Linux/Alpha 活用講座

清水尚彦　nshimizu@keyaki.cc.u-tokai.ac.jp

HPCプログラミング入門（第2回）

Playstation 2（以下PS2）でLinuxが動作するということが話題になっています。PS2は安価であるという以上に、その10個も搭載した積和演算器を使って、行列乗算を始めとした多くの数値計算が高速に実行できる可能性が高いので、高性能計算機として面白い存在になるのではないかと思っています。

もちろん、「ゲーム機として作られているから連続運転に耐えられるのか」など真面目に使おうとすれば、いろいろな部分をチェックしなくてはいけませんが、小さなワークロードにはPentium 4以上の性能が期待できますね。ただし、コンパイラがこれらの演算器を使いこなせないので、対応するライブラリを充実させて数値計算向けにチューニングしていくなどの地道な作業が必要になります。十分な資料が提供されるのであれば、PS2のための行列演算ライブラリを作るというのも面白いプロジェクトですね。経緯を見守っていきたいと思います。

HPC分野で必要とされる長時間連続稼働にPS2が耐えられるかどうかは、ゲーム機を使ったことのない私には何とも言えないので、その面では若干不安を感じます。確か「ファミコン」の出始めのころ、MTBF（Mean Time Between Failures、平均故障間隔）の設計値があまりに低くて、「さすがゲーム機は基準が違う」と驚いたことがありましたが、今では事情が違ってきたのでしょうか？　もっとも、超並列機ではMTBFが低いのは当り前のようになっていて、その面では「チェックポイント・リスタート」の仕組みさえ作れれば、MTBFの低さはあまり障害にはならないのかもしれません。

PS2の次世代機、「PS3（？）」のCPUは、IBMがCMOS 10Sファブで製造するらしいので、SOI（Silicon-on-insulator）技術を用いたスマートなプロセッサが量産されることになりそうですね。SCE（ソニー・コンピュータエンタテインメント）の狙いはPS3だけではないはずだから、これらのチップでLinuxが動くと、いろいろと面白い展開がありそうです。

このように、ゲーム機のような小さいマシンでも高性能を目指しているのですが、一方、いわゆるスーパーコンピュータにも新しくニュースが入っています。

CrayがAlphaのクラスタを用いて超並列機を計画しているという話を2001年5月号で書きました。その後、ベクトル機をNECからOEMで調達するという報道もなされています。マルチスレッド機をどうするのかは分かりませんが、ベクトルも超並列もハードウェアの開発から撤退するというのは、Tera Computer社の買収の経緯を考えると仕方がないのかもしれません。とはいえ、古くからのユーザーにはショッキングな出来事です。ベクトル機の市場そのものが小さくなっていく中で、スーパーコンピュータ専業ではもはや開発費を維持できなくなってきつつあるんでしょうね。

アドレスの拡張

メモリの価格低下と容量の拡大を考えると、2001年中に、ワークステーション上位機種のユーザー要求メモリ搭載量は32ビットのアドレスの限界を超えると思っています。その一方で、64ビットアプリケーションがなかなか増えていかないからか、Sun Microsystems社は、Solaris 8環境でのみ動作するワークステーションのエントリマシンを1000ドルを切った価格で提供することになりました。

私は、前からこういったエントリマシンの必要性を感じていました。というのも、これで性能を云々するのではなく、ソフトウェア開発のベースとして開発者が気軽に入手できる価格帯のマシンが必要だと考えるからです。Compaqも、かつては在庫処分などで思い切った価格のマシンを販売すること

もあったのですが、今はすっかりそういったこととは無縁になっています。1つには、x86でエントリを構成できるというお家の事情があるのでしょうが、戦略としてはあんまり得策ではありませんね。

Alpha 21164PCあたりで、インターネットアプライアンス用でもなんでもいいから量産して低価格の開発ベースマシンを出せるようになっていたら良かったのですけどね。特に現在はCompaqがコンパイラを提供している関係で、命令スケジュールが難しい21164の世代のプロセッサでも相当性能が出せるようになっていますから、余計に低価格マシンの不利な点が減少しているのですが、すでに時遅しです。

