (l_x, u_x) \times (l_y, u_y)$ における二重積分を

$$\int_{I^X}\int_{I^Y} f(x,y)dydx \approx \frac{u_x - l_x}{2}\frac{u_y - l_y}{2}\sum_{i=0}^{n_x-1}\sum_{j=0}^{n_y-1} w_i w_j f(x_i, y_j) \tag{4}$$

$$x_i = \frac{u_x + l_x}{2} + a_i\frac{u_x - l_x}{2}$$

$$y_j = \frac{u_y + l_y}{2} + a_j\frac{u_y - l_y}{2}$$

と近似する。なお、積分点 (a_i, a_j) とウエイト (w_i, w_j) は、一次関数の定積分と同様に求める。すなわち、積分点の数 n_x に対して式 (3) の根を積分点 a_i とし、式 (2) からウエイト w_i を計算する。(a_j, w_j) についても同様に求める。

一変数関数の定積分と同様、二重積分の場合も積分点数 (n_x, n_y) の選択が数値積分の近似の精度とその計算時間を決める。ただし、二重積分における積分点数の増加に対する計算時間の上昇は、一変数関数の定積分の場合よりも大きい。例えば、一変数関数の定積分では、変数 x の積分点数が 1 増えたとき $f(x)$ の計算が 1 回増えるに過ぎない。しかし、二重積分で変数 x の積分点数 n_x が 1 増えた場合、$f(x,y)$ の計算が n_y 回増えることになる。

その反面、二重積分の数値積分では、GPU による並列計算の恩恵は大きい。二重積分の場合、一回の数値積分につき、$f(x,y)$ の評価が $n_x \times n_y$ 回必要である。GPU のコア数が大きいほど、これらの評価をより少ないサイクルでまとめて計算できることになる。その結果、GPU による並列化を二重積分に導入することで、そうでない場合よりも短い時間で二重積分の計算が可能となる。

並列化の導入以外にも、二重積分の数値積分を効率化する方法がある。例えば、ある関数 $f(x,y)$ について二重積分を複数回行う場合、各積分点における $f(x,y)$ の値をメモリに保存しておくことでさらなる計算時間の短縮化が見込める。特に、$f(x,y)$ の計算に時間がかかるほど、その効果は大きい。例えば、積分区間 $I^X \times I^Y = (l_x, u_x) \times (l_y, u_y)$ について、

表 1　GMM 推定の計算にかかった時間

	CPU のみ	CPU + GPU による並列計算
GMM 推定のみ	43 分 31 秒	19.5 秒
全体	44 分 7 秒	20.1 秒

$\int_{I^X}\int_{I^Y} g(x,y)f(x,y)dydx$ と $\int_{I^X}\int_{I^Y} h(x,y)f(x,y)dydx$ という二つの二重積分を求める場合、初めに各積分点における $f(x,y)$ を計算し、その後で式 (4) から

$$\int_{I^X}\int_{I^Y} g(x,y)f(x,y)dydx \approx \frac{u_x-l_x}{2}\frac{u_y-l_y}{2}\sum_{i=0}^{n_x-1}\sum_{j=0}^{n_y-1} w_i w_j g(x_i,y_j)f(x_i,y_j)$$

$$\int_{I^X}\int_{I^Y} h(x,y)f(x,y)dydx \approx \frac{u_x-l_x}{2}\frac{u_y-l_y}{2}\sum_{i=0}^{n_x-1}\sum_{j=0}^{n_y-1} w_i w_j h(x_i,y_j)f(x_i,y_j)$$

としてそれぞれの数値積分を求めれば良い。このとき、$f(x,y)$ は最初に計算した値をメモリから取り出せば良く、改めて $f(x,y)$ を計算する必要はない。この方法は、並列計算を行わない場合にも有効である。

以上より、本研究では、二重積分 $\int_{I^X}\int_{I^Y} f(x,y)dydx$ の計算を次の手順で行う。(1) 積分点数 (n_x,n_y) を設定し、積分点 (a_i,a_j) とウエイト (w_i,w_j) を事前に求めておく、(2) 積分点ごとに $f(x,y)$ の値を並列的に求め、GPU のメモリ上に記録する、(3) 二重積分を計算する都度、(2) で記録した $f(x,y)$ の値を読み込み、式 (4) に基づいて数値積分を求める。

4　GMM 推定での実装例

以下では、筆者が過去に行った GMM 推定を題材として、GPU による並列計算の導入がどの程度の効率化をもたらすのかを検証する。GMM 推定は、所与のモーメント式から構築した目的関数について、その値を最小にするパラメータを推定値とする。福井 (2015) では、所得および消費の度数分布表（階層別データ）から、所得と消費の同時密度関数のパラメータを GMM 推定する方法を示した。付録 A では、その推定方法を簡潔に説明している。この GMM 推定では、目的関数を 1 回計算するごとに、一変数関数の数値積分と二重積分を複数回計算する。したがって、パラメータの推定値を求めるまでに、多くの回数の数値積分を行わなくてはならない。福井 (2015) では推定に必要なこれらの計算を CPU のみで行っていたため、推定結果を得るまでに多大な時間が必要だった[*2]。そこで、これらの数値積分に対して GPU による並列計算を導入することで、GMM 推定の計算時間がどれほど短縮されるかを見てみよう。

表 1 はその計測結果である。表 1 において、「GMM 推定のみ」は GMM 推定における数

[*2] 実際には、二重積分の数値積分において、積分点ごとの $f(x,y)$ を計算する際にのみ CPU による並列計算を導入していた。

表 2　使用するコンピュータの性能

CPU	Intel Core i7-5960X
GPU	NVIDIA Tesla K40
チップセット	Intel X99 Chipset
メモリ	DDR4 16GB

値最適化部分のみを計測した時間であり、「全体」はファイルからのデータ読み込み・推定結果に基づく各種代表値の計算・ファイルへの推定結果の書き込みなどの、GMM 推定以外の処理も含めた計測時間である。

