カラー・リカレンス by gnuplot with image

2023/06/19, 29 山田 泰司 <taiji@aihara.co.jp> and 高橋 純 <junt@aihara.co.jp>

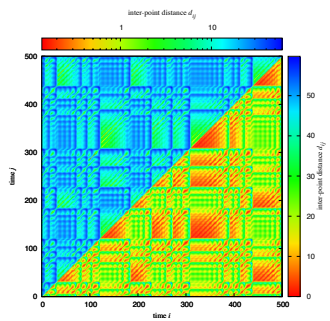
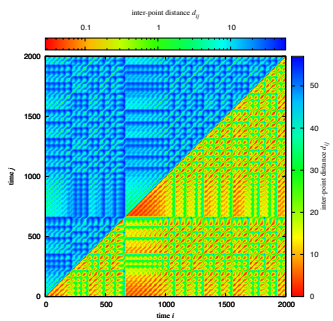
はじめに

gnuplot でカラー・リカレンスプロットを書くとき、普通に plot コマンドを用いるとファイルサイズが大きくなりがちで、ゆえに解像度を上げることが困難になる。本稿ではスケーラブル・ベクタ形式 (EPS, PDF, SVG) のなかにラスタ形式の画像を埋め込む plot with image コマンドによる方法を紹介する。

ファイルサイズの比較

まずは結論から、EPS, PDF, SVG 形式のファイルサイズの比較を示す。

表 1 内部形式によるファイルサイズの比較

\ 分岐図の内部形式	ベクタ形式 — vbirfucation	ラスタ形式 — pbifurcation
時系列長	500	2000
EPS ファイルのサイズ (kB)	5352	2500
PDF ファイルのサイズ (kB)	7232	5184
SVG ファイルのサイズ (kB)	19056	3840
挿入図 (ローレンツアトラクタ)		

このように EPS 内ラスタ形式では 4^2 倍の画素数でもファイルサイズが半分程度になる。よって、ラスタ形式であればさらに解像度を上げることが容易である。ちなみに、SVG は cairo ライブラリを gnuplot 内で利用しており、重量級の図には適さないと思われる。また、gnuplot にて `set`

`terminal eps` とすると cairo ライブラリを利用した EPS となり (epscairo)、設計上、これも効率的な PostScript とはなっておらず、重量級の図には適さない。よって、`set terminal postscript eps level3` とするとよい[1]。

データファイルの作成方法

しかし、ラスタ形式では生成しておくデータ形式は座標 (i, j) 昇順で普通に印字すればよく、近傍のみ印字したベクタ形式のファイルよりもサイズが大きくなる。詳しくは文献 [2] のソースコードで事足りるが、以下に一部を紹介する。

```
double l2norm(const std::vector<double> a, const std::vector<double> b)
{
  double d = 0;
  for (size_t i = 0; i < a.size(); ++i)
    d += (a[i] - b[i])*(a[i] - b[i]);
  return std::sqrt(d);
}
std::vector<double> runge_kutta_map(double &t, const double delta_t,
  std::function<std::vector<double>(const double t,
  const std::vector<double>)> dvdt, const std::vector<double> v0)
{
  std::vector<double> dv(v0.size()), d1(v0.size()), d2(v0.size()),
       d3(v0.size()), va(v0.size()), v1(v0.size());
  dv = dvdt(t, v0);
  for (size_t i=0; i<v0.size(); ++i) {
    d1[i] = delta_t*dv[i];
    va[i] = v0[i] + d1[i]/2;
  }
  dv = dvdt(t + delta_t/2, va);
  for (size_t i=0; i<v0.size(); ++i) {
    d2[i] = delta_t*dv[i];
    va[i] = v0[i] + d2[i]/2;
  }
  dv = dvdt(t + delta_t/2, va);
  for (size_t i=0; i<v0.size(); ++i) {
    d3[i] = delta_t*dv[i];
    va[i] = v0[i] + d3[i]/2;
  }
  dv = dvdt(t + delta_t, va);
  for (size_t i=0; i<v0.size(); ++i)
    v1[i] = v0[i] + (d1[i]+d2[i]*2+d3[i]*2+delta_t*dv[i])/6;
  t += delta_t;
  return v1;
}
#define x v0[0]
#define y v0[1]
#define z v0[2]
static std::vector<double> map(double &t, const double dt,
       const std::vector<double> v0)
{
  std::function<std::vector<double>(const double t,
       const std::vector<double>)> df =
       [](const double t, const std::vector<double> v0) ->
       std::vector<double> {
    (void)t;
    std::vector<double> dvdt(v0.size());     // Lorenz equations
    dvdt[0] = -y - z;
    dvdt[1] = x + a*y;
    dvdt[2] = b + z*(x - c);
    return dvdt;
  };
  return runge_kutta_map(t, dt, df, v0);
}
#undef x
#undef y
#undef z

int main(int argc, char *argv[])
{
       : // 中略
  switch (t_output_format) {
  case pixels_output_format:
    { // plot with image 向けの出力
      std::vector<std::vector<double> > V(n_iteration,
       std::vector<double>(v0.size()));
```