64ビットのモードしか持っていないAlphaが、開発環境として低価格で出ていれば、アプリケーションの互換性の問題はかなり改善されているはずと思うのは、歴史に「if」を求める愚なのでしょうね。

32／64の切り替え式のプロセッサでは、よほど必然性が整わないと、あえて64ビットに挑む人は少ないのではないでしょうか。Windows 2000の64ビット版は、マイクロソフト内ではAlphaで試作されていたという話もありますが、IA64がみんなが望むような値段になって世代交替するまでには、まだ少し時間がかかるような気がします。それまでは皆さんせいぜいAlphaを使って、64ビットの世界を楽しみましょう(Alphaも高いって？　Compaqに要望を出してください)。

64ビット化には、実アドレスの64ビット化と仮想アドレスの64ビット化の2段階のステップを踏んで拡張していくプロセッサの戦略もあります。これは、いにしえのIBM System/370が、24ビットアドレスが手狭になってきたときに、実アドレス拡張として実アドレスだけ26ビットをサポート可能にしたころを思い出しますが、メモリが高密度になるときに、一度は通る道のようですね。この仕組みは、インテルのIA32にもすでに備わっていて、Linuxが「64Gbytesまでサポート」と言っているのは実アドレスの拡張を使っているからです。実アドレス拡張は、ページテーブルだけいじれば簡単にサポートできるし、ユーザーモードには何の変更も必要とされないため、お手軽なアドレス拡張といえます。

IBMの370の末裔であるzSeriesでもLinuxがサポートされていますが、ここでもこの実アドレス拡張によって32ビットの範囲を越える(IBM ESA/390は31ビットですが)アドレス範囲をアクセスできるようにしています。1つのプロセスから多くのメモリを使いたいという「普通の」要求に対して、ESA/390はベースレジスタごとに別の論理空間をマッピングすることで対応していました。

同じようにIA32も、セグメントごとに別の論理空間をマッピングでき、セグメントディスクリプタをプログラム中で書き換えることができるので、理論上は4Gbytes×nの空間が使えることになるのですが、Linuxでのプログラミングモデルとの相性はあまり良いとはいえませんね。昔からのx86ユーザーはセグメントディスクリプタを切り替えながら大きな空間をアクセスするプログラミングには慣れているから、案外この方法も評判は悪くもないかもしれまんせんけど……。Linuxのプログラムにラージメモリモデルとかコンパクトメモリモデルとか導入されたりするなんて、あまり考えたくはないですね。

プロセッサを支える技術

デバイス技術の進歩とともになくてはならない製造技術の進歩ですが、インテルはEUV(遠紫外線)を用いた製造技術を開発して、あと10年はムーアの法則通り、3年で4倍のトランジスタ数の増加が見込めるとしているようです。シリコンの時代はまだまだ続きそうです。リソグラフィ技術を使って、詳細なマスクを作る技術ができたことが発表されたわけですが、国際会議IDEMで発表された0.03μmのゲート長トランジスタは、すでにゲート間の不純物の濃度が統計的な扱いができるレベルを超えていると思われるので、これらの微細加工のトランジスタを集積したチップが無事動作するのは奇跡でも見ているような気持ちがします。

プロセスだけが頑張っても、これからの大規模開発はうまくいくことはなくて、同時に回路技術や論理設計、アーキテクチャとさまざまな技術が組み合わされて初めて有効な技術として結実するので、アーキテクチャ設計者や論理設計者などまだまだやることは多いですね。

昔は、LSIから出力できる入出力ピンの数がそれほど多くは取れなかったので、エンジニアリングのトレードオフは、入出力を低減するようにアーキテクチャを設計したものです。が、前出のzSeriesのチップを見ると、信号ピンだけでも1600ピンを超えるピンが備わっているし、チップ外の信号の周波数もどんどん上がっているため、トレードオフの設計そのものも大きく変化しつつあるようです。アーキテクチャ屋が育っていかないと、たくさん使えるようになったトランジスタも、「キャッシュにする以外に使い道がない」ないなんてことにもなるかもしれませんね(もちろん、キャッシュにした方が性能が上がるのであれば、それも1つのやり方ですから否定はしません)。

