＜目次＞

１　日程 2
２　関連サイト 2
３　ロジスティック回帰 3
４　ロジスティック回帰の可視化 8
５　ロジスティック回帰＋確率的勾配降下法 19
６　線形SVM 24
７　Anaconda を利用した Python のインストール 29
８　PYTHON プログラム 34

１　日程

【第１回】　Python とは何か？　　※Anaconda のインストール
【第２回】　「データから学習する能力」をコンピュータに与える（第１章）
【第３回】　分類問題 −機械学習アルゴリズムのトレーニング（第２章）
【第４回】　分類問題 −機械学習ライブラリ scikit-learn の活用（第３章）
【第５回】　データ前処理 −よりよいトレーニングセットの構築（第４章）
【第６回】　次元削減でデータを圧縮する（第５章）
【第７回】　モデル評価とハイパーパラメータのチューニングのベストプラクティス（第６章）
【第８回】　アンサンブル学習 −異なるモデルの組み合わせ（第７章）
【第９回】　機械学習の適用１ −感情分析（第８章）
【第１０回】　機械学習の適用２ −Web アプリケーション（第９章）
【第１１回】　回帰分析 −連続値をとる目的変数の予測（第１０章）
【第１２回】　クラスタ分析 −ラベルなしデータの分析（第１１章）
【第１３回】　ニューラルネットワーク −画像認識トレーニング（第１２章）
【第１４回】　ニューラルネットワーク
−数値計算ライブラリ Theano によるトレーニングの並列化(第１３章)
【第１５回】　まとめ

２　関連サイト

- ANACONDA
- JUPYTER
- k-means（可視化）
- MLDEMOS（可視化）
- PYTHON 機械学習プログラミング（ダウンロード）
- すうがくぶんか
- カーネルトリック（可視化）
- ギブスサンプリング（可視化）
- テキストマイニングツール
- テンソルフロー
- ニューラルネットワーク（可視化）
- ニューラルネット（可視化）
- ハミルトニアンモンテカルロ（可視化）
- バックプロパゲーション（可視化）
- パラメトリック推定（可視化）
- ヤフーファイナンス
- リッジ回帰（可視化）
- ロジスティック回帰（可視化）

- 人工知能(可視化)
- 個人投資家が株式市場で勝てない本当の理由
- 多層パーセプトロン関数近似(動画)
- 多層パーセプトロン(動画)
- 大阪取引所
- 日新ビジネス開発
- 日本個人投資家育成協会
- 日本投資顧問業協会
- 日本証券業協会
- 日本銀行
- 最急降下法(可視化)
- 東京証券取引所
- 株価予測(グーグル)
- 機会学習パーセプトロン(動画)
- 機会学習MATLAB
- 混合ディレクレ分布(可視化)
- 画像分類(Youtube)
- 金融庁
- BLM(PYTHON での実装)
- EMアルゴリズム(可視化)
- SVMソフトマージン(可視化)
- SVMハードマージン(可視化)
- SVM動画
- SVM(youtube)

3　ロジスティック回帰

今回は実践編，前回導出した確率的勾配降下法でのロジスティック回帰の学習を実装してみましょう。
環境はこれまでと同じく Python/numpy/matplotlib を用います。インストールなどの準備は第 6 回を参照してください。

パーセプトロンの実装の復習

今回のロジスティック回帰の実装は，連載第 17 回のパーセプトロンの実装と非常によく似たものになります。
そこでパーセプトロンの実装を再掲しておきます。パーセプトロンそのものの復習はここではしませんので，必要なら連載第 15 回をご覧ください。

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import random

# データ点の個数
N = 100

# データ点のために乱数列を固定
np.random.seed(0)

# ランダムな N×2 行列を生成 = 2 次元空間上のランダムな点 N 個
X = np.random.randn(N, 2)

def h(x, y):
    return 5 * x + 3 * y - 1   #  真の分離平面 5x + 3y = 1

T = np.array([ 1 if h(x, y) > 0 else -1 for x, y in X])

# 特徴関数
def phi(x, y):
    return np.array([x, y, 1])

w = np.zeros(3)  # パラメータを初期化(3 次の 0 ベクトル)

np.random.seed() # 乱数を初期化
while True:
    list = range(N)
    random.shuffle(list)

    misses = 0 # 予測を外した回数
    for n in list:
        x_n, y_n = X[n, :]
        t_n = T[n]

        # 予測
        predict = np.sign((w * phi(x_n, y_n)).sum())

        # 予測が不正解なら、パラメータを更新する
        if predict != t_n:
            w += t_n * phi(x_n, y_n)
            misses += 1

    # 予測が外れる点が無くなったら学習終了(ループを抜ける)
    if misses == 0:
        break

# 図を描くための準備
seq = np.arange(-3, 3, 0.02)
xlist, ylist = np.meshgrid(seq, seq)
zlist = np.sign((w * phi(xlist, ylist)).sum())

# 分離平面と散布図を描画
plt.pcolor(xlist, ylist, zlist, alpha=0.2, edgecolors='white')
```

```
plt.plot(X[T== 1,0], X[T== 1,1], 'o', color='red')
plt.plot(X[T==-1,0], X[T==-1,1], 'o', color='blue')
plt.show()
```

ロジスティック回帰の実装

これをロジスティック回帰の実装に差し替えるにあたって，変更する必要があるのはラベル変数の値，予測確率の計算と，パラメータ w の更新式，あとはパラメータの初期化とループの終了条件です。
順に見ていきましょう。

人工データの生成

パーセプトロンは線形分離可能な問題しか解けなかったので（連載第 15 回参照），このコードでは線形分離可能なデータを生成しています。
一方，ロジスティック回帰は線形非分離な問題でも大丈夫なのですが，ここではパーセプトロンと同じ問題をきちんと（期待通りに）解けるかを確認しておきましょう。
というわけでデータ生成のコードは基本そのまま使いますが，正解ラベルの値だけはロジスティック回帰にあわせて変更する必要があります。
パーセプトロンの正解ラベルは+1 と-1 であるのに対し，ロジスティック回帰は 1 と 0 を使うため，その点のみ反映したコードは次のようになります。

```
TT = np.array([ 1 if f(x, y) > 0 else 0 for x, y in X])
```

これに関連して，出力の図を描くところも少し変更します。
パーセプトロンによって得られる予測は「正か負か」，特に+1 または-1 であり，描画する必要があるのは分離平面だけでした。一方，ロジスティック回帰の予測は確率，つまり 0 から 1 の間で変化する関数として得られます。
よって，なだらかに変化する予測分布を描画したいですから，そのための書き換えのついでに，今回は matplotlib の imshow()という関数を使ってみましょう。

```
plt.imshow(zlist, extent=[-3,3,-3,3], origin='lower', cmap=plt.cm.PiYG_r)
```

imshow 関数は，引数 cmap に描画する色マップを指定します。PiYG_r の場合は，予測分布が 1 に近い点はピンク（Pi），0 に近い点は緑（G）に，そしてちょうど中間は白っぽい黄色（Y）となるグラデーションで描画されます（_r は逆順を意味する）。

予測確率とパラメータの更新式

パーセプトロンにしてもロジスティック回帰にしても，「データを選んで，予測して，それを使ってパラメータを更新」という大きい流れは全く同じです。
パーセプトロンのコードでは，訓練データセットからランダムに 1 つ取りだした点（x_n,y_n），その点での正解ラベル t_n を使って予測とパラメータの更新をし，それをデータセットを一周するまで繰り返すということをしていますが，その枠組みはロジスティック回帰でもそのまま使えるということですね。
一方，予測やパラメータの更新式はモデルの要ですから，こちらは当然書き換える必要があるでしょう。
ロジスティック回帰の予測確率は次の式で表されるのでした。

$$y = \sigma(\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}))$$

ただし $\sigma(z) = \frac{1}{1+\exp(-z)}$

これを実装するには，まずシグモイド関数 σ(z)が必要ですね。このシグモイド関数やその多変数版にあたるソフトマックス関数は numpy か scipy で標準サポートしてくれててもいいのになあとか個人的には思うのですが，無いものは仕方ありません。
せっかくですから関数として実装しておきましょう。

