システム工学概論

3. Cによる数値計算と可視化（導入編）

山田泰司

taiji@aihara.co.jp

株式会社あいはら 研究開発チーム

数値計算に必要なテクニック

- 問題領域に応じたアルゴリズム本、参考書、論文を探す技術及び問題解決能力 — やる気と経験、及び考え抜くこと
- 試行錯誤しやすいプログラミング技術 — プログラムの完成品を目指すよりも、様々な側面から問題に取り組むことが重要。但し、手続きにおいて共有する部分をうまく抽出し、適度なコード再利用によりバグの生じ難いプログラミングを心がける。
- 数値計算結果の理解や表現を助ける可視化技術 — 論文に載せるためにはグラフで表現する技術が必要。それ以上に、自らが理解を深めるには可視化された数値計算結果を積極的に観ることが重要。

（例題１）差分方程式：ロジスティック写像

以下のロジスティック写像と呼ばれる差分方程式：

$$x_{t+1} = ax_t(1 - x_t)$$

において $a = 4$, $x_0 = 0.1$ **における** $t < 5000$ **までの値を出力せよ。**

（例題１）の解答例

```
#include <stdio.h>  /* printf */
int main(void)
{
  int i;
  double a = 4, x0 = 0.1, x1;

  for (i=0; i<5000; i++) {
    x1 = a*x0*(1-x0);
    printf("%g\n", x0);
    x0 = x1;
  }
  return 0;
}
```

（例題２）差分方程式：ミラの写像

以下のグモウスキー・ミラの写像と呼ばれる差分方程式：

$$\begin{aligned} x_{t+1} &= y_t + 0.008(1 - 0.05y_t^2)y_t + F(x_t) \\ y_{t+1} &= -x_t + F(x_{t+1}) \end{aligned}$$

(但し、 $F(x) = \mu x + 2(1-\mu)x^2/(1+x^2)$**) において** $\mu = -0.8$, $x_0 = 0.1$, $y_0 = 0$ **における** $t < 5000$ **までの値を出力せよ。**

（例題２）差分方程式：任意の写像

以下の差分方程式：

$$x_{t+1} = F(p, x_t)$$

(但し、$p \in \mathbb{R}^K$, $x \in \mathbb{R}^N$**) において、**$t < 5000$ **までの値を出力せよ。**

（例題２）の解答例

```
#include <stdio.h>  /* printf */
struct de {
  int K, N;
  void (*init)(double p[], double v0[]);
  void (*map)(double p[], double v0[], double v1[]);
};
        :
int main(void)
{
  int i, j;
  double p[de->K], v0[de->N], v1[de->N];

  de->init(p, v0);
  for (i=0; i<5000; i++) {
    de->map(p, v0, v1);
    for (j=0; j<de->N; j++) printf("%g\t", v0[j]);
    printf("\n");
    for (j=0; j<de->N; j++) v0[j] = v1[j];
  }
  return 0;
}
```

（例題２）の解答例（部分）

```
          :
#define mu p[0]
#define x0 v0[0]
#define y0 v0[1]
#define x1 v1[0]
#define y1 v1[1]
#define gm_F(x) (mu*x + 2*(1-mu)*x*x/(1+x*x))
void gm_init(double p[], double v0[])
{
  mu = -0.8;
  x0 = 0.1; y0 = 0.0;
}
void gm_map(double p[], double v0[], double v1[])
{
  x1 = y0 + 0.008*(1 - 0.05*y0*y0)*y0 + gm_F(x0);
  y1 = -x0 + gm_F(x1);
}
struct de gm_de = {
  1, 2, gm_init, gm_map,
}, *de = &gm_de;
          :
```

（例題３）差分方程式：池田写像

以下の池田写像と呼ばれる差分方程式：

$$\begin{aligned} x_{t+1} &= q + b(x_t \cos(\theta) - y_t \sin(\theta)) \\ y_{t+1} &= b(x_t \sin(\theta) + y_t \cos(\theta)) \end{aligned}$$

(但し、 $\theta = k - \alpha/(1 + x_t^2 + y_t^2)$**) において** $q = 1$, $k = 0.4$, $\alpha = 6$, $b = 0.9$, $x_0 = 0.1$, $y_0 = 0$ **における** $t < 5000$ **までの値を出力せよ。**