[1] level3 としないと PostScript level1 となり内部のイメージデータが圧縮されないので、ベクタ形式の方式と同程度のファイルサイズとなる。

```
    std::vector<std::vector<double> > M(n_iteration,
      std::vector<double>(n_iteration));
    v0 = parameters.v;
    for (int i = 0; i < n_transient; ++i)
      v0 = map(t = delta_t*i, delta_t, v0);
    for (int i = 0; i < n_iteration; ++i) {
      v1 = map(t = delta_t*i, delta_t, v0);
      V[i] = v1;
      for (int j = 0; j < i; ++j) {
        const double d = l2norm(V[i], V[j]);
        M[i][j] = M[j][i] = d;
      }
      v0 = v1;
    }
    for (int i = 0; i < n_iteration; ++i)
      for (int j = 0; j < n_iteration; ++j)
        std::cout << i << '\t' << j << '\t' << M[i][j]
          << std::endl;
  }
  break;
default:
  { // plot with dots 向けの出力
```

```
    std::vector<std::vector<double> > V(n_iteration,
      std::vector<double>(v0.size()));
    v0 = parameters.v;
    for (int i = 0; i < n_transient; ++i)
      v0 = map(t = delta_t*i, delta_t, v0);
    for (int i = 0; i < n_iteration; ++i) {
      v1 = map(t = delta_t*i, delta_t, v0);
      V[i] = v1;
      for (int j = 0; j < i; ++j) {
        const double d = l2norm(V[i], V[j]);
        std::cout << i << '\t' << j << '\t' << d << std::endl;
        std::cout << j << '\t' << i << '\t' << d << std::endl;
      }
      const int j = i;
      const double d = 0;
      std::cout << i << '\t' << j << '\t' << d << std::endl;
      v0 = v1;
    }
  }
  break;
}
    : // 略
```

参考文献

[1] T. Williams & C. Kelley, “gnuplot 5.4 — An Interactive Plotting Program, “ http://www.gnuplot.info/docs_5.4/gnuplot-ja.pdf, 2022.

[2] T. Yamada, https://www.aihara.co.jp/~taiji/lecture/recurrences-color-gnuplot-20230619.tar.xz, 2023.

[3] 平田 祥人、陳 洛南、合原 一幸、非線形時系列解析の基礎理論、東京大学出版会、2023. (注) 装丁の要望により、我々は 10000^2 画素数のカラー・リカレンスプロットをこの書籍の表紙に提供した。EPS, PDF, SVG 形式において、先に記したように 2000^2 画素のファイルサイズが 2500, 5184, 3840 kB に対して、25 倍の 10000^2 画素のファイルサイズは 20 倍強の 51076, 114752, 89500 kB となる。

付録 プロット方法と出力ファイルの編集

本稿のリカレンスプロットでは、上対角は対数スケールになっている。そこで、PDF, SVG では NaN を利用して下対角の線形軸の描画では $!(i < j)$ は NaN、上対角の対数軸の描画では $i < j$ は NaN というように互いに一方の対角を透明にしている。

しかし、この方法は `set terminal postscript eps` の EPS だと一方の対角が透明にはならずに、意図しない色で塗りつぶされてしまう。