表 1 から明らかなように、GPU による並列計算を導入することで、計算時間の大幅な短縮を実現している。実際には、後述の通り、CPU のみの計算と GPU を導入した計算とでは前者の積分点数が大きいため、それが計算時間の違いに影響を与えているものの、おそらくその影響は小さい。したがって、積分点の違いを差し引いても計算時間は大きく短縮したといえる。

計算時の環境・設定等について説明しておこう。今回の計算で用いたコンピュータの性能は表 2 に示している。また、使用したプログラミング言語は、CUDA C/C++, C++/CLI, F#の 3 種類であり、プログラムの作成には Visual Studio 2015 を用いた。福井 (2015) の計算では F#を使っており、このプログラムの一部を今回の計算に用いるために以上の構成を採用した。具体的には、GPU による計算・CPU–GPU 間のデータ転送といったアンマネージドな部分を CUDA C/C++ で、アンマネージドな部分とマネージドな部分とを橋渡しする部分を C++/CLI で、マネージドな部分を F#で、それぞれ記述した。以上の構成は $\epsilon\pi\iota\sigma\tau\eta\mu\eta$ (2015) を参考にしている。

GMM 推定における数値最適化に関しては、福井 (2015) と同じ設定を用いている。実際の数値最適化では、局所解への収束・境界外への推定値の移動を防ぐため、Nelder-Mead 法を 30 回繰り返した後 BFGS 法を適用した。

同時密度の二重積分の計算に関して、福井 (2015) の計算ではシンプソン法を使ったが、本研究における GPU を導入した計算では先述の Gauss-Legendre 法を使うよう変更した。その際、GPU による計算効率を向上するため、積分点数についても修正を行っている。二重積分を求める際、シンプソン法を用いたときは積分点数を 1200×1200 としたが、本研究の Gauss-Legenre 法では積分点数を 1216×768 と設定している。そのため、二重積分における $f(x,y)$ の評価回数は、前者の方が多い。

この GMM 推定のプログラムについてその一部を示すが、その前に、NVIDIA 社製 GPU の構造と CUDA C/C++ におけるプログラミングモデルを簡潔に説明しておく。NVIDIA 社の GPU は、所定の数のコアを SM(Streaming Multiprocesser) という機構に内包し、複数の SM をまとめて GPU を構成するという仕組みを採用している。Tesla K40 の場合、一つの SM は 192 個の単精度コア（単精度小数点数値計算用のコア）と 64 個の倍精度コア（倍精

度小数点数値計算用のコア）を内包し、さらに15個のSMでGPUを構成している。CUDA C/C++では、GPU上のスレッドの動作を関数で記述する。この関数をカーネル（関数）という。スレッドは一定数ごとにブロックという集まりにまとめられ、ブロックの中のスレッドが協調して並列に動作する。さらにこれらブロック全体をグリッドという。CUDA C/C++上では、スレッドの動作をカーネルに記述し、ブロックとグリッドの大きさを設定してCPU側からカーネルを実行する。グリッド内の各ブロックはGPU上のいずれかのSMに割り当てられ、ブロック内の各スレッドは、SM内のコアによりカーネルの記述に従って動作するのである。実際には、ブロック内のスレッドがすべて並列に計算されるのではなく、スレッドを32個ずつに分割して並列に実行する。このスレッドの集まりをワープという。

ソースコード1　第2種の一般化ベータ分布に関わるプログラム（一部抜粋）

```
// ワープ内の前半 16個のスレッドと後半 16個のスレッドについて、スレッドの和を計算する 。
__inline__ __device__ double halfWarpReduce(double sum)
{
  sum += __shfl_xor(sum, 8);
  sum += __shfl_xor(sum, 4);
  sum += __shfl_xor(sum, 2);
  sum += __shfl_xor(sum, 1);
  return sum;
}

// Gauss-Legendre 法のウエイト、積分点、積分区間の幅
struct QuadratureInfoOnDevice
{
  double* Weights;        // ウエイト
  double* RealAbscissas;  // 積分点
  double* Widths;         // 積分区間の幅を 2で割った値
};

// 密度関数の計算に関わるデータ
struct DensityInfoOnDevice
{
  double* GlobalGrid;             // 度数分布表の区切り
  double* GlobalLowerBounds;      // 度数分布表の下限
  double* GlobalUpperBounds;      // 度数分布表の上限
  double* LocalGrids;             // 各階層内に設定した格子点
                                  //(同時密度計算時の積分点)
  double* PDFValues;              // 上記格子点における密度関数の値
  double* PartialProbabilities;   // 上記格子点間の相対度数
  double* ClassProbabilities;     // 各階層の相対度数
  double* CDFValues;              // 各階層の累積相対度数
  double* Percentiles;            // 各階層の上限におけるパーセント点

  double* ValuesAtAbscissas;      // LocalGrids の各格子間に設定した積分点

  unsigned int NumberOfClasses;   // 度数分布表の階層数
  unsigned int NumberOfLocalIntegration;
                                  // 累積相対度数等の計算で必要となる積分の数
                                  // (= NumberOfClasses * LocalGridSize)

  int GlobalGridLength;           // GlobalGrid の要素数
  int LocalGridsLength;           // LocalGrids の要素数
  unsigned int LocalGridSize;     // LocalGrids の各格子間に設定する積分点の数
};