```
# シグモイド関数
```

```
def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))
```
上の式では予測確率に y の文字が当てられていますが，その文字はすでにデータ点の y 座標を表すのに使ってしまいました。ここではパーセプトロンのコードにあわせて predict（予測）という名前の変数を使うことにしましょう。
φ_n もこの後またすぐ使いますから，これは変数 feature（特徴，素性）に入れておきます。すると次のようなコードになります。
```
        # 特徴量と予測確率
        feature = phi(x_n, y_n)
        predict = sigmoid(np.inner(w, feature))
```
特徴関数 phi は定数項にあたる 1 を付け加えただけの，元のパーセプトロンのコードと同じものを使います。
予測確率がわかると，パラメータを更新できます。
ロジスティック回帰の確率的勾配降下法では，次の式でパラメータ w を更新します。

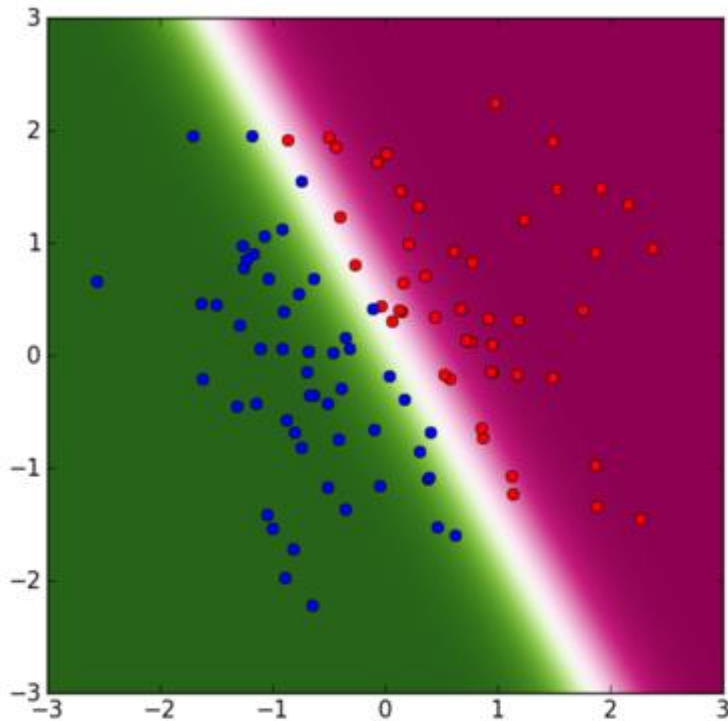
$$\boldsymbol{w}^{(i+1)} = \boldsymbol{w}^{(i)} - \eta \cdot (y_n - t_n)\boldsymbol{\phi}(\boldsymbol{x}_n)$$

η は「学習率」と呼ばれる適当な正の値で，一回の更新でパラメータをどれくらい動かすかを調整します。η が大きいほど学習は速くなりますが，予測確率が安定しません。そこで初期値は 0.1 くらいにして，学習が進むにつれて徐々に小さくしていく，というのが一般的です。
```
# 学習率の初期値
eta = 0.1
    # イテレーションごとに学習率を小さくする
    eta *= 0.9
```
この eta を使ってパラメータを更新します。
```
        w -= eta * (predict - t_n) * feature
```

パラメータの初期値と学習の終了条件

後はパラメータ w の初期化と学習の終了条件だけです。この 2 つは実は少々関連があります。
パーセプトロンの解法アルゴリズムは，0 で初期化したパラメータで始めるとすべてのデータ点を完全に正しく分離する解にたどり着くことが示されています。
この時の学習ループの終了条件は明確で，「すべてのデータ点での予測が正解と一致するまで」となります。「線形分離可能な問題に限る」という強い条件が付いているだけあって，非常に強い性質を持っているわけですね。
一方の確率的勾配降下法で求まるのは，パラメータ w の初期値に依存した局所解になります。そこでランダムな初期値で始めるのが一般的ですが，パラメータが大きくなりすぎるのを嫌って，あえて 0 から始めることもあります。
```
w = np.random.randn(3)  # パラメータをランダムに初期化
```
そうして求めた局所解は，一般に全データ点で正解を予測できるものにはなりませんので，学習ループの終了条件もパーセプトロンのままというわけにはいきません。
そこで確率的勾配降下法では尤度などの指標値があまり変化しなくなったら終了したり，あるいはあらかじめ学習の回数を適当に決めておきます。
今回はシンプルに学習回数を 50 周と決めておきましょう。
これらをすべて反映したコードは次のようになります。
```
import numpy as np
import matplotlib.pyplot as plt

# データ点の個数
N = 100
```

```
# データ点のために乱数列を固定
np.random.seed(0)

# ランダムな N×2 行列を生成 = 2 次元空間上のランダムな点 N 個
X = np.random.randn(N, 2)

def f(x, y):
    return 5 * x + 3 * y - 1  #  真の分離平面 5x + 3y = 1

T = np.array([ 1 if f(x, y) > 0 else 0 for x, y in X])

# 特徴関数
def phi(x, y):
    return np.array([x, y, 1])

# シグモイド関数
def sigmoid(z):
    return 1.0 / (1 + np.exp(-z))

np.random.seed() # 乱数を初期化
w = np.random.randn(3)  # パラメータをランダムに初期化

# 学習率の初期値
eta = 0.1

for i in xrange(50):
    list = range(N)
    np.random.shuffle(list)

    for n in list:
        x_n, y_n = X[n, :]
        t_n = T[n]

        # 予測確率
        feature = phi(x_n, y_n)
        predict = sigmoid(np.inner(w, feature))
        w -= eta * (predict - t_n) * feature

    # イテレーションごとに学習率を小さくする
    eta *= 0.9

# 図を描くための準備
seq = np.arange(-3, 3, 0.01)
xlist, ylist = np.meshgrid(seq, seq)
zlist = [sigmoid(np.inner(w, phi(x, y))) for x, y in zip(xlist, ylist)]

# 散布図と予測分布を描画
plt.imshow(zlist, extent=[-3,3,-3,3], origin='lower', cmap=plt.cm.PiYG_r)
plt.plot(X[T==1,0], X[T==1,1], 'o', color='red')
plt.plot(X[T==0,0], X[T==0,1], 'o', color='blue')
plt.show()
```



さて，長々と 20 回続けてきた本連載ですが，次回で最終回となります。
最終回は連載の流れの中では触れられなかった落ち穂拾いをしつつ，これまでのまとめをしようと思います。

４　ロジスティック回帰の可視化

ロジスティック回帰（勾配降下法 / 確率的勾配降下法）を可視化する

可視化 Python
いつの間にかシリーズ化して、今回はロジスティック回帰をやる。自分は行列計算ができないクラスタ所属なので、入力が 3 次元以上 / 出力が多クラスになるとちょっときつい。教科書を読んでいるときはなんかわかった感じになるんだが、式とか字面を追ってるだけだからな、、、やっぱり自分で手を動かさないとダメだ。
また、ちょっとした事情により今回は Python でやりたい。Python のわかりやすい実装ないんかな？と探していたら 以下の ipyton notebook を見つけた。
http://nbviewer.ipython.org/gist/mitmul/9283713
こちらのリンク先に 2 クラス/多クラスのロジスティック回帰（確率的勾配降下法）のサンプルがある。ありがたいことです。理論的な説明も書いてあるので ロジスティック回帰って何？という方は上を読んでください（放り投げ)。
この記事ではロジスティック回帰で使われる勾配法について書きたい。

補足 この記事ではデータを pandas で扱う / matplotlib でアニメーションさせる都合上、元のプログラムとは処理/変数名などだいぶ変わっているため見比べる場合は注意。

補足 実際の分析で使いたい場合は scikit-learn の SGDClassifier 。

サンプルデータのロードと準備

iris で、"Species" を目的変数にする。
補足 rpy2 を設定している方は rpy2 から、そうでない方は こちら から .csv でダウンロードして読み込み（もしくは read_csv のファイルパスとして直接 URL 指定しても読める)。

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import matplotlib.animation as animation

# iris の読み込みはどちらかで

# rpy2 経由で R から iris をロード
# import pandas.rpy.common as com
# iris = com.load_data('iris')

# csv から読み込み
# http://aima.cs.berkeley.edu/data/iris.csv
names = ['Sepal.Length', 'Sepal.Width', 'Petal.Length', 'Petal.Width', 'Species']
iris = pd.read_csv('iris.csv', header=None, names=names)

np.random.seed(1)
# 描画領域のサイズ
FIGSIZE = (5, 3.5)
```

ロジスティック回帰（2 クラス)

まず iris データの一部を切り出して、説明変数 2 次元 / 目的変数 2 クラスのデータにする。

```
# 2 クラスにするため、setosa, versicolor のデータのみ抽出
data = iris[:100]

# 説明変数は 2 つ = 2 次元
columns = ['Petal.Width', 'Petal.Length']

x = data[columns]            # データ（説明変数)
y = data['Species']          # ラベル（目的変数)
```