アーキテクチャや論理設計の教育というと、日本でもVDEC(東京大学大規模集積システム設計教育研究センター)などの

組織によって、ようやく米国との20年のギャップを埋めようとしているということはこの連載でも何回か取り上げています。そこで使われるソフトウェアは、米国のベンダがほとんど独占的に供給しているものです。教育の裾野を広げるには、これらのソフト自身も教育の対象とすべきだと思っていますが、今まではなかなか簡単にはいかなかったのです。

ハードウェア記述言語1つとってみても、VHDLやVerilogやPARTHENONなど国内の教育で使われる言語はさまざまです。しかも、まともに言語仕様を満たした処理系は、どの言語を持ってきても、バイナリ以外供給されていませんでした。

ところが、先日ふとしたことでVerilogのフリーの処理系を見付けることができました。まだ、使い込んではいませんが、IEEEのIEEE 1364標準に準拠しているコンパイラになっていて、シミュレーションもネットリストの生成も可能になっているようです。VHDLに対しても、FreeHDLプロジェクトがあるのですが、こちらはシミュレータだけです。どちらのプロジェクトもEDAをターゲットとしているので、ネットリストのサポートまでで、レイアウトは別になっています。このあたりはElectricやAllianceとうまく統合してLinuxのEDAのツールだけで上流から下流まで流したいものですね。

Icarus Verilog Linux/Alpha版

なんで唐突にこんな話に振ったかというと、このVerilog処理系「Icarus」というのですが、実はLinux/Alphaで開発されているらしく、Alphaのバイナリも置いてあるのです。オープンソースでソースコードもあるのですが、Vine Linux 2.1を入れた私のローカル環境では構築に失敗しています(実は5月号でJavaの評価をしたとき、GCJを入れようとしてコンパイル環境をおかしくしてしまったらしく調査中なのです)。ただし、研究室のDebian GNU/Linux 2.2環境では問題なく構築できています。

これらLSI CAD関係のリンクについては、記事末のRESOURCE[1]〜[7]を参照してください。また、Verilogそのものについての資料は、30ページ程度でコンパクトに要領良くまとめられた「Handbook」があります([8])。ただし、英語なので、手こずる人は市販のVerilogの本を入手されることをお勧めします。しばらく待っていただけるなら、Hyde教授にこのドキュメントの翻訳の許可をいただいたので、翻訳して私のホームページに公開していきたいと思っています。

さて、では簡単なサンプルプログラムを流してみましょう。a.vlという名前のソースを用意します(リスト1)。これをコンパイルして実行してみると、実行例1のようになります。ここに示したように、このコンパイラは通常のプログラム言語と同様に、シミュレーション実行形式をネイティブ環境のロードモジュールに変換します。ハードウェア記述言語といってもちっともハードウェアらしくないですよね。例えばこんなこともできます。リスト2をb.vlとして保存し、同様にコンパイルして実行してみると、実行例2のようになります。

ハードウェアを記述する言語でこんなことができるなんて不思議ですね。実はこれはVerilogがハードウェアシミュレーション用の言語として作成されたという歴史があるのでシ

リスト1　a.vl

```
module main;

initial begin
$display("テスト");
$finish;
end

endmodule
```

実行例1　b.vlの実行結果

```
$ iverilog a.vl -o a
$ ./a
テスト
```

リスト2　b.vl

```
module main;
integer i;
initial begin

for(i=0; i<10; i = i + 1)
      $display("i=%d, i^2 = %d",i, i*i);
$finish;
end

endmodule
```

実行例2　b.vlの実行結果

```
$ iverilog b.vl -o b
$ ./b
i=0, i^2 = 0
i=1, i^2 = 1
i=2, i^2 = 4
i=3, i^2 = 9
i=4, i^2 = 16
i=5, i^2 = 25
i=6, i^2 = 36
i=7, i^2 = 49
i=8, i^2 = 64
i=9, i^2 = 81
```