（例題３）の解答例（部分）

```
          :
#define q p[0]
#define k p[1]
#define alpha p[2]
#define b p[3]
          :
#define ikeda_theta (k-alpha/(1+x0*x0+y0*y0))
void ikeda_init(double p[], double v0[])
{
  q = 1; k = 0.4; alpha = 6; b = 0.9;
  x0 = 0.1; y0 = 0.0;
}
void ikeda_map(double p[], double v0[], double v1[])
{
  x1 = q+b*(x0*cos(ikeda_theta)-y0*sin(ikeda_theta));
  y1 =   b*(x0*sin(ikeda_theta)+y0*cos(ikeda_theta));
}
struct de ikeda_de = {
  4, 2, ikeda_init, ikeda_map,
}, *de = &ikeda_de;
          :
```

（例題１）の解答例（部分）

```
          :
#define a p[0]
#define x0 v0[0]
#define x1 v1[0]
void logistic_init(double p[], double v0[])
{
  a = 4;
  x0 = 0.1;
}
void logistic_map(double p[], double v0[], double v1[])
{
  x1 = a*x0*(1-x0);
}
struct de logistic_de = {
  1, 1, logistic_init, logistic_map,
}, *de = &logistic_de;
          :
```

数値計算プログラミングへのコメント

- 先の例にて、真のモジュール化を目指すのであれば、各写像に関するコードを例えば `ikeda.c` へ、共通化された `struct de` に関するコードをファイル `de.h` へ、同じく共通化された `main` 関数を `de.c` へモジュール毎にファイルを分け、これらをコンパイルしたすべてのオブジェクトをリンクした実行形式 `de` を作成する事となる (参照: `de-20061108.tar.gz`)。
- しかし、そういった完成形に必ずしも執着しなくてもよい。例えば、問題に直面したときに新たな試みとして構造体を拡張する必要性が生じたとして、しかし変更が広範に及ぶ場合、変更を躊躇してしまう可能性がある。きちんとしたモジュール化はコードの保守等も行ないやすいので、しないよりもした方がよいのであるが、数値計算においてはそれが目的になってはならない。

（例題４）差分方程式：ミラの写像

以下のグモウスキー・ミラの写像と呼ばれる差分方程式：

$$\begin{aligned} x_{t+1} &= y_t + 0.008(1 - 0.05y_t^2)y_t + F(x_t) \\ y_{t+1} &= -x_t + F(x_{t+1}) \end{aligned}$$

(但し、 $F(x) = \mu x + 2(1 - mu)x^2/(1 + x^2)$**) において任意の** μ**,** x_0, y_0 **における** $t < 5000$ **までの値を出力せよ。**

（例題４）の解答例

```
#include <stdio.h>  /* printf */
#include <stdlib.h> /* atof */
#include <string.h> /* strcmp */
struct de {
  int K, N;
  void (*init)(double p[], double v0[]);
  void (*options)(int argc, char *argv[], double p[], double v0[]);
  void (*map)(double p[], double t, double v0[], double v1[]);
};
        :
int main(int argc, char *argv[])
{
  int i, j;
  double p[de->K], v0[de->N], v1[de->N];

  de->init(p, v0);
  de->options(argc, argv, p, v0);
  for (i=0; i<5000; i++) {
    de->map(p, i, v0, v1);
    for (j=0; j<de->N; j++) printf("%g\t", v0[j]);
    printf("\n");
    for (j=0; j<de->N; j++) v0[j] = v1[j];
  }
  return 0;
}
```

（例題４）の解答例（部分）

```
          :
void gm_options(int argc, char *argv[], double p[], double v0[])
{
  int i;

  for (i=1; i<argc; i++)
    if (strcmp(argv[i], "-mu") == 0 && i+1 < argc)
      mu = atof(argv[i+1]);
    else if (strcmp(argv[i], "-x") == 0 && i+1 < argc)
      x0 = atof(argv[i+1]);
    else if (strcmp(argv[i], "-y") == 0 && i+1 < argc)
      y0 = atof(argv[i+1]);
}
struct de gm_de = {
  1, 2, gm_init, gm_options, gm_map,
}, *de = &gm_de;
          :
```