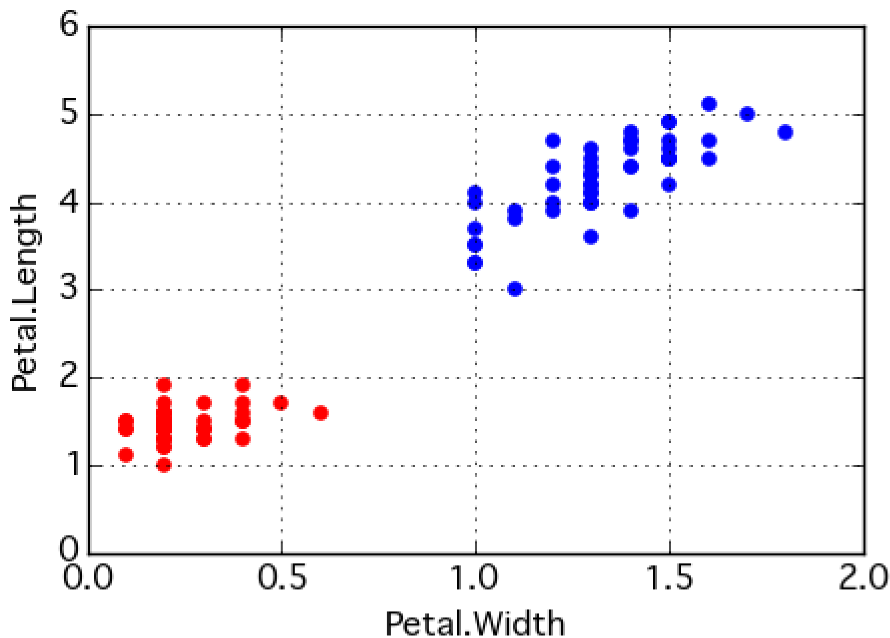
ロジスティック回帰での目的変数は 0, 1 のみからなるベクトルである必要があるので、以下のようにして変換。論理演算で bool の pd.Series を作り、int 型に変換すればよい。

```
# ラベルを 0, 1 の列に変換
y = (y == 'setosa').astype(int)
y
# 1      1
# 2      1
```

```
# ...
# 99        0
# 100       0
# Name: Species, Length: 100, dtype: int64
```

プロットしてみる。線形分離できるのでサンプルとしては OK だろう。

```
def plot_x_by_y(x, y, colors, ax=None):
    if ax is None:
        # 描画領域を作成
        fig = plt.figure(figsize=FIGSIZE)

        # 描画領域に Axes を追加、マージン調整
        ax = fig.add_subplot(1, 1, 1)
        fig.subplots_adjust(bottom=0.15)

    x1 = x.columns[0]
    x2 = x.columns[1]
    for (species, group), c in zip(x.groupby(y), colors):
        ax = group.plot(kind='scatter', x=x1, y=x2,
                        color=c, ax=ax, figsize=FIGSIZE)
    return ax

plot_x_by_y(x, y, colors=['red', 'blue'])
plt.show()
```



最適化法

ロジスティック回帰における予測値 $\hat{y}$ は、$\hat{y} = \sigma(xw + b)$ で計算される。それぞれの意味は、

- $\hat{y}$: 真のラベル y の予測値ベクトル。次元は（入力データ数, 1）
- σ: シグモイド関数。計算結果を（0, 1）区間に写像する

- x：入力データ。次元は（入力データ数, 説明変数の数）
- w：係数ベクトル。次元は(クラス数 = 2)
- b：バイアス（スカラー）

ロジスティック回帰では、予測値 y^と真のラベル yの差から計算される損失関数を最小化する回帰直線を求めることになる。この回帰直線を求める方法として勾配法が知られている。勾配法とは、損失関数から 勾配（Gradient）を求めることによって損失関数をより小さくする w, b を逐次更新していく手法（詳細は上記リンク)。

ここでは以下 3 種類の勾配法を試す。上のリンクでやっているのは 3 つ目の ミニバッチ確率的勾配降下法 / MSGD になる。

- 勾配降下法（Gradient Descent)
- 確率的勾配降下法（Stochastic Gradient Descent / SGD)
- ミニバッチ確率的勾配降下法（Minibatch SGD / MSGD)

※ MSGD は SGD と特に区別されない場合も多い。

まず、共通で使う関数を定義する。

```
def p_y_given_x(x, w, b):
    # x, w, b から y の予測値（yhat）を計算
    def sigmoid(a):
        return 1.0 / (1.0 + np.exp(-a))
    return sigmoid(np.dot(x, w) + b)

def grad(x, y, w, b):
    # 現予測値から勾配を計算
    error = y - p_y_given_x(x, w, b)
    w_grad = -np.mean(x.T * error, axis=1)
    b_grad = -np.mean(error)
    return w_grad, b_grad
```

勾配降下法（Gradient Descent)

勾配法の中ではもっともシンプルな考え方で、全データ（x, y）を使って勾配を求める。実装は以下のようになる。

引数の意味は、

- x, y, w, b：上記数式に対応
- eta：学習率（勾配をどの程度 w, b に反映させるか)
- num：イテレーション回数

補足 num の既定値はそれぞれの手法がある程度収束する値にしている。ちゃんとやる場合は収束判定すべき。

```
def gd(x, y, w, b, eta=0.1, num=100):
    for i in range(1, num):
        # 入力をまとめて処理
        w_grad, b_grad = grad(x, y, w, b)
        w -= eta * w_grad
        b -= eta * b_grad
        e = np.mean(np.abs(y - p_y_given_x(x, w, b)))
        yield i, w, b, e
```

実行してみる。w, b の初期値は全て 0 のベクトル / スカラーにした。

```
# w, b の初期値を作成
w, b = np.zeros(2), 0
gen = gd(x, y, w, b)

# gen はジェネレータ
gen
# <generator object gd at 0x11108e5f0>

# 以降、gen.next() を呼ぶたびに 一回 勾配降下法を実行して更新した結果を返す。
# タプルの中身は (イテレーション回数, w, b, 誤差)
gen.next()
# (1, array([-0.027    , -0.06995]), 0.0, 0.47227246182037463)

gen.next()
# (2, array([-0.04810306, -0.12007078]), 0.0054926687253766763, 0.45337584157628485)

gen.next()
# (3, array([-0.06522081, -0.15679859]), 0.014689105435764377, 0.44035208019270661)

gen.next()
# (4, array([-0.07963026, -0.18443466]), 0.026392079742058178, 0.43101983106700503)

# ...
```

実行のたびに w, b が更新されるとともに、誤差の値が小さくなっていることがわかる。ここでの誤差 は 誤判別率 ではなく、真のクラスと クラス所属確率の差。

<u>確率的勾配降下法</u> (Stochastic Gradient Descent / <u>SGD</u>)

勾配降下法では勾配の計算を全データを行列とみて行うため、データ量が増えると計算が厳しい。<u>確率的勾配降下法</u>では、入力をランダムに (確率的に) 並べ替えて、データ一行ずつから勾配を求めて係数/バイアスを更新する。一行ずつの処理になるため計算量の問題は解消する。実装は以下のようになる。

```
def sgd(x, y, w, b, eta=0.1, num=4):
    for i in range(1, num):
        for index in range(x.shape[0]):
            # 一行ずつ処理
            _x = x.iloc[[index], ]
            _y = y.iloc[[index], ]
            w_grad, b_grad = grad(_x, _y, w, b)
            w -= eta * w_grad
            b -= eta * b_grad
            e = np.mean(np.abs(y - p_y_given_x(x, w, b)))
            yield i, w, b, e
```

確率的に処理するため、データをランダムに並べ替える。ランダムに並べ替えられた結果の表示から、x, y の index が一致している = 対応関係が維持されていることを確認。

```
# index と同じ長さの配列を作成し、ランダムにシャッフル
indexer = np.arange(x.shape[0])
np.random.shuffle(indexer)
```

```
# x, y を シャッフルされた index の順序に並び替え
x = x.iloc[indexer, ]
y = y.iloc[indexer, ]

x.head()
#        Petal.Width   Petal.Length
# 81           1.1           3.8
# 85           1.5           4.5
# 34           0.2           1.4
# 82           1.0           3.7
# 94           1.0           3.3

y.head()
# 81     0
# 85     0
# 34     1
# 82     0
# 94     0
# Name: Species, dtype: int64
```