```

```
45 // LocalGrids 間の相対度数の計算
46 __global__ void kernel_GB2PartialProbabilities(DensityInfoOnDevice* densityInfo,
47   QuadratureInfoOnDevice* quadratureInfo, unsigned int totalGridSize)
48 {
49   // 各種インデックスの計算
50   unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
51   unsigned int gridIdx = threadIdx.x % constantsUInt[0];
52   unsigned int quadIdx = i / constantsUInt[0];
53
54   // 密度関数とウエイトの積を計算
55   if (i < totalGridSize)
56   {
57     double temp =
58       1.0 + pow((quadratureInfo->RealAbscissas[i] / GB2Parameters[1]), GB2Parameters[0]);
59     temp = (GB2Parameters[0] * GB2Parameters[2]) * log(GB2Parameters[1]) +
60       GB2ConstantsDouble[0] + (GB2Parameters[2] + GB2Parameters[3]) * log(temp);
61     temp = log(GB2Parameters[0]) + (GB2Parameters[0] * GB2Parameters[2] - 1.0) *
62       log(quadratureInfo->RealAbscissas[i]) - temp;
63     densityInfo->ValuesAtAbscissas[i] = exp(temp) * quadratureInfo->Weights[gridIdx];
64   }
65
66   __syncthreads();
67
68   // 密度関数とウエイトの積について、和を計算。
69   double tempSum = halfWarpReduce(densityInfo->ValuesAtAbscissas[i]);
70
71   // 数値積分の結果をPartialProbabilities に代入
72   if (i % constantsUInt[0] == 0)
73   {
74     densityInfo->PartialProbabilities[quadIdx] =
75       tempSum * quadratureInfo->Widths[quadIdx];
76   }
77 }
```

今回の推定では、同時密度を計算する前に、所得の密度関数と年齢の密度関数を設定し所与の区間に対して相対度数を計算する必要がある。ソースコード 1 は、所得の密度関数として第 2 種の一般化ベータ分布を仮定し、相対度数の計算部分を CUDA C/C++ で記述したものである[*3]。密度関数 $f(x)$ に基づき区間 $\tilde{I}_i^X$ の相対度数を計算する場合、一次関数 $f(x)$ の定積分

$$\int_{\tilde{I}_i^X} f(x)dx$$

について数値積分を行う必要がある。ソースコード 1 はその計算を行った部分のみを抜粋している。

計算に先立って、`QuadratureInfoOnDevice` 型と `DensityInfoOnDevice` 型の変数を作り、計算に必要なデータをそれらの変数に代入しておく。実際には、必要となるデータをホスト側（CPU）で作成してデバイス（GPU）上のこれらの変数にコピーする。`QuadratureInfoOnDevice` は、Gauss-Legendre 法による数値積分に必要なデータを保持する。`Weights` と `RealAbscissas` については、事前に計算されたウエイトおよび積分点の値に基づく。`Widths` は、後述する `LocalGrids` において隣接する値間の幅を 2 で割った値であ

[*3] 第 2 種の一般化ベータ分布については、McDonald (1984) および McDonald and Xu (1995) を参照。

る。DensityInfoOnDevice は、所得分布と年齢分布の双方についてその計算に必要なデータを記録・保持する。まず、推定の元となる度数分布表の区切りをメンバ変数 GlobalGrid に記録する。次に、同時密度の計算で行う数値積分に備えて、GlobalGrid の各区切りの間に積分点を作成し、その積分点すべてをメンバ変数 LocalGrids に保持する。なお、積分点数は、所得分布については $n_x = 64$, 年齢分布については $n_y = 64$ としている。さらに、今回の同時密度の計算では LocalGrids ごとの累積相対度数が必要となるため、LocalGrids の各積分点間にも積分点を作成し、そのすべてをメンバ変数 ValuesAtAbscissas に記録する。この積分点数を 16 とし、変数 constantsUInt[0] にその値を保存している。今回のデータでは、所得の度数分布表の階層数は 19 であるため, その GlobalGrid, LocalGrids, ValuesAtAbscissas の大きさはそれぞれ $(19, 19 \times 64, 19 \times 64 \times 16)$ となり、年齢の度数分布表の階層数は 12 であるから、GlobalGrid, LocalGrids, ValuesAtAbscissas の大きさは $(12, 12 \times 64, 12 \times 64 \times 16)$ となる。以上で準備した各種データに基づき、PDFValues, PartialProbabilities, ClassProbabilities, CDFValues, および、Percentiles の値を GPU を使って計算する。

カーネル関数 kernel_GB2PartialProbabilities は、以上のデータに基づいて LocalGrids 間の相対度数を計算している。カーネル関数を定義する場合、関数に接頭語 __global__を付与する。カーネル関数内で、blockIdx.x はブロックの番号を、blockDim.x はブロックサイズ（ブロックに含まれるスレッドの数）を、threadIdx.x はスレッドの番号を示している[*4]。変数 i は、各スレッドと ValuesAtAbscissas 内のデータとを関連づけるインデックスである。例えば、ブロックサイズが 16 であるとき、5 番目のブロックに含まれる 3 番目のスレッドは、ValuesAtAbscissas 内の $16 \times 5 + 3 = 83$ 番目のデータを扱うことになる。また、グリッドとブロックは二次元・三次元的にまとめて表現することもできる。二次元的に表す場合、(blockIdx.x, blockIdx.y) は x 軸方向・y 軸方向のブロック番号を、(blockDim.x, blockDim.y) は x 軸方向・y 軸方向のブロックサイズを、(threadIdx.x, threadIdx.y) は x 軸方向・y 軸方向のスレッド番号を、それぞれ表す。

各スレッドは、ValuesAtAbscissas の各点ごとに密度関数の値を計算し、Gauss-Legendre 法により LocalGrids の各点間の相対度数を求めている。その結果は、PartialProbabilities に記録される。具体的には、$w_i f(x_i)$ に相当する値を各スレッドが計算した後、ワープ内の 16 個のスレッドごとにその和を計算し、最後に LocalGrids の各点間の幅を 2 で割った値 ($(u-l)/2$ に相当) を掛けて数値積分を行い、相対度数を求めた。なお、GB2Parameters[0] から GB2Parameters[3] には、それぞれ第 2 種の一般化ベータ分布のパラメータ (α, β, p, q) の値が事前に代入されている。また、GB2ConstantsDouble[0] には密度関数の計算で頻繁に使う値

$$\log\left(\frac{\Gamma(p)\Gamma(q)}{\Gamma(p+q)}\right)$$

[*4] ブロックの番号およびスレッドの番号は 0 から始まる。

をあらかじめ代入している[*5]。

LocalGrids 間の相対度数 (PartialProbabilities) から、度数分布表の相対度数 (ClassProbabilities), 累積相対度数 (CDFValues), および、標準正規分布におけるパーセント点 (Percentiles) の計算が可能となる。PartialProbabilities に含まれる相対度数について、$n_x = 64$ 個ごとの和を求めることで ClassProbabilities が計算できる。さらに、その累積和が CDFValues であり、CDFValues の各値について標準正規分布のパーセント点を計算することで Percentiles が求められる。