そこで、致し方がないので、線形軸での上対角への対数スケールを式 (1) のように自前で算出してプロットし、さらに、対数軸での下対角への逆対数スケールを式 (2) のように自前で算出してプロットする (後者の描画は対数スケールのカラーバーを描画するためだけであって、等価な画像が 2 枚重なっており、その無駄は後に対処する)。

$$f(x) = u\frac{\log(x) - \log(l)}{\log(u) - \log(l)} \qquad \cdots (1)$$

$$f^{-1}(x) = le^{x\frac{\log(u) - \log(l)}{u}} \qquad \cdots (2)$$

このように gnuplot では、グラフは実際には描かずにカラーバーのみ描くということが、筆者が調べた限りでは実現できないと思われる。

さて、EPS 出力形式、及び、SVG 出力形式にて、本稿のようなカラー・リカレンスプロットを実現する gnuplot スクリプトを紹介する。

EPS ファイルの生成と編集

```
lb=0.0284443; ub=57.0224;
scale=1.25; xsize=5; ysize=3.5; vsize=3.09;
set terminal postscript eps enhanced level3 size xsize*scale,ysize*scale;
set palette model HSV functions gray*4/6,1,1;
set xlabel '{/Times-Roman time {/Times-Italic i}}';
set ylabel '{/Times-Roman time {/Times-Italic j}}';
set size square;
set xrange [0:2000];
set yrange [0:2000];
set cblabel '{/Times-Roman inter-point distance {/Times-Italic d_{ij}}}';

set tmargin at screen .1125+(vsize/scale)/ysize;
set bmargin at screen .1125;
set lmargin at screen ((xsize-(vsize/scale))/2)/xsize;
set rmargin at screen ((xsize-(vsize/scale))/2+(vsize/scale))/xsize;

set multiplot;

set colorbox vertical default;
```

```
set cbrange [0:ub];
plot 'filename.dat' using 1:2:(!($$1 < $$2) ? $$3 : ub*(log($$3)-log(lb))/(log(ub)-log(lb))) with image notitle;

set colorbox horizontal user origin ((xsize-(vsize/scale))/2)/xsize,.8375 size vsize/scale/xsize,.035;
set cblabel offset 0,7.5;
set cbtics offset 0,4,0;
set cbrange [lb:ub]; set log cb;
plot 'filename.dat' using 1:2:(($$1 < $$2) ? $$3 : lb*exp($$3*(log(ub)-log(lb))/ub)) with image notitle;

unset multiplot;
```

このように生成した EPS 出力ファイルにて、以下のようにすると 2 枚目の画像を除去する。

```
perl -ne 'if (!$$sw) {
  $$sw = !0 if (/^%%%%EndImage$$/);
  print;
}
else {
  $$flag = !0 if (/^%%%%BeginImage$$/);
  print if (!$$flag);
  $$flag = 0 if (/^%%%%EndImage$$/);
}' -i '$@'
```

ここで「$@」は makefile 中のターゲットファイルである。いずれにしても、ページ記述言語 PostScript 3 及び W3C SVG 1.1 はオープン規格であるので、なんとかなるのである。

SVG ファイルの生成

```
lb=0.0284443; ub=57.0224;
scale=1.2; xsize=600; ysize=480; vsize=404.4;
set terminal svg enhanced size xsize*scale,ysize*scale;
set palette model HSV functions gray*4/6,1,1;
set xlabel '{/Times-Roman time {/:Italic i}}';
set ylabel '{/Times-Roman time {/:Italic j}}';
set size square;
set xrange [0:2000];
set yrange [0:2000];
set cblabel '{/Times-Roman inter-point distance {/:Italic d_{ij}}}';

set tmargin at screen .1125+(vsize/scale)/ysize;
set bmargin at screen .1125;
set lmargin at screen ((xsize-(vsize/scale))/2)/xsize;
set rmargin at screen ((xsize-(vsize/scale))/2+(vsize/scale))/xsize;
```

```
set multiplot;

set colorbox vertical default;
set cbrange [0:ub];
plot 'filename.dat' using 1:2:(!($$1 < $$2) ? $$3 : NaN) with image notitle;

set colorbox horizontal user origin ((xsize-(vsize/scale))/2)/xsize,.8375 size vsize/scale/xsize,.035;
set cblabel offset 0,6;
set cbtics offset 0,3,0;
set cbrange [lb:ub]; set log cb;
plot 'filename.dat' using 1:2:(($$1 < $$2) ? $$3 : NaN) with image notitle;

unset multiplot;
```

このように生成した SVG 出力ファイルは、先の EPS 形式のような 2 枚目の画像を除去する必要はなく、1 枚目が線形スケールの下対角、2 枚目が対数スケールの上対角の画像となっている。