実行してみよう。

```
# w, b の初期値を作成
w, b = np.zeros(2), 0
gen = sgd(x, y, w, b)

# 以降、gen.next() を呼ぶたびに 一行分のデータで更新した結果を返す。
# タプルの中身は (イテレーション回数, w, b, 誤差)
gen.next()
# (1, array([-0.055, -0.19 ]), -0.050000000000000003, 0.43367318357380596)

gen.next()
# (1, array([-0.09571092, -0.31213277]), -0.077140615717 20722, 0.40715863230 43157)

gen.next()
# (1, array([-0.08310602, -0.22389845]), -0.014116100082250033, 0.42197958122210588)
```

gen.next によって 一行処理した結果が返ってくる。一行あたりの処理をみると損失は増加していることもある（が徐々に減っていく)。

補足 イテレーション回数は全データに対する回数のため、データが一回りするまで増えない。

ミニバッチ確率的勾配降下法（Minibatch SGD / MSGD）

確率的勾配降下法について、一行ずつではなく 計算できる程度の固まり（Minibatch）単位で処理していく方法。実装は以下のようになる。

※ 事前にデータをランダムに並べ替えるのは確率的勾配降下法と同様。

引数はこれまでから一つ増えて、

- batch_size：Minibatch として切り出すデータ数（行数)

```
def msgd(x, y, w, b, eta=0.1, num=25, batch_size=10):
    for i in range(1, num):
```

```
        for index in range(0, x.shape[0], batch_size):
            # index 番目のバッチを切り出して処理
            _x = x[index:index + batch_size]
            _y = y[index:index + batch_size]
            w_grad, b_grad = grad(_x, _y, w, b)
            w -= eta * w_grad
            b -= eta * b_grad
            e = np.mean(np.abs(y - p_y_given_x(x, w, b)))
            yield i, w, b, e
```

実行。

```
# w, b の初期値を作成
w, b = np.zeros(2), 0
gen = msgd(x, y, w, b)

# 読み方/補足は 確率的勾配降下法と同じ。
gen.next()
# (1, array([ 0.011 ,   0.0725]), 0.050000000000000003, 0.52633544830099133)

gen.next()
# (1, array([ 0.02251519,   0.13792904]), 0.096115503575706834, 0.54788728897754557)
```

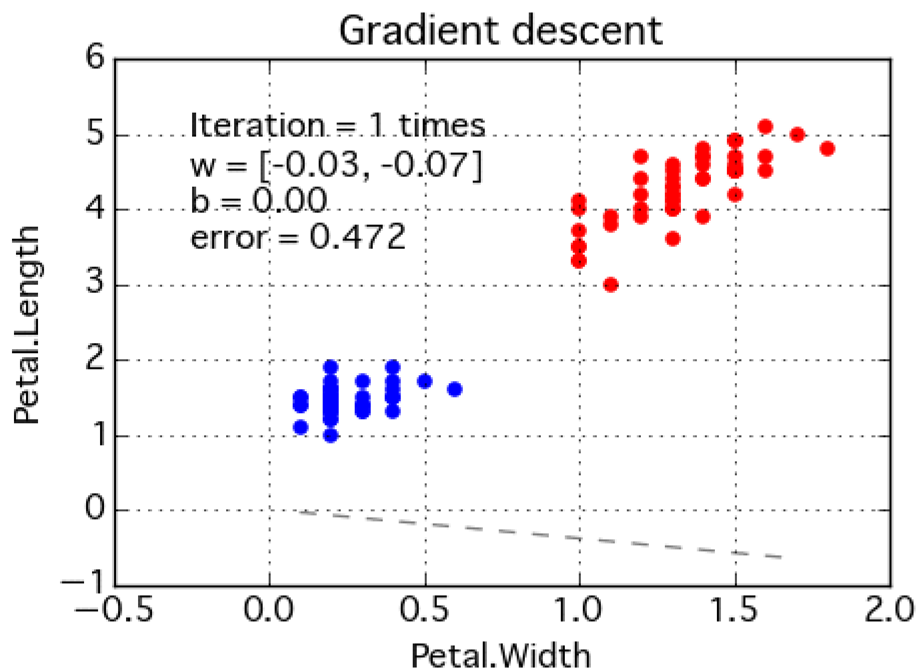
可視化

以上 3 つの処理を見比べてわかるとおり、差異は勾配を計算するのに利用するデータの切り出し処理のみ。それぞれどんな動きをしているのか可視化してみる。

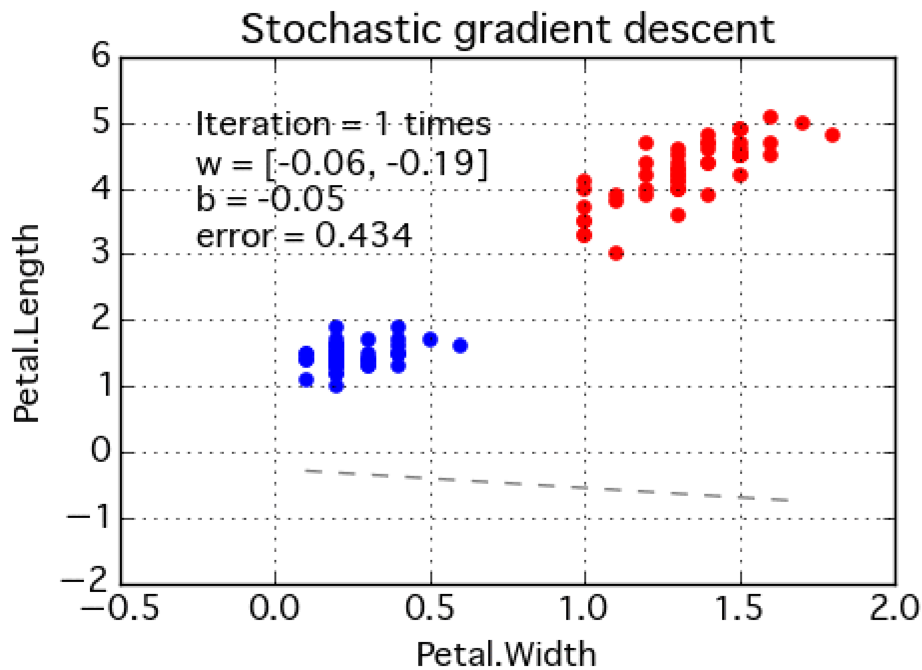
```
def plot_logreg(x, y, fitter, title):
    # 描画領域を作成
    fig = plt.figure(figsize=FIGSIZE)

    # 描画領域に Axes を追加、マージン調整
    ax = fig.add_subplot(1, 1, 1)
    fig.subplots_adjust(bottom=0.15)

    # 描画オブジェクト保存用
    objs = []

    # 回帰直線描画用の x 座標
    bx = np.arange(x.iloc[:, 0].min(), x.iloc[:, 0].max(), 0.1)

    # w, b の初期値を作成
    w, b = np.zeros(2), 0
    # 勾配法の関数からジェネレータを生成
    gen = fitter(x, y, w, b)
    # ジェネレータを実行し、勾配法 1 ステップごとの結果を得る
    for i, w, b, e in gen:
        # 回帰直線の y 座標を計算
        by = -b/w[1] - w[0]/w[1]*bx
        # 回帰直線を生成
        l = ax.plot(bx, by, color='gray', linestyle='dashed')
```

```
            # 描画するテキストを生成
            wt = """Iteration = {0} times
w = [{1[0]:.2f}, {1[1]:.2f}]
b = {2:.2f}
error = {3:.3}""".format(i, w, b, e)
            # axes 上の相対座標 (0.1, 0.9) に text の上部を合わせて描画
            t = ax.text(0.1, 0.9, wt, va='top', transform=ax.transAxes)
            # 描画した line, text をアニメーション用の配列に入れる
            objs.append(tuple(l) + (t, ))

        # データ, 表題を描画
        ax = plot_x_by_y(x, y, colors=['red', 'blue'], ax=ax)
        ax.set_title(title)
        # アニメーション開始
        ani = animation.ArtistAnimation(fig, objs, interval=1, repeat=False)
        plt.show()

plot_logreg(x, y, gd, title='Gradient descent')
plot_logreg(x, y, sgd, title='Stochastic gradient descent')
plot_logreg(x, y, msgd, title='Minibatch SGD')
```