以上の計算を年齢についても同様に行う。ただし、年齢の密度関数については、カーネル関数を正規カーネルとするノンパラメトリック推定を適用するため、密度関数の計算方法は上記と異なる[*6]。

ソースコード 2　同時密度の計算に関わるプログラム（一部抜粋）

```
// 同時密度の計算
__global__ void kernel_GaussianCopulaDensity(double* jointDensities, double rho,
  int rowLength, int columnLength, int shiftRow, int shiftColumn,
  const DensityInfoOnDevice* incomeDensityInfo, const DensityInfoOnDevice* ageDensityInfo)
{
  // 各種インデックスの計算
  int column = blockDim.x * blockIdx.x + threadIdx.x + shiftColumn;
  int row = blockDim.y * blockIdx.y + threadIdx.y + shiftRow;
  int index1D = row * columnLength + column;

  if (row >= rowLength + shiftRow || column >= columnLength + shiftColumn)
  {
    return;
  }

  // 積分点における同時密度の計算
  jointDensities[index1D] = 0.0;
  if (incomeDensityInfo->LocalGrids[column] > nonZeroBoundOfGridValue &&
      ageDensityInfo->LocalGrids[row] > nonZeroBoundOfGridValue &&
      incomeDensityInfo->CDFValues[column] > 0.0 &&
      incomeDensityInfo->CDFValues[column] < 1.0 &&
      ageDensityInfo->CDFValues[row] > 0.0 &&
      ageDensityInfo->CDFValues[row] < 1.0)
  {
    jointDensities[index1D] =
      standardNormalCopula(rho, incomeDensityInfo->Percentiles[column],
        ageDensityInfo->Percentiles[row]) *
      incomeDensityInfo->PDFValues[column] * ageDensityInfo->PDFValues[row];
  }
}

// 同時密度の値に基づく、各種モーメントの計算
template<typename F>
__global__ void kernel_EstimateMomentTemporal(DensityInfoOnDevice* incomeDensityInfo,
  DensityInfoOnDevice* ageDensityInfo, const double* GLWeights,
  const double* jointDensityValues, double* resultTemp,
  const double* incomeProbabilities, const double* ageProbabilities,
  const double* incomeClassMeans, const double* ageClassMeans,
  const double* conditionalIncomeMeans, const double* conditionalAgeMeans,
  unsigned int incomeBlocksPerRange, unsigned int ageBlocksPerRange, F integrand)
```