ミュレーションの環境設定のためのさまざまなプログラム能力が備えられているのです。

では、ハードウェアのネットリストを合成してみましょう。現時点ではIcarus VerilogはXilinxのFPGA用のネットリストであるXNFだけしかサポートしていません。私はXNFのフォーマットを詳しく知っている訳ではないので簡単な回路で試してみましょう。

リスト3は、2つの入力信号i1、i2の論理和を取った結果を出力信号outに出力するというVerilogのプログラムです。これをシミュレーションしても出力するコマンドがどこにもないので何の結果も出ませんが、ネットリストにコンパイルすることはできます。上のファイルをc.vlとして次のようにコンパイルします。

```
$ iverilog -txnf -o c.xnf c.vl
```

するとc.xnfというネットリストが作成されます。その中身はリスト4のようになっています。変換したネットリストの中身はなんとなく想像できるのではないでしょうか？　もう少し論理合成らしいことができないかと、リスト5のようなプログラムを作ってみました。簡単なものですけれどね。このコードも問題なくネットリストに合成できましたが、長くなるのでここでは割愛させていただきます。さすがに乗算の合成はコンパイラから文句を言われてできなかったことを付け加えておきます。

さて、それではなんでこの連載でVerilogなんて取り上げているかというと、今までAlphaでハードウェア記述言語を使おうと思ってもなかなか良いツールがなかったからなのです。ソースが公開されていても実は結構移植が難しいものが多いのでした。

その理由の多くは、ハッシュ関数の作り方に問題があったのです。例によって、x86系などの32ビット系のプログラムの常でポインタを整数型の変数に格納できることを前提に考えており、ハッシングで検索の高速化を計るのはいいのですが、独自ハッシュ関数を用いるため、この罠にかかっているアプリケーションが多かったのです。このIcarus Verilogも当然ハッシュ関数を使っていますが、独自の物を使わずに「GNU gperf」というハッシュ生成ツールを用いています。もちろん、64ビットを意識して独自で作ってくれれば問題ないのですが、こういったすでにあるパッケージを利用するのがLinux流らしくていいですね。

リスト3　XNFによるネットリスト作成プログラム

```
module OR(i1,i2,out);
input  i1,i2;
output out;

 assign out = i1 | i2;

endmodule
```

リスト4　ネットリストc.xnf

```
LCANET,6
PROG,verilog,$Name: v0_4 $,"Icarus Verilog"
SIG, OR/i1, PIN=i1
SIG, OR/i2, PIN=i2
SIG, OR/out, PIN=out
SYM, OR/_L1, OR, LIBVER=2.0.0
    PIN, O, O, OR/out
    PIN, I0, I, OR/i1
    PIN, I1, I, OR/i2
END
EOF
```

リスト5　加算を加えたネットリスト作成プログラム

```
module ADD(in,out);
input  [0:3] in;
output [0:3] out;

assign out = in + 3;

endmodule
```

HPCプログラミング入門（第2回）

2001年3月号の1回目のプログラミング入門はいかがでしたか？　1変数の1階の微分方程式が簡単に解けてビジュアル化できることが分かったと思うのですが、一般の微分方程式に出てくる変数はもっと多いし、1階とは限らないので、そのままでは実用性はあんまりありませんでした。そこで、今回は2変数以上の微分方程式についての解法を見ていきましょう。

まずは簡単な例から始めます。

```
x1' = a1*x1
x2' = a2*x2
```

この2つの微分方程式の系を考えてみましょう。これは3月号で示した1変数の1階の微分方程式を2つ並べただけなので、すぐに解は求まって、

```
x1 = k1*e^(a1*t)
x2 = k2*e^(a2*t)
```

となります。が、このままではあまり面白さがでませんね。

「X=[x1; x2]」となるベクトルXを考えて、式を書き直すことにします。係数行列は、「A=[a1 0; 0 a2]」となる行列とすると

```
X' = A*X
```

と微分方程式は1つの式で書き表せます。これを積分すると

```
X = e^(A.*t)*K
```

となります。ここでKは「K=[k1; k2]」となる定数を表すベクトルを表します。Aとtの積に「.*」という演算子を使っているのはスカラーtをAの各要素に掛けることを表しています。これを3月号でスカラーの変数を扱ったときのように、コマンド一発でパッと計算できると格好良いのですが、そう簡単にはいきません。そこで、tの各値ごとに別々に計算して結果をベクトルに保持してグラフ化することにします。といっても100個のデータを計算するのに100個のコマンドを打つのは情けないので、ループ構文を用います。