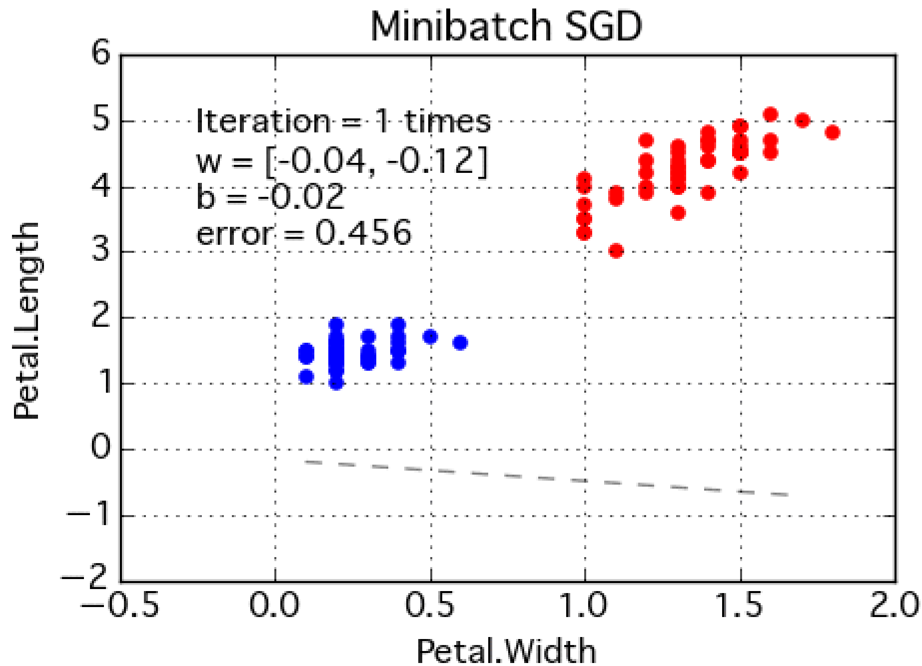
勾配降下法（Gradient Descent）
勾配降下法では 全データから勾配をもとめるので、イテレーション時に損失が増えることはない。ただしデータ数が多くなると使えない。



確率的勾配降下法（Stochastic Gradient Descent / SGD）
確率的勾配降下法では 一行ずつで勾配をもとめるので、処理される行によっては損失が増えることもある。が、全体を通じて徐々に損失は減っていく。



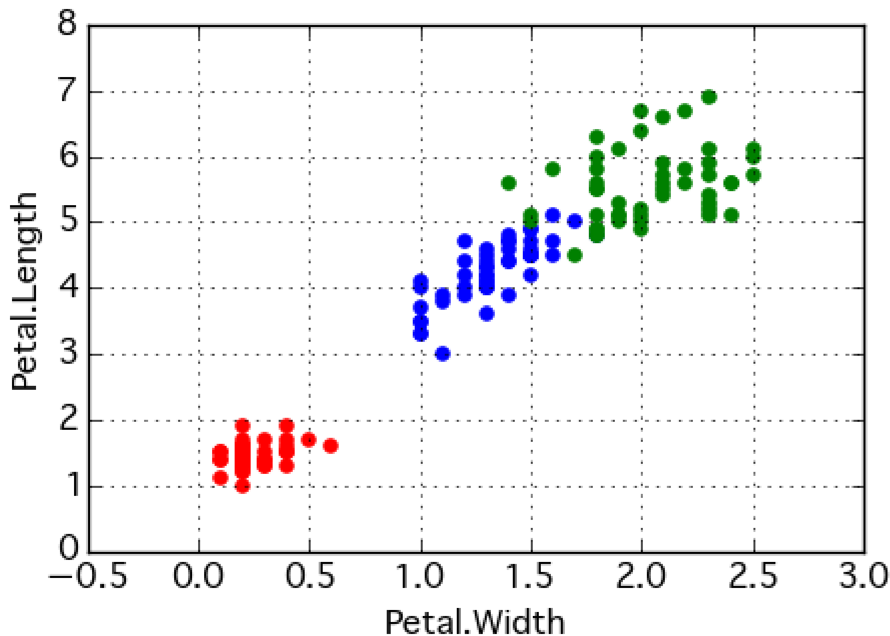
ミニバッチ確率的勾配降下法（Minibatch SGD / MSGD）
ミニバッチ確率的勾配降下法では ある程度の固まりから勾配をもとめるので、確率的勾配降下法と比べると 更新処理ごとの損失のばらつきは小さい。



多項ロジスティック回帰（多クラス）
続けて、目的変数のクラス数が 2 より大きい場合 = 多項ロジスティック回帰を行う。データは同じく iris を使う。説明変数は 2 次元のままとする。

```
data = iris

x = data[columns]
y = data['Species']
```

線形分離できないクラスもあるが、まあやってみよう。



多項ロジスティック回帰の場合は y の次元が変わる。伴って、予測値を求める式は
$\hat{y} = \pi(xw+b)$ となり関連する変数の次元も以下のように変わる。

- $\hat{y}$：真のラベル y の予測値ベクトル。次元は（入力データ数, クラス数）
- π：ソフトマックス関数。$xw+b$ の計算結果が各クラスに所属する確率を計算する。詳細は上k(略)
- x: 入力データ。次元は（入力データ数, 説明変数の数）
- w：係数行列。次元は（クラス数, 説明変数の数）
- b：バイアスベクトル。次元は（クラス数, 1）

上記のとおり y を多項ロジスティック回帰の形式に変換する。

```
# ラベルを カテゴリに対応した 0, 1 からなる 3 列に変換
y = pd.DataFrame({'setosa': (y == 'setosa').astype(int),
                  'versicolor': (y == 'versicolor').astype(int),
                  'virginica': (y == 'virginica').astype(int)})

y.head()
#      setosa   versicolor   virginica
# 1         1            0           0
# 2         1            0           0
# 3         1            0           0
# 4         1            0           0
# 5         1            0           0
```

続けて、多項ロジスティック回帰で使う関数を定義する。

```
def p_y_given_x(x, w, b):

    def softmax(x):
        return (np.exp(x).T / np.sum(np.exp(x), axis=1)).T

    return softmax(np.dot(x, w.T) + b)

def grad(x, y, w, b):
    error = p_y_given_x(x, w, b) - y
    w_grad = np.zeros_like(w)
    b_grad = np.zeros_like(b)

    for j in range(w.shape[0]):
        w_grad[j] = (error.iloc[:, j] * x.T).mean(axis=1)
        b_grad[j] = error.iloc[:, j].mean()

    return w_grad, b_grad, np.mean(np.abs(error), axis=0)
```

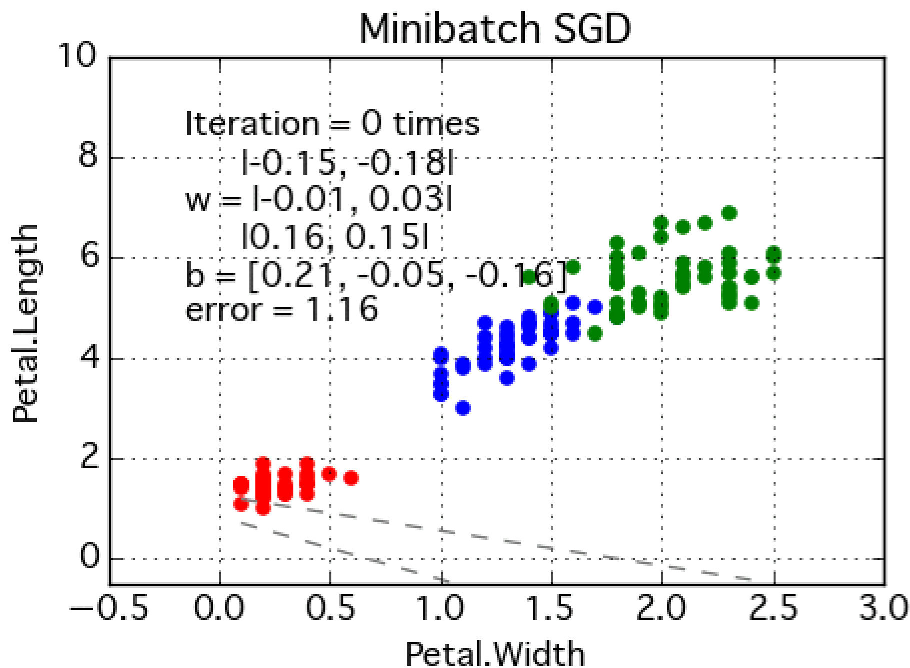
ミニバッチ確率的勾配降下法（Minibatch SGD / MSGD）

勾配法については考え方は一緒なので、MSGD のみ。ほかの手法の場合は ループ箇所とデータ切り出し処理を "2 クラスのロジスティック回帰" の場合と同様に書き換えればよい。

```
def msgd2(x, y, w, b, eta=0.1, num=800, batch_size=10):
    for i in range(num):
        for index in range(0, x.shape[0], batch_size):
            # index 番目のバッチを切り出して処理
            _x = x[index:index + batch_size]
            _y = y[index:index + batch_size]
            w_grad, b_grad, error = grad(_x, _y, w, b)
            w -= eta * w_grad
            b -= eta * b_grad
            e = np.sum(np.mean(np.abs(y - p_y_given_x(x, w, b))))
            yield i, w, b, e
```