[*5] 本研究の計算では、GB2Parameters, GB2ConstantsDouble を GPU 内のコンスタントメモリに保持している。

[*6] ノンパラメトリック推定による密度関数の推定に関しては、竹澤 (2007) 等を参照せよ。

```
{
  // シェアードメモリの確保
  extern __shared__ double partialDensityValues[];

  // 各種インデックスの計算
  int idxX = blockIdx.x * blockDim.x + threadIdx.x;
  int idxY = blockIdx.y * blockDim.y + threadIdx.y;
  if (idxX >= incomeDensityInfo->LocalGridsLength ||
    idxY >= ageDensityInfo->LocalGridsLength)
  {
    return;
  }
  int idxSMem = threadIdx.y * blockDim.x + threadIdx.x;
  int globalIdxX = idxX / (incomeDensityInfo->LocalGridSize);
  int globalIdxY = idxY / (ageDensityInfo->LocalGridSize);

  int blockColumnInIntegration = blockIdx.x % incomeBlocksPerRange;
  int blockRowInIntegration = blockIdx.y % ageBlocksPerRange;

  int integrationColumnIdx = blockIdx.x / incomeBlocksPerRange;
  int integrationRowIdx = blockIdx.y / ageBlocksPerRange;

  // 密度関数、被積分関数、および、ウエイトの積を計算
  partialDensityValues[idxSMem] =
    integrand(idxX, idxY, globalIdxX, globalIdxY, incomeProbabilities, ageProbabilities,
      incomeClassMeans, ageClassMeans, conditionalIncomeMeans, conditionalAgeMeans) *
    jointDensityValues[idxY * (incomeDensityInfo->LocalGridsLength) + idxX] *
    GLWeights[blockColumnInIntegration * blockDim.x + threadIdx.x] *
    GLWeights[blockRowInIntegration * blockDim.y + threadIdx.y];

  // 上記の値について和を求める（インターリーブペア方式）
  if (blockDim.x * blockDim.y >= 1024 && idxSMem < 512)
  {
    partialDensityValues[idxSMem] += partialDensityValues[idxSMem + 512];
  }
  __syncthreads();
  if (blockDim.x * blockDim.y >= 512 && idxSMem < 256)
  {
    partialDensityValues[idxSMem] += partialDensityValues[idxSMem + 256];
  }
  __syncthreads();
  if (blockDim.x * blockDim.y >= 256 && idxSMem < 128)
  {
    partialDensityValues[idxSMem] += partialDensityValues[idxSMem + 128];
  }
  __syncthreads();
  if (blockDim.x * blockDim.y >= 128 && idxSMem < 64)
  {
    partialDensityValues[idxSMem] += partialDensityValues[idxSMem + 64];
  }
  __syncthreads();
  if (blockDim.x * blockDim.y >= 64 && idxSMem < 32)
  {
    partialDensityValues[idxSMem] += partialDensityValues[idxSMem + 32];
  }
  __syncthreads();

  // 和を求める対象が1ワープになったとき、その和をワープシャッフル命令で求める。
  double temp = (idxSMem < 32) ? partialDensityValues[idxSMem] : 0.0;
  if (idxSMem < 32)
  {
    temp += __shfl_xor(temp, 16);
    temp += __shfl_xor(temp, 8);
```

```
104     temp += __shfl_xor(temp, 4);
105     temp += __shfl_xor(temp, 2);
106     temp += __shfl_xor(temp, 1);
107
108     temp *=
109       0.25 * (incomeDensityInfo->GlobalUpperBounds[integrationColumnIdx] -
110         incomeDensityInfo->GlobalLowerBounds[integrationColumnIdx]) *
111       (ageDensityInfo->GlobalUpperBounds[integrationRowIdx] -
112         ageDensityInfo->GlobalLowerBounds[integrationRowIdx]);
113   }
114
115   // ブロックごとに計算した上記の和を、一時的なメモリ(resultTemp)に代入
116   if (idxSMem == 0)
117   {
118     resultTemp[(integrationRowIdx *
119       (incomeDensityInfo->LocalGridsLength / incomeDensityInfo->LocalGridSize) +
120       integrationColumnIdx) * (incomeBlocksPerRange * ageBlocksPerRange) +
121       (blockRowInIntegration * incomeBlocksPerRange + blockColumnInIntegration)] = temp;
122   }
123 }
124
125 // kernel_EstimateMomentTemporal で計算した一時データ(resultTemp)から、
126 // 最終的なモーメントの値を求める。
127 __global__ void kernel_ResultTempSum(const double* resultTemp, unsigned int resultLength,
128   unsigned int incomeBlocksPerRange, unsigned int ageBlocksPerRange, double* result)
129 {
130   // 各種インデックスの計算
131   unsigned int blocksPerRange = incomeBlocksPerRange * ageBlocksPerRange;
132   unsigned int idx = blockDim.x * blockIdx.x + threadIdx.x;
133   double currentSum = resultTemp[idx];
134
135   __syncthreads();
136
137   // 一時的なデータについて、incomeBlocksPerRange と ageBlocksPerRange の積ごとに
138   // 和を求める。
139   if (blocksPerRange == 32U)
140   {
141     currentSum += __shfl_xor(currentSum, 16);
142   }
143   if (blocksPerRange >= 16U)
144   {
145     currentSum += __shfl_xor(currentSum, 8);
146   }
147   if (blocksPerRange >= 8U)
148   {
149     currentSum += __shfl_xor(currentSum, 4);
150   }
151   if (blocksPerRange >= 4U)
152   {
153     currentSum += __shfl_xor(currentSum, 2);
154   }
155   if (blocksPerRange >= 2U)
156   {
157     currentSum += __shfl_xor(currentSum, 1);
158   }
159
160   // 結果をresult に代入
161   if(idx % blocksPerRange == 0U && idx < resultLength * blocksPerRange)
162   {
163     result[idx / blocksPerRange] = currentSum;
164   }
165 }
```

ソースコード2は、同時密度の計算とそれに基づくモーメント式の共分散行列の計算について、一部を抜き出したものである。

カーネル関数 kernel_GaussianCopulaDensity は、所得分布に関するデータ incomeDensityInfo と年齢分布に関するデータ ageDensityInfo に基づいて積分点上の同時密度を計算し、その結果を jointDensities に保存する[*7]。この密度関数の値を記録しておき、GMM 推定に必要な各種一次モーメント・モーメント式の共分散行列を計算する際に用いるのである。

カーネル関数 kernel_EstimateMomentTemporal とカーネル関数 kernel_ResultTempSum は、上で計算した同時密度に基づいてモーメントを計算する。kernel_EstimateMomentTemporal はテンプレート関数であり、引数 integrand に被積分関数を指定することで、各種モーメントを計算することができる[*8]。実際には、kernel_EstimateMomentTemporal は $I^X \times I^Y$ をさらに分割し、分割した各部分について GPU 内の各ブロックが数値積分を計算する。関数 kernel_ResultTempSum は、kernel_EstimateMomentTemporal の結果である数値積分の値を再結合するものである[*9]。

いま、所得の度数分布表における第 i 階層の区間を I_i^X, 年齢の度数分布表における第 j 階層の区間を I_j^Y とすると、関数 kernel_EstimateMomentTemporal と関数 kernel_ResultTempSum は、区間 $I_i^X \times I_j^Y$ ごとに二重積分の数値積分を計算する。そうすることで、様々な区間について数値積分を行うことができる。例えば、区間 $[0,\infty) \times I_j^Y$ についての二重積分

$$\int_0^\infty \int_{I_j^Y} g(x,y)f(x,y)dydx$$

は、

$$\int_0^\infty \int_{I_j^Y} g(x,y)f(x,y)dydx = \sum_i \int_{I_i^X} \int_{I_j^Y} g(x,y)f(x,y)dydx$$

として計算することができる。

表1が示すように、GPU による並列計算の導入は計算時間の大幅な減少をももたらしうることが分かった。このような効率化を実現するためには、いくつかの留意点がある。第一に、より多くの計算を並列に行うことができるほど、並列化の恩恵が大きくなる。言い換えれば、計算の並列化が難しい局面では、並列化による効率向上はそれほど見込めないということになる。したがって、GPU を導入する前に、計算内容について十分な並列化が可能であるかどうかを見極める必要がある。第二に、CUDA C/C++ によるプログラムを効率的に行うためには、NVIDIA 製 GPU による処理の仕組みを正しく理解しなくてはならない。一般的なプログラミング言語と CUDA C/C++ とで、プログラムを記述する際に考慮すべき点は必ずしも同一ではない。例えば、ホストとデバイス間でのメモリ転送・ス

[*7] 同時密度関数は、所得と年齢の密度関数をコピュラにより結合したものを仮定する。なお、コピュラについては、Joe (1997), Trivedi and Zimmer (2005), Nelsen (2006), および、Jondeau et al. (2007) 等を参照せよ。