ここで使うループ構文は、FORTRANのDOに相当する構文で、

```
for <var> = <init>{:<step>}:<end>
⋮
文
⋮
endfor
```

といった形になります。「:<step>」はオプションで、指定されない場合、変数「<var>」の値は、初期値「<init>」から「<end>」まで1ずつ増加します。「<step>」が指定されると、指定された増分ずつの増加になります。

では、実際に値を入れて試してみましょう。octaveを起動して、次のように、AとKを設定します。ここの値は適当に決めています。設定値は自分の解きたい問題に合わせて変えてください。

```
A = [1 0; 0 -2];
K = [1; -1];
```

次に変数tを少しずつ変えながら微分方程式の解を求めていきます(リスト6)。

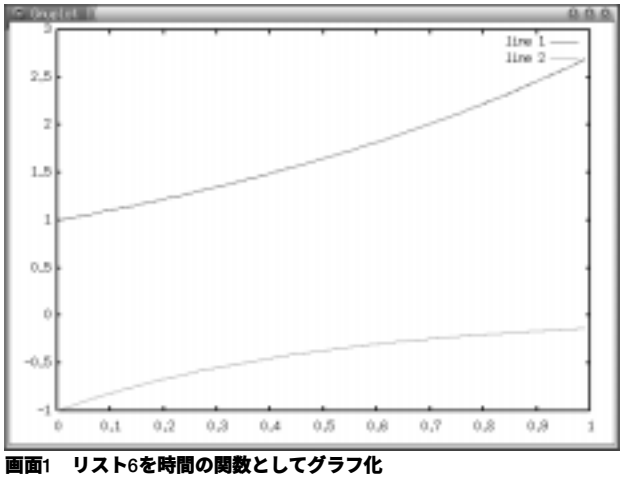
ここで、「ちょっと変なやり方をしている」と思われるかもしれません。代入文の左辺に直接「X(i)」のように、添字が使えれば簡単に表記できそうです。しかしoctaveでは、添字を持った変数への代入はスカラー値もしくは同じ長さのベクトルしか許されていないため、解ベクトルをいったん変数Xに入れた後、スカラーとしてベクトルx1、x2に代入し直すというやり方をしているのです。

ともかく、これで微分方程式の解が得られているはずですので、グラフを表示して確かめてみましょう。時間の関数として2つの変数を表示する場合には「plot(t,x1,t,x2)」のように入力すると、画面1のようになります。また、この解曲線を表示するには「plot(x1,x2)」と入力すると画面2のようになります。解曲線も1本だけでは方程式の味が出ないので、初期値

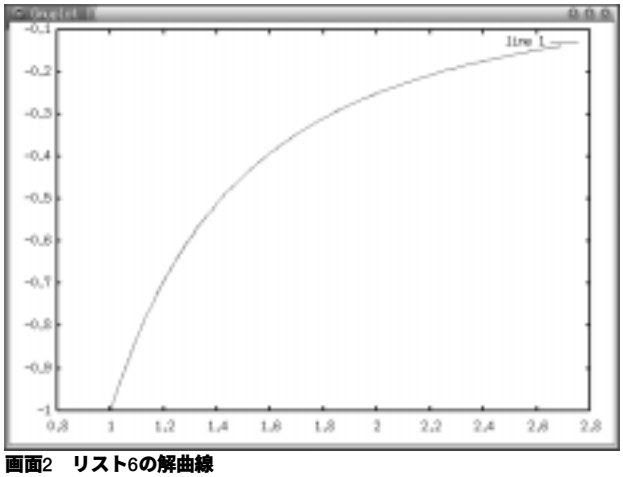
リスト6　微分方程式の解を求める

```
t = [0:0.01:0.99];

for i = 1:100
  X = e^(A.*t(i))*K;
  x1(i) = X(1);
  x2(i) = X(2);
endfor
```