可視化

可視化部分も似たようなコードなので、全体は gist で。各クラスを分割する位置に少しずつ回帰直線が寄っていく。



補足 行列 w の出力がずれていて済まぬ、、済まぬ、、。 tex 記法がうまく動かなかったんだ、、。

まとめ

ロジスティック回帰、とくに勾配法についてまとめた。目的変数のクラス数によってデータの次元 / 予測値算出のロジックは若干異なる。が、おおまかな流れは一緒。

5 ロジスティック回帰＋確率的勾配降下法

2011-07-10 ロジスティック回帰＋確率的勾配降下法

ロジスティック回帰＋確率的勾配降下法

PRML, 機械学習, ロジスティック回帰, R

次やってみたいことにロジスティック回帰を使おうとしているので、PRML 4 章に従ってさらっと実装してみる。

最終的には Python + numpy で実装し直すことになると思うけど、R の手触り感が好きなので、今回は R。
データセットには R なら簡単に扱える iris を使う。iris は 3 クラスあるので、多クラスロジスティック回帰（PRML 4.3.4）を実装することになる。
推論は IRLS (PRML 4.3.3) が PRML 的にはあうんだろうけど、ちょっと考えがあって確率的勾配降下法（PRML 3.1.3, 5.2.4）を使うことにする。

まずは、こちらで説明した方法で iris のデータを正規化&行列化して使いやすくしておく。

```
xlist <- scale(iris[1:4])
tlist <- cbind(
        ifelse(iris[5]=="setosa",1,0),
        ifelse(iris[5]=="versicolor",1,0),
        ifelse(iris[5]=="virginica",1,0)
```

```
)
N <- nrow(xlist) # データ件数
K <- ncol(tlist) # クラス数
```

次は基底関数を決める。手始めに 1 次項＋バイアスとしてみよう。
下のように書けば、phi のように定義した特徴関数を xlist の各行に適用して、N×M のいわゆる計画行列 PHI を生成することができる。

```
phi <- function(x) c(1, x[1], x[2], x[3], x[4])
PHI <- t(apply(xlist, 1, phi))   # NxM - design matrix
M <- ncol(PHI)   # 特徴数(特徴空間の次元)
```

パラメータとなる重み w を乱数で初期化しよう。
w は M×K の行列なので、正規乱数を使って初期化するなら次のように書けばいい。

```
w <- matrix(rnorm(M * K), M)
```

これで準備が整ったので、ロジスティック回帰を実装できる。
一番重要なのが、ロジスティック回帰の事後確率の仮定。多クラスなので、ソフトマックス関数の形をしている。

- $$p(C_k|\phi) = y_k(\phi) = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$ (PRML (4.104) 式)

ただし $a_k = w^T \phi_k$ である。
これは k=1, ..., K と動くから y がベクトルになることに気をつけると、次のように書ける。

```
# y_k = p(t=k | phi, w)   (k=1, ..., K)
y <- function(phi, w) {
        y <- exp(c(phi %*% w))
        return(y / sum(y))
}

# 例： n 番目の特徴ベクトルについて、その事後確率を求める
y_n <- y(PHI[n, ], w)
```

だいたいはこれで大丈夫なのだが、特徴を増やしたりするとうまくいかなくなったりし始める。
exp の結果が浮動小数点の範囲をオーバーフローして Inf に落ちてしまう現象が起きうる。具体的には exp の中身が 700 を超えるあたりから Inf になってしまうので、ちょっと大きめの特徴量があるだけで結構簡単に発生する。やっかい。
ロジスティック回帰に限らず exp を使うアルゴリズムで、学習後のパラメータがなぜか NaN だらけ……！？　という現象が発生したら、まずは exp のオーバーフローを疑うといい。

ソフトマックス関数は正規化のために exp の総和で割り算するので、a_k たちに定数を足したり引いたりしても結果は変わらない。だから適当な数を全体から引いて、オーバーフローしない範囲に収めてしまえばこの問題を解決することができる。
しかし定数の加減ではまだオーバーフローする可能性は残ってしまう。というわけで、一番簡単には最大値を引いてしまえばいい。
というわけで、以下がその対応版。

```
y <- function(phi, w) {
	y <- c(phi %*% w)
	y <- exp(y - max(y))  # exp の中身から、その最大値を引く(オーバーフロー対策)
	return(y / sum(y))
}
```

次は誤差関数とその勾配を求める。
ロジスティック回帰の誤差関数は交差エントロピー誤差関数であり、これはおなじみの負の対数尤度と同等。

- $$E(\boldsymbol{w}_1,\cdots,\boldsymbol{w}_K)=-\ln p(\boldsymbol{T}|\boldsymbol{w}_1,\cdots,\boldsymbol{w}_K)=-\sum_{n=1}^{N}\sum_{k=1}^{K}t_{nk}\ln y_{nk}$$ (PRML (4.108) 式)

これを微分した勾配は次のようになる。

- $$\nabla_{\boldsymbol{w}_j}E(\boldsymbol{w}_1,\cdots,\boldsymbol{w}_K)=\sum_{n=1}^{N}(y_{nj}-t_{nj})\phi_n$$ (PRML (4.109) 式)

どちらの式にも y がでてくるが、これは上で実装した関数が使えるので、それぞれ次のように簡潔に書ける。
ただし n についての和は関数の外でとるようにしている(確率的勾配降下法にあわせて)。

```
En <- function(phi, t, w) -log(sum(y(phi, w) * t))
dEn <- function(phi, t, w) outer(phi, y(phi, w) - t)
```

R ではベクトルや行列に対して一度に演算ができる関係から、このように簡潔に書ける反面、元となる数式と項の順番を変えないといけなかったり、outer*1 とかにうまく置き換える必要があったりする。
このあたりはいきなりやれと言われても出来ないので、最初のうちは要素ごとに考えて、少し慣れてきてからでいいと思う。

今回学習に用いる「確率的勾配降下法」は次のようにパラメータ w を更新していくことで学習する手法。

- $$\boldsymbol{w}^{(\tau+1)}=\boldsymbol{w}^{(\tau)}-\eta\nabla E_n$$ (PRML (3.22) 式)

ここで η は「学習率」、E_n は特徴ベクトル φ_n に対する勾配。
学習率 η は、適当な値(0.1 とか 0.01 とか)からはじめて、徐々に小さくしていく。
この更新式を全てのデータ点について回していく。順序はシャッフルすることが望ましい。
というわけでデータ点を一周回して更新するのは次のような実装になる。

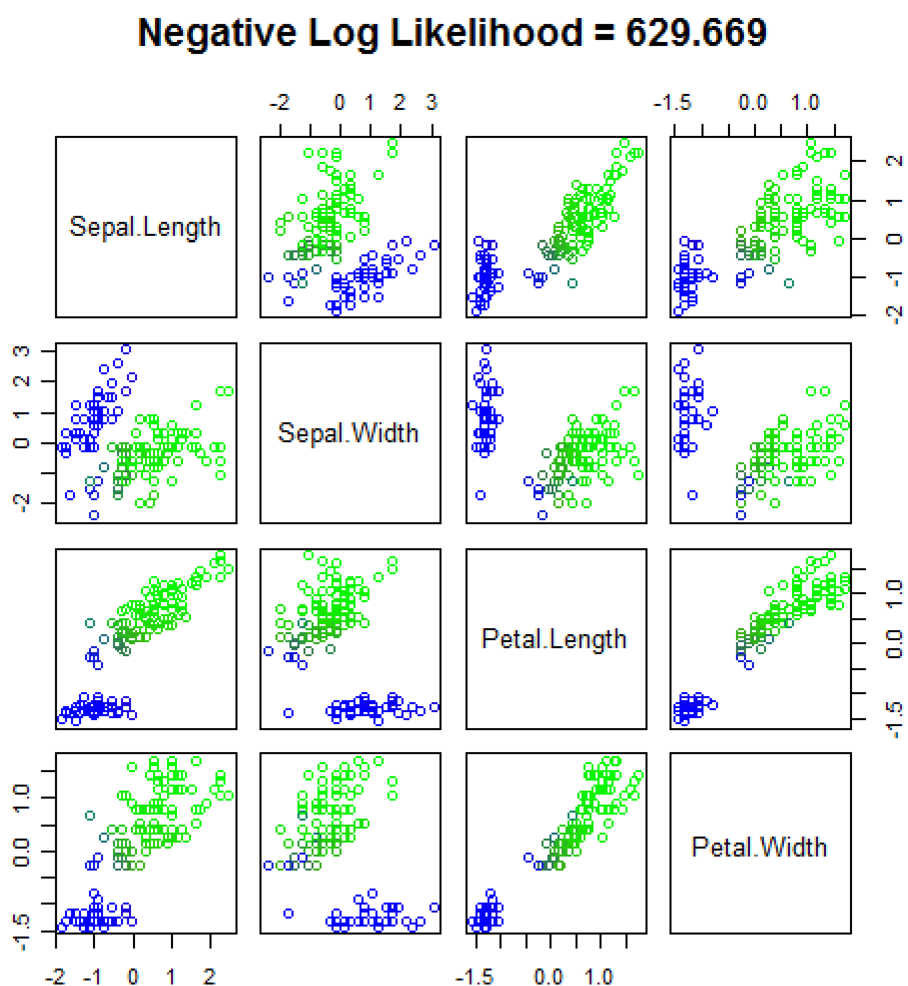
```
eta <- 0.1  # 学習率
for(n in sample(N)) {
	w <- w - eta * dEn(PHI[n,], tlist[n,], w)  # 確率的勾配降下法
}
```