[*8] ここでの被積分関数は、$\int_{I^X}\int_{I^Y} g(x,y)f(x,y)dydx$ を $\int_{I^X}\int_{I^Y} g(x,y)dP(x,y)$ と表したときの $g(x,y)$ をいう。

[*9] アトミック関数を用いることにより、この分割と再結合を一つのカーネル関数内で実現することも可能である。

レッド間の同期・グリッドおよびブロックサイズの選択といった事柄は、CUDA C/C++ によるプログラミング特有の重要な概念である。さらに、ループ展開やメモリパディングなどのより高度な手法を導入することで、計算時間をさらに短縮できるだろう。Cheng et al. (2015) は、CUDA C/C++ による効果的な開発に必要な概念や技術について、網羅的にかつ詳しく説明している。

謝辞

本研究は JSPS 科研費 JP16K21558 の助成を受けたものである。

付録 A　所得と年齢の同時分布の GMM 推定

この付録では、福井 (2015) で示した所得と年齢の同時分布の GMM 推定について、その推定方法と同時分布のモデル化について簡潔に説明する。詳細な説明については、当該論文を参照されたい。

いま、所得と年齢のそれぞれについての度数分布表（階層別データ）があるとする。すなわち、所得に関して、階層 $I_i^X (i = 1, \ldots, m)$ の上限と下限 $(L_i^X, U_i^X)(i = 1, \ldots, m)$, 相対度数 $R_i^X (i = 1, \ldots, m)$, 階層平均 $\bar{X}_i (i = 1, \ldots, m)$, および平均年齢 $\tilde{Y}_i (i = 1, \ldots, m)$ が与えられており、年齢に関しては、階層 $I_j^Y (j = 1, \ldots, n)$ の上限と下限 $(L_j^Y, U_j^Y)(j = 1, \ldots, n)$, 相対度数 $R_j^Y (j = 1, \ldots, n)$, および平均所得 $\tilde{X}_j (j = 1, \ldots, n)$ が与えられているとする。

GMM 推定では、データより求めるモーメントとモデルから計算するモーメントとの差が最小になるようなモデルパラメータを、その推定値とする。これらモーメント間の差を表す式をモーメント式という。各所得階層の相対度数、各所得階層のの階層平均、各年齢階層の平均所得、および、各所得階層の平均年齢について、それぞれのモーメント式の確率測度と積分の範囲が同一になるように、

$$
\mathrm{E}[\boldsymbol{m}] = \mathrm{E}\begin{pmatrix}
1_1(x) - p_1 \\
\vdots \\
1_m(x) - p_m \\
(x - \mu_1)\left\{1_1(x)/P(x \in I_1^X)\right\} \\
\vdots \\
(x - \mu_m)\left\{1_m(x)/P(x \in I_m^X)\right\} \\
(x - \phi_1)\left\{1_1(y)/P(y \in I_1^Y)\right\} \\
\vdots \\
(x - \phi_n)\left\{1_n(y)/P(y \in I_n^Y)\right\} \\
(y - \psi_1)\left\{1_1(x)/P(x \in I_1^X)\right\} \\
\vdots \\
(y - \psi_m)\left\{1_m(x)/P(x \in I_m^X)\right\}
\end{pmatrix}
$$

とまとめることができる。ただし、p_i は第 i 所得階層の相対度数の母数、μ_i は第 i 所得階

層の階層平均、ϕ_j は第 j 年齢階層の平均所得、ψ_i は第 i 所得階層の平均年齢である。また、$P(x \in I_i^X)$ は所得が階層 I_i^X に入る確率、$P(y \in I_j^Y)$ は年齢が階層 I_j^Y に入る確率であり、

$$1_i(x) = \begin{cases} 1 & x \in I_i^X \\ 0 & \text{otherwise} \end{cases}$$
$$1_j(y) = \begin{cases} 1 & y \in I_j^Y \\ 0 & \text{otherwise} \end{cases}$$

である。

このモーメント式に基づき、所得と消費の同時分布 $P(x,y;\boldsymbol{\theta})$ についてGMM推定を行う。実際には、これらのモーメント式を標本で計算した標本対応 $\bar{\boldsymbol{m}}$ を求め、さらに、モーメント式の分散共分散行列 $\boldsymbol{\Sigma}$ を計算し、

$$\min_{\boldsymbol{\theta}} \ \bar{\boldsymbol{m}}' \boldsymbol{\Sigma}^{-1} \bar{\boldsymbol{m}} \tag{5}$$

を満たすパラメータ $\boldsymbol{\theta}$ を推定値とするのである。ここで、モーメント式の標本対応 $\bar{\boldsymbol{m}}$ は、

$$\bar{\boldsymbol{m}} = \begin{pmatrix} R_1^X - \int_{I_1^X \times [0,\infty)} dP(x,y) \\ \vdots \\ R_m^X - \int_{I_m^X \times [0,\infty)} dP(x,y) \\ \bar{X}_1 - \left\{ \int_{I_1^X \times [0,\infty)} x dP(x,y) \right\} / P(x \in I_1^X) \\ \vdots \\ \bar{X}_m - \left\{ \int_{I_m^X \times [0,\infty)} x dP(x,y) \right\} / P(x \in I_m^X) \\ \tilde{X}_1 - \left\{ \int_{[0,\infty) \times I_1^Y} x dP(x,y) \right\} / P(y \in I_1^Y) \\ \vdots \\ \tilde{X}_n - \left\{ \int_{[0,\infty) \times I_n^Y} x dP(x,y) \right\} / P(y \in I_n^Y) \\ \tilde{Y}_1 - \left\{ \int_{I_1^X \times [0,\infty)} y dP(x,y) \right\} / P(x \in I_1^X) \\ \vdots \\ \tilde{Y}_m - \left\{ \int_{I_m^X \times [0,\infty)} y dP(x,y) \right\} / P(x \in I_m^X) \end{pmatrix}$$