画面1　リスト6を時間の関数としてグラフ化



画面2　リスト6の解曲線

リスト7　複数の解曲線を表示する方程式

```
k1 = [1; 1];
k2 = [1; -1];
k3 = [-1; 1];
k4 = [-1; -1];

for i = 1:100
  X = e^(A.*t(i))*k1;
  x11(i) = X(1);
  x12(i) = X(2);
  X = e^(A.*t(i))*k2;
  x21(i) = X(1);
  x22(i) = X(2);
  X = e^(A.*t(i))*k3;
  x31(i) = X(1);
  x32(i) = X(2);
  X = e^(A.*t(i))*k4;
  x41(i) = X(1);
  x42(i) = X(2);
endfor

plot(x11,x12,x21,x22,x31,x32,x41,x42)
```

を変えて複数の解曲線を出してみましょう(リスト7)。初期値の異なる微分方程式の解曲線の動きが把握できますね(画面3)。

ところで、Aの形が変わったらどうなるのでしょうか？　リスト7を同じ式でAを変化させてみましょう。まずは、

```
A = [1 0.5; -0.5 1];
```

として前出のfor文をもう一度実行してみます。いかがですか？　解曲線はかなり変化しましたね(画面4)。ついでにもう1つ見てみましょう。

```
A = [2 1; -3 -2];
```

とします(画面5)。これらのグラフから微分方程式の解の定性的な様子が見えるのではないでしょうか？

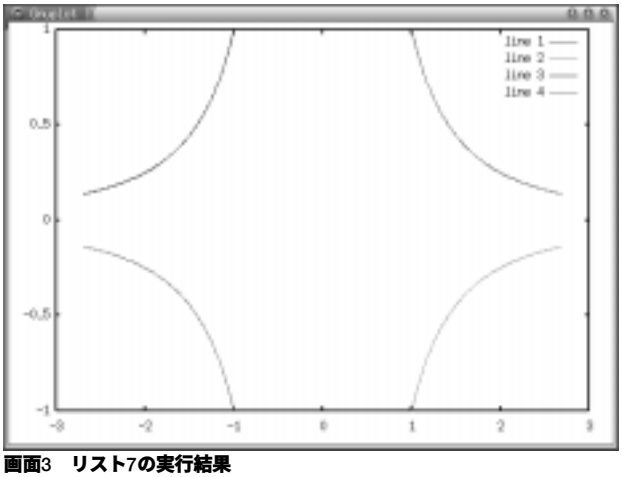
もう少し初期値をいろいろ変えてグラフを表示してみると、もっとよく分かります。「hold on」コマンドでプロットしたグラフを重ね合わせてみましょう。重ね合わせの状態で初期値をランダムに選んでグラフを作ってみると(リスト8)、結果は画面6のようになります。重ね合わせを解除する場合には「hold off」とコマンドを入力してください。

さらに、リスト9のようにしてグラフを表示してみてください(画面7)。時間に対する解の動きが読み取れます。もっとも、2つの変数の曲線が一緒になってしまっているので、もう少し工夫して分かりやすくした方がいいですね。