勾配さえ計算できてしまえば、確率的勾配降下法はこのようにめちゃめちゃ簡単に実装できて実に嬉しい。が、簡単すぎて、こんなので学習とか本当にできるの？　と少し不安になる。
そういう場合は可視化してみればいい。

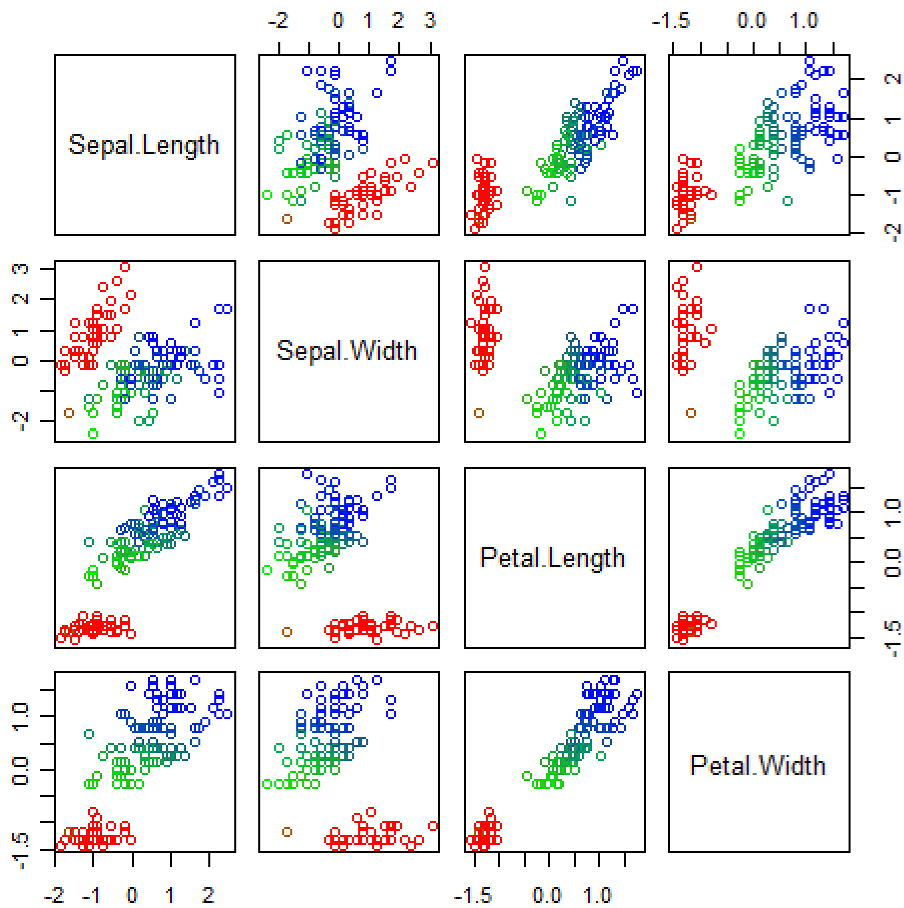
```
ylist <- t(apply(PHI, 1, function(phi) y(phi, w)))
error <- sum(sapply(1:N, function(n) En(PHI[n,], tlist[n,], w)))
```

```
pairs(xlist, col=rgb(ylist), main=sprintf("Negative Log Likelihood = %.3f", error))
```

このコードで、試しに「初期化しただけの、全く学習させていない w (つまり乱数)」を使って分布図を描くと次のようになった。



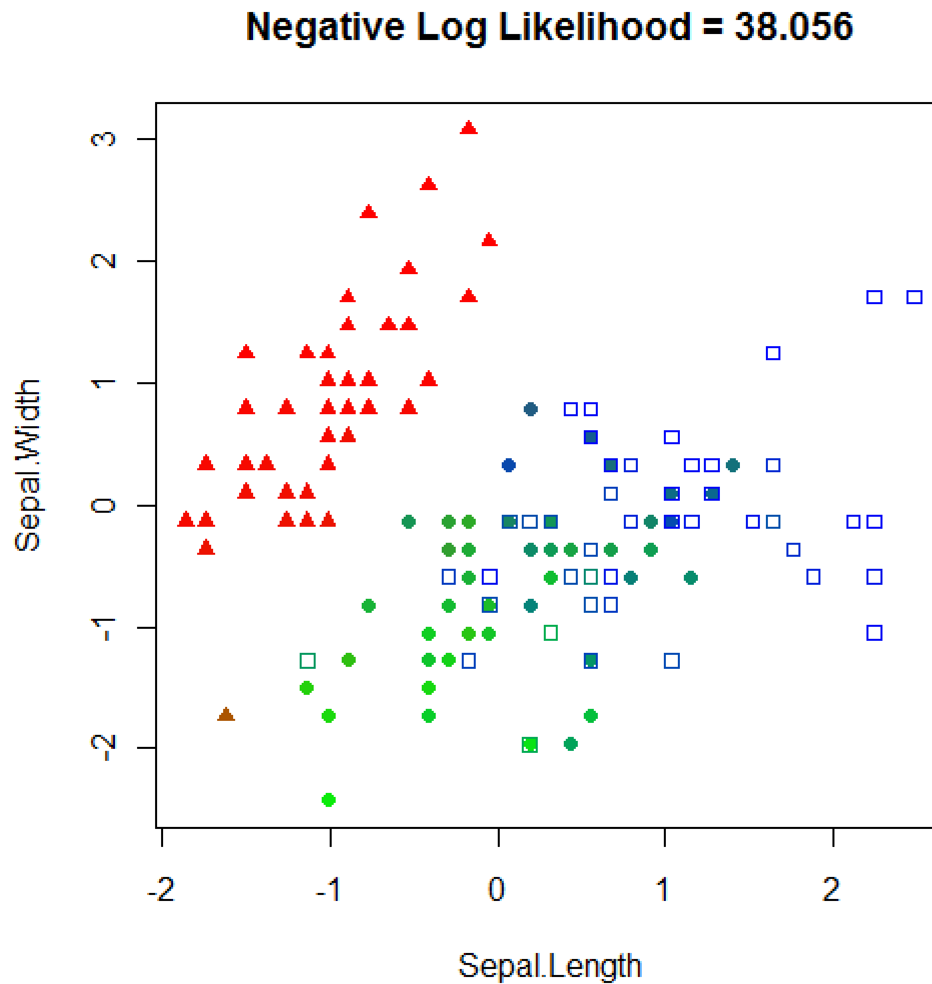
次に、学習率 $\eta=0.1$ でデータ点を一周だけ学習させたパラメータ w を使って描いた。



1 周回しただけで、なかなかいい感じに分類が行われていることがわかる。誤差(負の対数尤度)も大きく下がっている。

いや、分類できてるっぽく見えるけど、正しく分類できているかどうかはこの分布図を見ただけではわからない。
そこで、正解(真のラベル)がちゃんとわかるように、「真に setosa」である点は▲という形状に、一方「setosa と予測」した点は赤でプロット、同様に「versicolor は ●-緑」、「virginica は、□-青」と指定しつつ、もう少し細かいところを見られるように 2 軸だけ取り出して可視化してみた。

```
plot(xlist[,c(1,2)],
        col=rgb(ylist),
        pch=(tlist %*% c(17,16,22)),
        main=sprintf("Negative Log Likelihood = %.3f", error)
)
```



いくつか「緑っぽい□」と「青っぽい●」は見受けられるものの、正しく分類が進んでいる。
というわけで、ロジスティック回帰に確率的勾配降下法を使えば、「こんなに簡単でいいの？」という実装でちゃんと学習できることがわかる。
もうちょっと学習を進めたらどうなるか、特徴関数をもう少し増やしたらどうなるか、というあたりはまた次回。