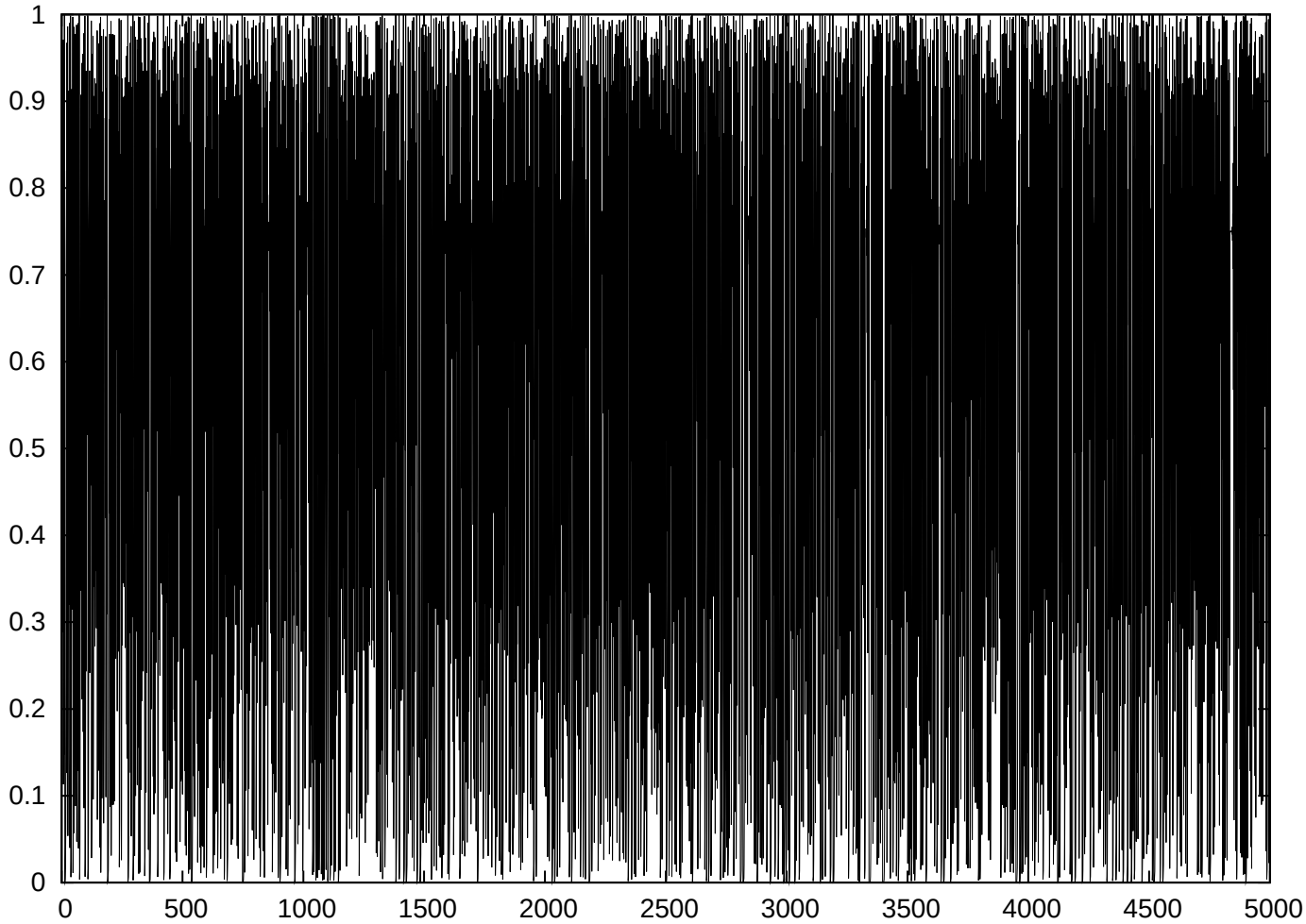
（例題５）数値計算結果の可視化（グラフ編）

例題１～３の実行ファイルの作成に加え、数値計算結果を gnuplot を使ってグラフ化する makefile を書け。

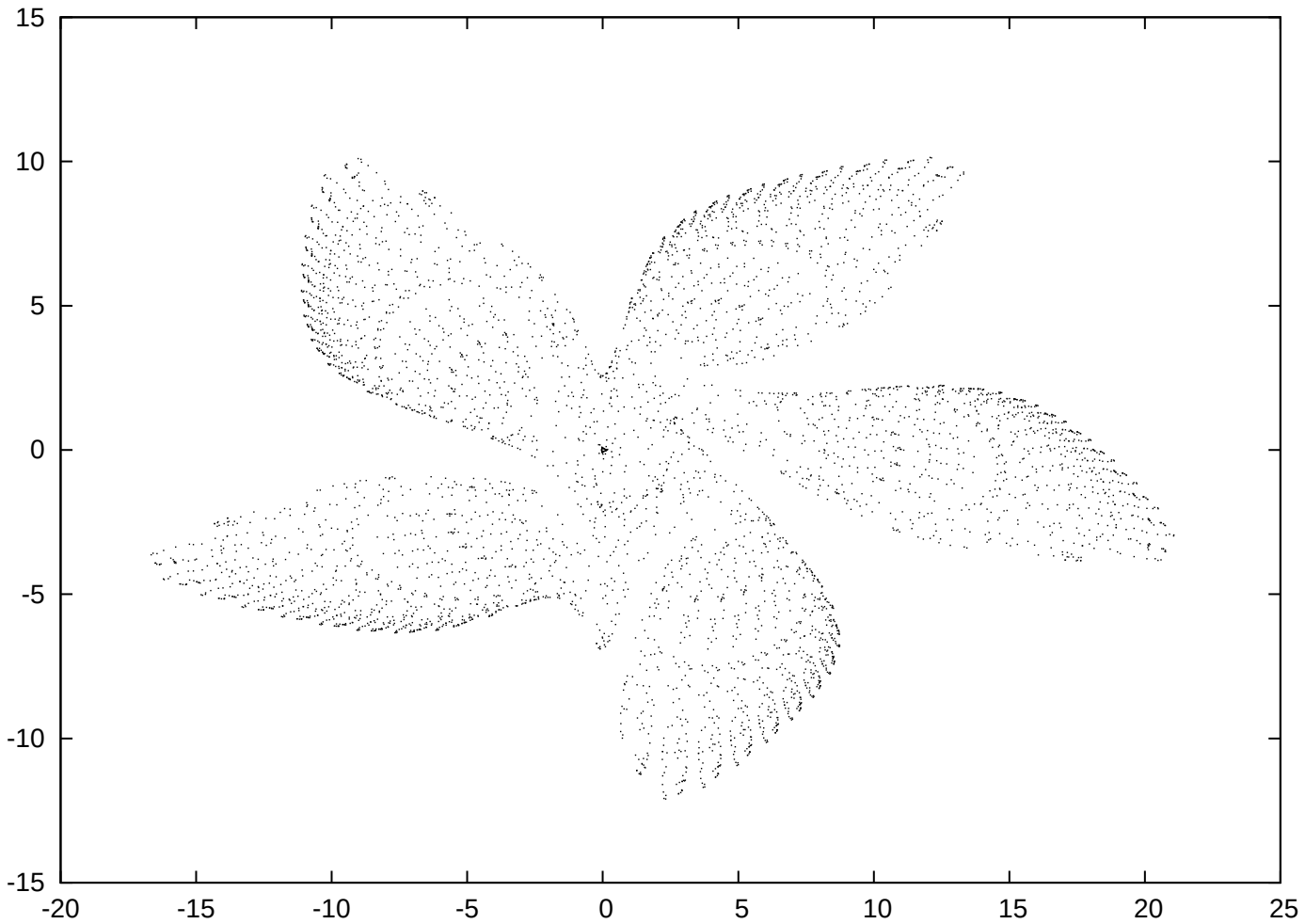
（例題５）の解答例

```
LDLIBS=-lm

SRCS=na00.c na01.c na02.c

EXES=$(SRCS:%.c=%)

DATS=$(EXES:%=%.dat)

EPSS=$(DATS:%.dat=%.eps)

all: $(EXES) $(DATS) $(EPSS)

%.dat: %
        ./$< > $@

%.eps: %.dat
        echo "set terminal postscript eps;\
set output '$@';\
plot '$<' notitle with dots;" | gnuplot -
```