であり、モーメント式の分散共分散行列 $\boldsymbol{\Sigma}$ は

$$\boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_p & \mathbf{0} & \boldsymbol{\Sigma}_{p\phi} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Sigma}_\mu & \boldsymbol{\Sigma}_{\mu\phi} & \boldsymbol{\Sigma}_{\mu\psi} \\ \boldsymbol{\Sigma}'_{p\phi} & \boldsymbol{\Sigma}'_{\mu\phi} & \boldsymbol{\Sigma}_\phi & \boldsymbol{\Sigma}_{\phi\psi} \\ \mathbf{0} & \boldsymbol{\Sigma}'_{\mu\psi} & \boldsymbol{\Sigma}'_{\phi\psi} & \boldsymbol{\Sigma}_\psi \end{pmatrix} \tag{6}$$

と計算できる。ただし、

$$\boldsymbol{\Sigma}_p = \begin{pmatrix} p_1(1-p_1) & -p_1p_2 & \cdots & -p_1p_m \\ -p_1p_2 & p_2(1-p_2) & \cdots & -p_2p_m \\ \vdots & \vdots & \ddots & \vdots \\ -p_1p_m & -p_2p_m & \cdots & p_m(1-p_m) \end{pmatrix}, \boldsymbol{\Sigma}_{p\phi} = \begin{pmatrix} \sigma_{1,1}^{p\phi} & \cdots & \sigma_{1,n}^{p\phi} \\ \vdots & \ddots & \vdots \\ \sigma_{m,1}^{p\phi} & \cdots & \sigma_{m,n}^{p\phi} \end{pmatrix},$$

$$\boldsymbol{\Sigma}_\mu = \begin{pmatrix} \sigma_{1,1}^{\mu} & & 0 \\ & \ddots & \\ 0 & & \sigma_{m,m}^{\mu} \end{pmatrix}, \boldsymbol{\Sigma}_{\mu\phi} = \begin{pmatrix} \sigma_{1,1}^{\mu\phi} & \cdots & \sigma_{1,n}^{\mu\phi} \\ \vdots & \ddots & \vdots \\ \sigma_{m,1}^{\mu\phi} & \cdots & \sigma_{m,n}^{\mu\phi} \end{pmatrix}, \boldsymbol{\Sigma}_{\mu\psi} = \begin{pmatrix} \sigma_{1,1}^{\mu\psi} & & 0 \\ & \ddots & \\ 0 & & \sigma_{m,m}^{\mu\psi} \end{pmatrix},$$

$$\boldsymbol{\Sigma}_\phi = \begin{pmatrix} \sigma_{1,1}^{\phi} & & 0 \\ & \ddots & \\ 0 & & \sigma_{n,n}^{\mu\phi} \end{pmatrix}, \boldsymbol{\Sigma}_{\phi\psi} = \begin{pmatrix} \sigma_{1,1}^{\phi\psi} & \cdots & \sigma_{1,m}^{\phi\psi} \\ \vdots & \ddots & \vdots \\ \sigma_{n,1}^{\phi\psi} & \cdots & \sigma_{n,m}^{\phi\psi} \end{pmatrix}, \boldsymbol{\Sigma}_\psi = \begin{pmatrix} \sigma_{1,1}^{\psi} & & 0 \\ & \ddots & \\ 0 & & \sigma_{m,m}^{\psi} \end{pmatrix},$$

$$\begin{aligned} \sigma_{k,l}^{p\phi} &= \mathrm{E}\left[\{1_k(x) - p_k\}\frac{(x-\phi_l)1_l(y)}{P(y \in I_l^Y)}\right] \quad (k = 1,\ldots,m; l = 1,\ldots,n) \\ &= \frac{1}{P(y \in I_l^Y)} \int_{I_k^X \times I_l^Y} (x - \phi_l) dP(x,y) \end{aligned}$$

$$\begin{aligned} \sigma_{k,l}^{\mu} &= \mathrm{E}\left[\frac{(x-\mu_k)1_k(x)}{P(x \in I_k^X)}\frac{(x-\mu_l)1_l(x)}{P(x \in I_l^X)}\right] \quad (k,l = 1,\ldots,n) \\ &= \begin{cases} \dfrac{1}{\{P(x \in I_k^X)\}^2} \displaystyle\int_{I_k^X} (x-\mu_k)^2 dP(x) & k = l \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} \sigma_{k,l}^{\mu\phi} &= \mathrm{E}\left[\frac{(x-\mu_k)1_k(x)}{P(x \in I_k^X)}\frac{(x-\phi_l)1_l(y)}{P(y \in I_l^Y)}\right] \quad (k = 1,\ldots,m; l = 1,\ldots,n) \\ &= \frac{1}{P(x \in I_k^X)P(y \in I_l^Y)} \int_{I_k^X \times I_l^Y} (x-\mu_k)(x-\phi_l) dP(x,y) \end{aligned}$$

$$\begin{aligned} \sigma_{k,l}^{\mu\psi} &= \mathrm{E}\left[\frac{(x-\mu_k)1_k(x)}{P(x \in I_k^X)}\frac{(y-\psi_l)1_l(x)}{P(x \in I_l^X)}\right] \quad (k,l = 1,\ldots,m) \\ &= \begin{cases} \dfrac{1}{\{P(x \in I_k^X)\}^2} \displaystyle\int_{I_k^X \times [0,\infty)} (x-\mu_k)(y-\psi_l) dP(x,y) & k = l \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} \sigma_{k,l}^{\phi} &= \mathrm{E}\left[\frac{(x-\phi_k)1_k(y)}{P(y \in I_k^Y)}\frac{(x-\phi_l)1_l(y)}{P(y \in I_l^Y)}\right] \quad (k,l = 1,\ldots,n) \\ &= \begin{cases} \dfrac{1}{\{P(y \in I_k^Y)\}^2} \displaystyle\int_{[0,\infty) \times I_k^Y} (x-\phi_k)^2 dP(x,y) & k = l \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