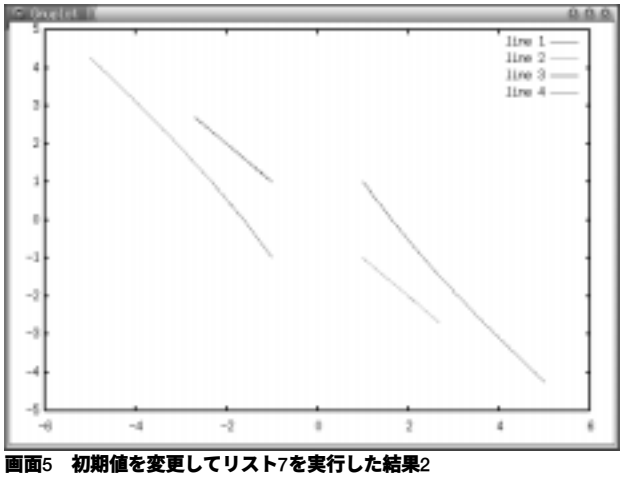
●最後に

CompaqがAlpha LinuxのCプログラマーのためのチューニングドキュメントを発表しています([9])。現在Compaqとこ

画面4　初期値を変更してリスト7を実行した結果1



画面3　リスト7の実行結果



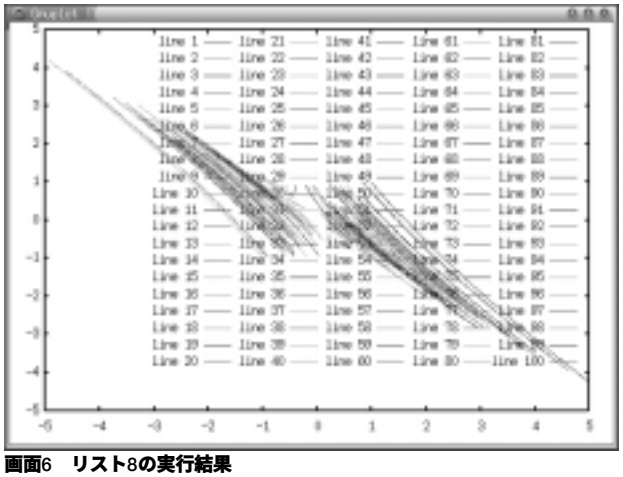
画面5　初期値を変更してリスト7を実行した結果2

リスト8　初期値をランダムに選んで重ね合わせるプログラム

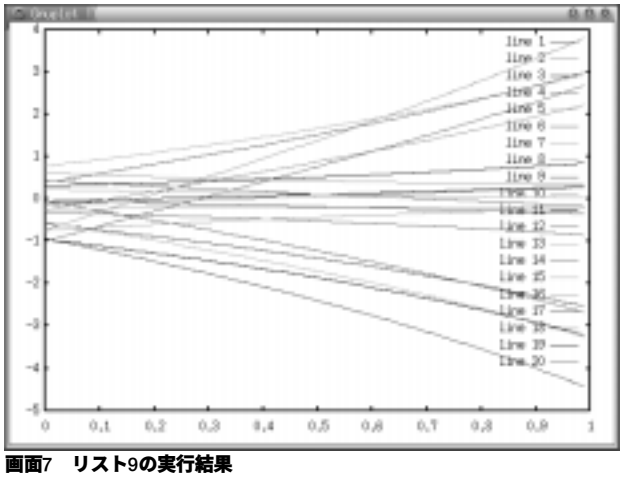
```
for j=1:100
  k(1)=(rand-0.5)*2;
  k(2)=(rand-0.5)*2;
  for i=1:100
    x=e^(A*t(i))*k;
    t1(i)=x(1);
    t2(i)=x(2);
  endfor
  plot(t1,t2)
endfor
```

リスト9　時間軸での動きを表示させるプログラム

```
for j=1:10
  k(1)=(rand-0.5)*2;
  k(2)=(rand-0.5)*2;
  for i=1:100
    x=e^(A*t(i))*k;
    t1(i)=x(1);
    t2(i)=x(2);
  endfor
  plot(t, t1, t, t2)
endfor
```



画面6　リスト8の実行結果



画面7　リスト9の実行結果

のドキュメントの翻訳権について交渉中です。無事翻訳権が取れたらこの連載の中で紹介していこうと思っています(コンパックの承認作業が進行中で基本的にはOKとなっていますので、早期に掲載できるように努力していきます)。

R E S O U R C E

[1] **Icarus**
http://www.icarus.com/eda/verilog/index.html

[2] **FreeHDL**
http://www.freehdl.seul.org/

[3] **gEDA**
http://www.geda.seul.org/

[4] **Alliance**
http://www-asim.lip6.fr/alliance/

[5] **Electric**
http://www.gnu.org/software/electric/

[6] **VHDL**
http://www.vhdl.org/

[7] **PARTHENON**
http://www.kecl.ntt.co.jp/parthenon/

[8] **Handbook**
http://www.eg.bucknell.edu/~cs320/1995-fall/verilog-manual.html

[9] **Alpha LinuxのCプログラマーのためのチューニングドキュメント**
http://www.alphalinux.org/docs/perf.html