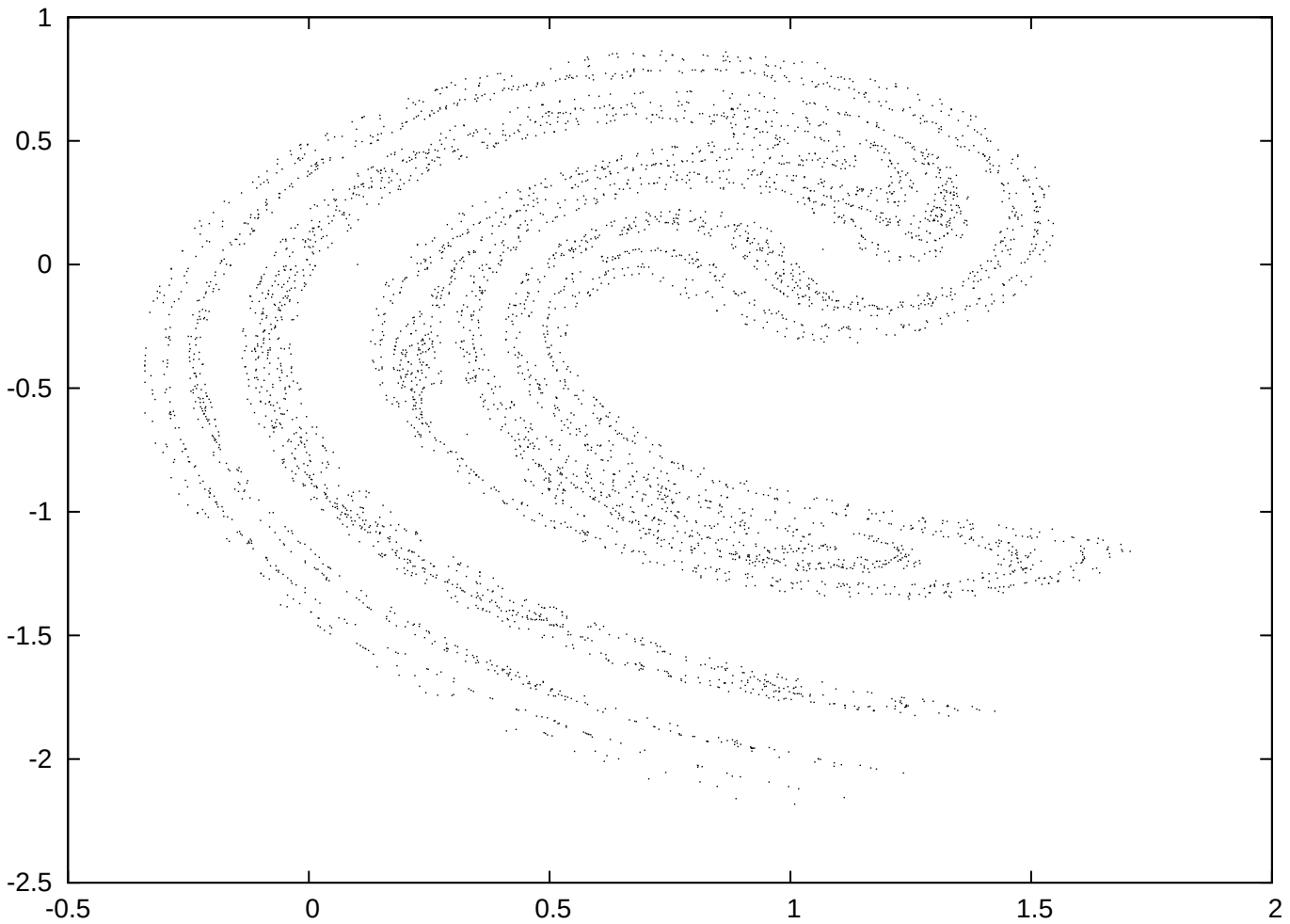
（例題５）の結果（ロジスティック写像）



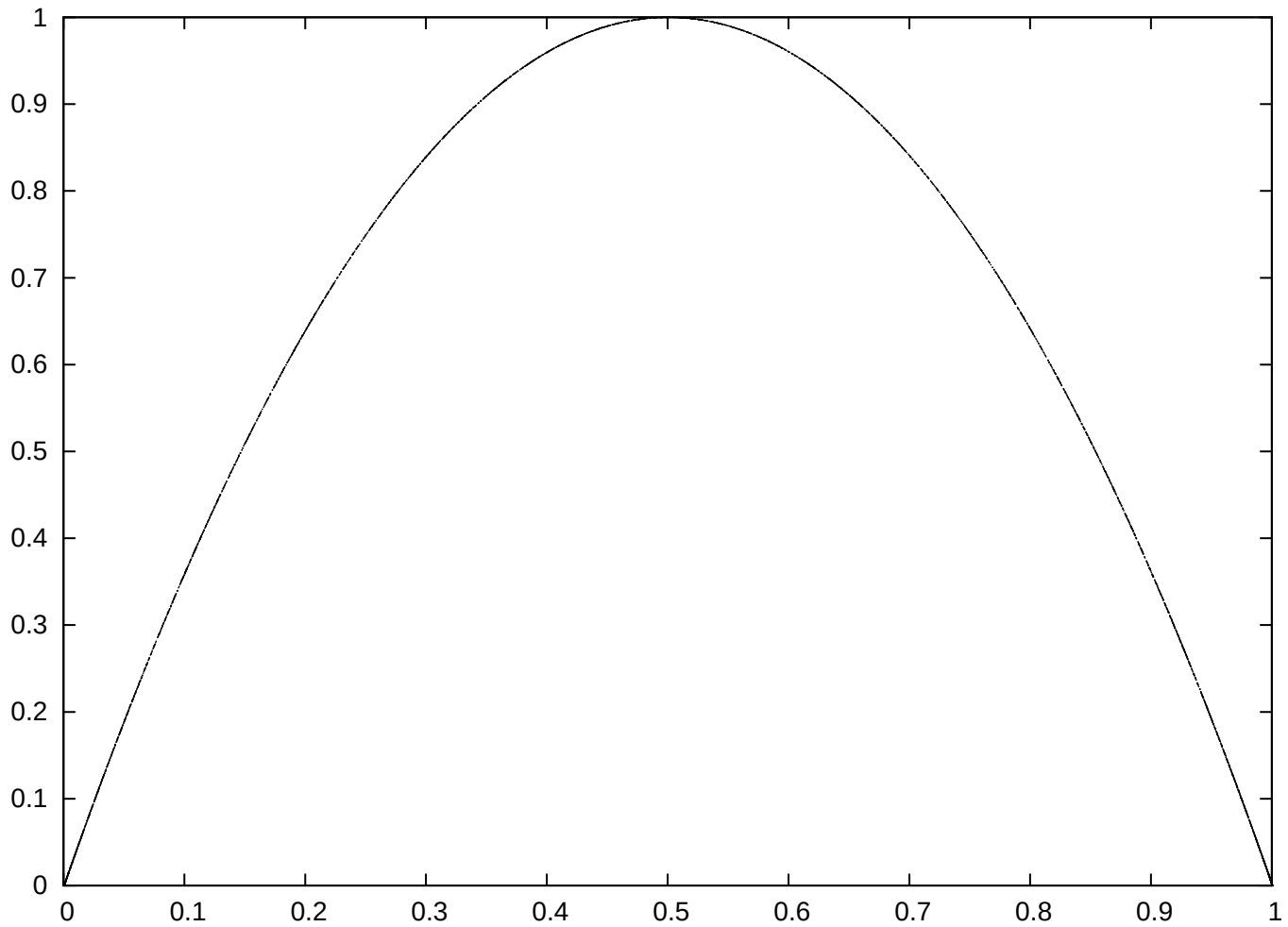
（例題５）の結果（ミラの写像）



（例題５）の結果（池田写像）



（例題５）の結果（ロジスティック写像）



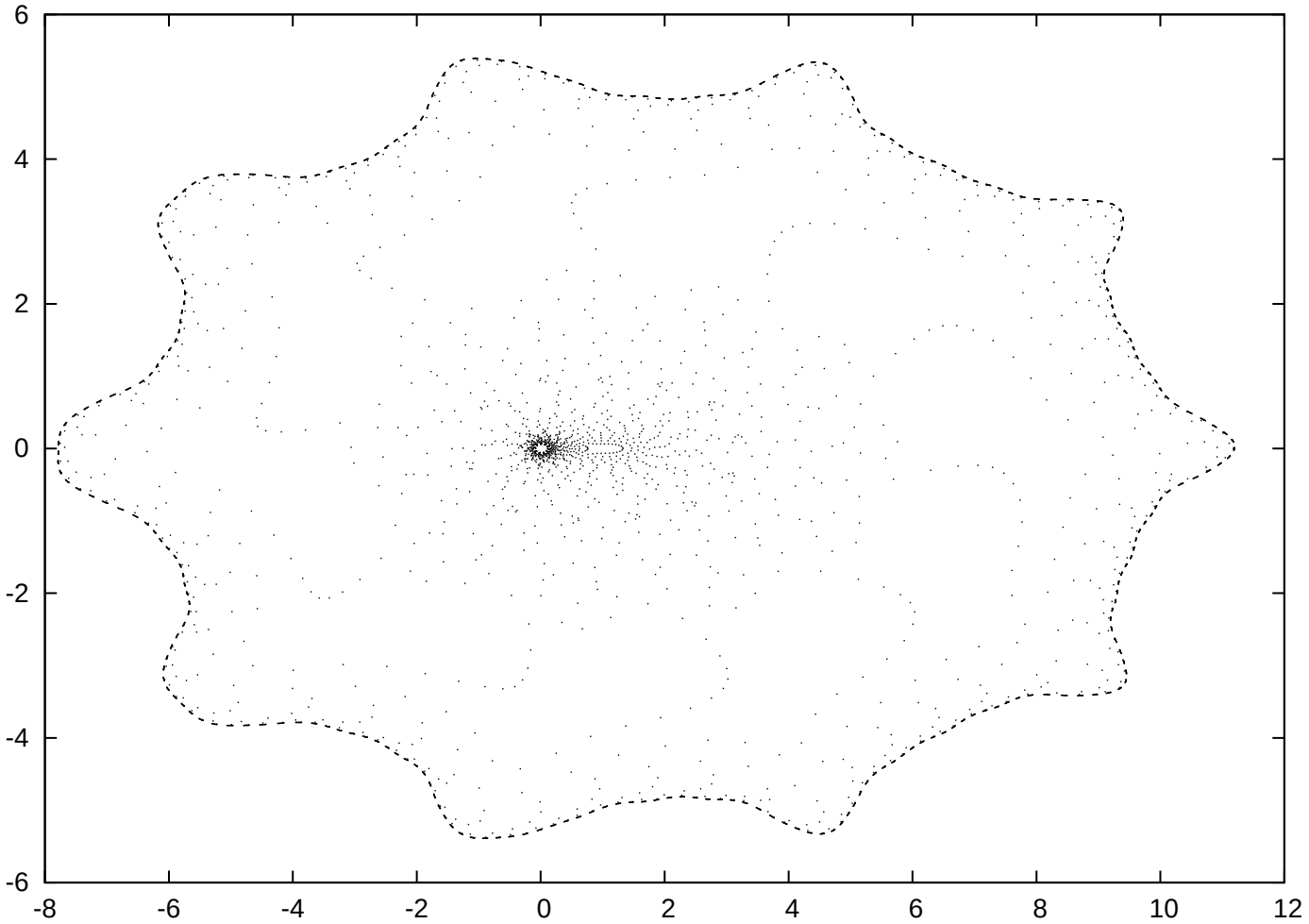
（例題６）数値計算試行と可視化（グラフ編）

例題４の実行ファイルを使って、いろいろなパラメータにおける数値計算結果を `gnuplot` **を使ってグラフ化するシェルスクリプトを書け。**

（例題６）の解答例

```
#!/bin/sh
for mu in -0.8 -0.4 -0.2 -0.1 0.1 0.2 0.4 0.8; do
  ./na04 -mu $mu > na04-mu$mu.dat
  cat <<EOF | gnuplot -
set terminal postscript eps
set output "na04-mu$mu.eps"
plot "na04-mu$mu.dat" notitle with dots
EOF
done
```

（例題５）の結果（ミラの写像）



数値計算プログラミング：まとめ

- 試行錯誤しやすいプログラミング技術 — 問題の抽象化及び一般化により、様々な問題に取り組めるようにする工夫、但し、テクニックが目的になってはならない。また、オプション (プログラム引数) の活用により、プログラムの挙動を容易に制御する工夫。
- 数値計算結果の理解や表現を助ける可視化技術 — 今回は論文執筆で著名な `gnuplot` で数値計算結果のグラフ描画を紹介。