$$\sigma_{k,l}^{\phi\psi} = \mathrm{E}\left[\frac{(x-\phi_k)1_k(y)}{P(y\in I_k^Y)}\frac{(y-\psi_l)1_l(x)}{P(x\in I_l^X)}\right] \quad (k=1,\ldots,n; l=1,\ldots,m)$$
$$= \frac{1}{P(x\in I_l^X)P(y\in I_k^Y)}\int_{I_l^X\times I_k^Y}(x-\phi_k)(y-\psi_l)dP(x,y)$$

$$\sigma_{k,l}^{\psi} = \mathrm{E}\left[\frac{(y-\psi_k)1_k(x)}{P(x\in I_k^X)}\frac{(y-\psi_l)1_l(x)}{P(x\in I_l^X)}\right] \quad (k,l=1\ldots,m)$$
$$= \begin{cases}\dfrac{1}{\{P(x\in I_k^X)\}^2}\displaystyle\int_{I_k^X\times[0,\infty)}(y-\psi_k)^2dP(x,y) & k=l\\ 0 & \text{otherwise}\end{cases}$$

である。

同時分布の推定に際して、その形状を設定しなくてはならない。そこで、所得と年齢に対して単変量の確率分布を仮定し、それらを正規コピュラにより結合することで同時分布 $P(x,y;\boldsymbol{\theta})$ を構築した。所得については第2種の一般化ベータを仮定し、所得分布の密度関数 $f_x(x;\boldsymbol{\lambda})$ を

$$f_x(x;\boldsymbol{\lambda}) = f_x(x;(a,b,p,q)')$$
$$= \frac{ax^{ap-1}}{b^{ap}B(p,q)\{1+(x/b)^a\}^{p+q}}$$

と置く。ただし、$a>0, b>0, p>0, q>0$ である。一方、年齢の密度関数については、カーネル関数を正規カーネルとするノンパラメトリック推定を事前に行う。すなわち、

$$f_y(y) = \frac{1}{Nh}\sum_j N_j K\left(\frac{y-M_j}{h}\right)$$

と置く。ここで、N は総度数、N_j は第 j 年齢階層の度数、$K(\cdot)$ はカーネル関数、h はバンド幅である。また、カーネル関数として、正規カーネル

$$K(z) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}z^2\right)$$

を用いる。なお、カーネル推定でのバンド幅は目算により設定している。これらの密度関数から、同時密度関数 $f_{x,y}(x,y;\boldsymbol{\theta})$ を

$$f_{x,y}(x,y;\boldsymbol{\theta}) = f_1(x,y;\boldsymbol{\theta}) + f_2(x,y;\boldsymbol{\theta})$$
$$f_1(x,y;\boldsymbol{\theta}) = \begin{cases}c(x,y;\rho_1)f_x(x;\boldsymbol{\lambda}_1)f_y(y) & y\le 60\\ 0 & y>60\end{cases}$$
$$f_2(x,y;\boldsymbol{\theta}) = \begin{cases}0 & y\le 60\\ c(x,y;\rho_2)f_x(x;\boldsymbol{\lambda}_2)f_y(y) & y>60\end{cases}$$

と構築する。ただし、

$$
\begin{aligned}
c(x,y;\rho) &= \frac{\exp B}{\sqrt{1-\rho^2}} \\
B &= \frac{2\rho\Phi^{-1}(u)\Phi^{-1}(v) - \rho^2\{(\Phi^{-1}(u))^2 + (\Phi^{-1}(v))^2\}}{2(1-\rho^2)} \\
u &= \int_{-\infty}^{x} f_x(z)dz \\
v &= \int_{-\infty}^{y} f_y(z)dz
\end{aligned}
$$

で、$\Phi^{-1}(\cdot)$ は標準正規分布の累積密度関数の逆関数である。

上記の同時密度における $(\lambda_1, \lambda_2, \rho_1, \rho_2)$ が、GMM により推定されるパラメータ $\boldsymbol{\theta}$ となる。

参考文献

Joe, H. (1997) *Multivariate Models and Dependence Concepts*: Chapman and Hall/CRC.

Jondeau, E., S.-H. Poon, and M. Rockinger (2007) *Financial Modeling under Non-Gaussian Distributions*: Springer.

McDonald, J. B. (1984) "Some Generalized Functions for the Size Distribution of Income," *Econometrica*, Vol. 52, No. 3, pp. 647-663.

McDonald, J. B. and Y. J. Xu (1995) "A Generalization of the Beta Distribution with Applications," *Journal of Econometrics*, Vol. 69, No. 2, pp. 427-428.

Monahan, J. F. (2001) *Numerical Methods of Statistics*: Cambridge University Press.

Nelsen, R. B. (2006) *An Introduction to Copulas*: Springer.

NVIDIA (2016) "CUDA Toolkit Documentation v8.0," URL: `http://docs.nvidia.com/cuda/`, accessed on 10th January, 2016.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (2002) *Numerical Recipes in C++*: Cambridge University Press.

Trivedi, P. K. and D. M. Zimmer (2005) "Copula Modeling: An Introduction for Practitioners," *Foundations and Trends in Econometrics*, Vol. 1, No. 1, pp. 1-111.

$\epsilon\pi\iota\sigma\tau\eta\mu\eta$ (2015) 「.NET から CUDA を使うひとつの方法– C++/CLI による WRAPPER のつくりかた」，URL : `http://www.slideshare.net/NVIDIAJapan/1075-netcuda`（アクセス日: 2016 年 1 月 10 日）.

竹澤邦夫 (2007)『みんなのためのノンパラメトリック回帰（下）』，吉岡書店，第 3 版.

Cheng, John, Max Grossman, and Ty McKercher (2015)『CUDA C プロフェッショナルプログラミング』，森野慎也訳，インプレス，森野 慎也 訳.

福井昭吾 (2015) 「分割表が利用できない状況における世帯主の所得と年齢の同時分布の GMM 推定」，『応用経済学研究』，第 8 巻，42-68 頁.