６　線形ＳＶＭ

線形 SVM ～ 数式の説明 ～

前ページで線形 SVM のコーディングに必要な式を紹介したけれど，それだけでは納得できん，中身までちゃんと教えてくれ，という向きには，このページでちゃんと説明する．

前ページの再掲になるけれど，線形識別関数を次のように定義する．

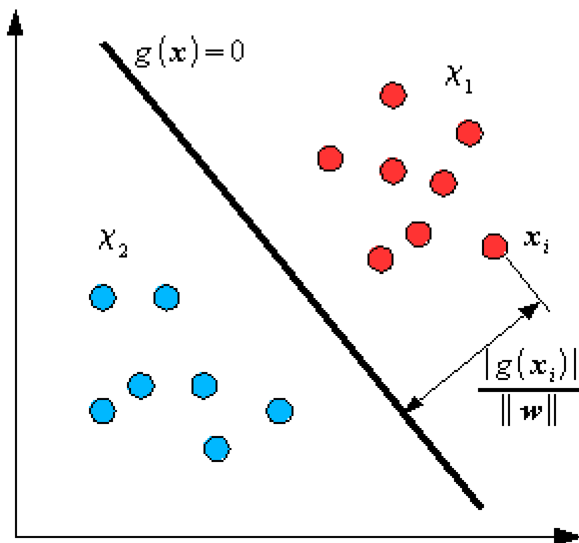
$f(\boldsymbol{x}) = sign(g(\boldsymbol{x}))$ ただし $g(\boldsymbol{x}) = \boldsymbol{w}^t \boldsymbol{x} + b$ ． (1.1)
$\boldsymbol{x}$ は入力ベクトル，ベクトル $\boldsymbol{w}$ およびスカラー b は識別関数を決定するパラメータ．

学習データは n 個与えられているとし， $\boldsymbol{x}_i (i = 1, 2, \ldots, n)$ と表す．これらのデータを 2 つのクラス X_1 および X_2 に分離することを考える．この学習データ集合に対して， $g(\boldsymbol{x})$ が次の条件を満たすようにパラメータを調節することを考える．

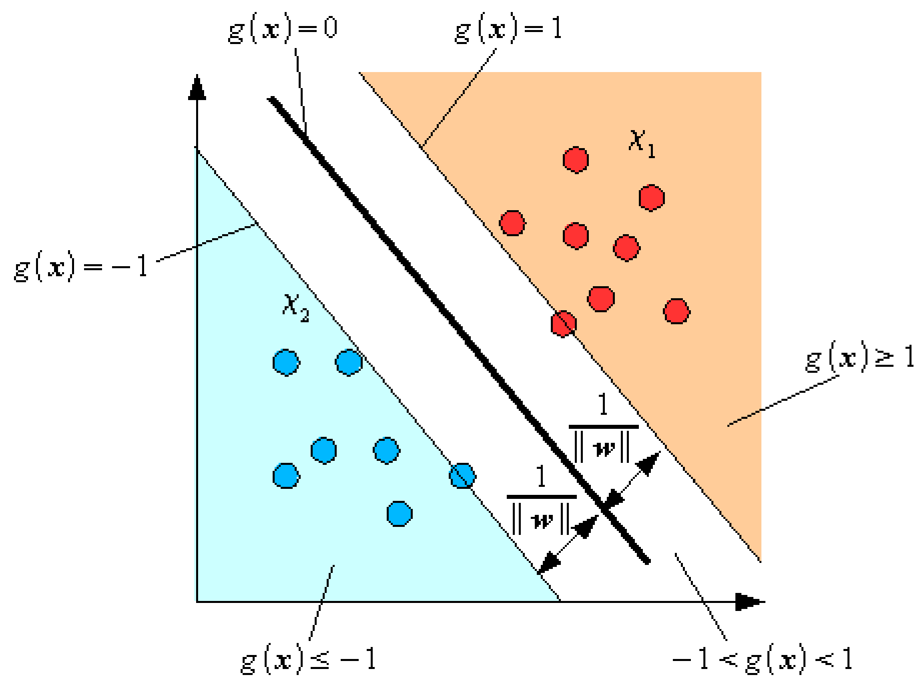
$g(\boldsymbol{x}_i) = \boldsymbol{w}^t \boldsymbol{x}_i + b$
$\geq 1 \ if \ \boldsymbol{x}_i \in X_1$
$\leq -1 \ if \ \boldsymbol{x}_i \in X_2$ (1.2)

（OpenOffice の数式ツールで左括弧ってどうやって出すの？）

ところで，点 $\boldsymbol{x}_i$ から分離境界 $g(\boldsymbol{x}) = 0$ との距離は $|g(\boldsymbol{x}_i)| / \|\boldsymbol{w}\|$ となる（どうしてそうなるかって？それは自分で考えてみてね）．



ということは，(1.2)式を満たす識別関数において，学習データは分離境界から距離 $1/\|\boldsymbol{w}\|$ の領域には存在しないことを意味する．



さて，境界が最も適した状態とはどんな場合かを考える．常識的に考えて，両クラスの距離（マージン）が広くなるような識別関数が最も汎化能力の高い識別関数と考えることができる．つまり，式(1.2)の条件の下でマージン $2/\|\boldsymbol{w}\|=1/\|\boldsymbol{w}\|+1/\|\boldsymbol{w}\|$ を最大にするようなパラメータ $\boldsymbol{w}$ および b を考えればよい．（**マージン最大化**）

ここで学習データ $\boldsymbol{x}_i$ に関する教師信号を y_i とし，次のように定義する．

$$y_i = \begin{matrix} 1 & \mathit{if}\ \boldsymbol{x}_i \in X_1 \\ -1 & \mathit{if}\ \boldsymbol{x}_i \in X_2 \end{matrix} \quad (1.3)$$

(だからね，左括弧だけ出すのってどうするの？)

これを使うと，式(1.2)を次のように場合わけせずに書きなおせる．

$$y_i(\boldsymbol{w}^t\boldsymbol{x}_i+b)-1\geq 0 \quad (1.4)$$

この条件の下でマージン $2/\|\boldsymbol{w}\|$ を最大化する問題を考えることになる．式の扱い上，$2/\|\boldsymbol{w}\|$ のままだと後々面倒なことになるので，

$$G(\boldsymbol{w})=\frac{1}{2}\|\boldsymbol{w}\|^2 \quad (1.5)$$

の最小化問題と置き換える．

さて，この最小化問題をそのまま解くのは大変なので，ここはラグランジュの未定乗数法を使って問題を書き直すのですよ．

$\boldsymbol{x}_i$ に対応するラグランジュ未定乗数 $\lambda_i(\lambda_i \geq 0, i = 1,2,\ldots,n)$ を要素とするベクトル $\boldsymbol{\lambda}$ を定義すると，ラグランジュ関数 L_p は次のようになる．

$$L_p(\boldsymbol{w}, b, \boldsymbol{\lambda}) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{i=1}^{n} \lambda_i [y_i(\boldsymbol{w}^t \boldsymbol{x}_i + b) - 1] \tag{1.6}$$

これを $\boldsymbol{w}$ および b で編微分して 0 とおくと，

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{n} \lambda_i y_i \boldsymbol{x}_i = 0 \tag{1.7}$$

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{n} \lambda_i y_i = 0 \tag{1.8}$$

ここで式(1.7)より， $\boldsymbol{\lambda}$ が求まれば $\boldsymbol{w}$ が求まることがわかる．

$$\boldsymbol{w} = \sum_{i=1}^{n} \lambda_i y_i \boldsymbol{x}_i \tag{1.9}$$

では, どうやって $\boldsymbol{\lambda}$ を計算するのかというと, 次のようにまた問題を置き換える操作をするのです. まず式(1.7)のラグランジュ関数 L_p を $F(\boldsymbol{\lambda})$ と書き換えて，式(1.7)および(1.9)を代入する．

$$\begin{aligned} F(\boldsymbol{\lambda}) &= \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{i=1}^{n} \lambda_i [y_i(\boldsymbol{w}^t \boldsymbol{x}_i + b) - 1] \\ &= \frac{1}{2}\|\boldsymbol{w}\|^2 - \boldsymbol{w}^t \sum_{i=1}^{n} \lambda_i y_i \boldsymbol{x}_i - b\sum_{i=1}^{n} \lambda_i y_i + \sum_{i=1}^{n} \lambda_i \\ &= \frac{1}{2}\|\boldsymbol{w}\|^2 - \boldsymbol{w}^t \boldsymbol{w} - 0 + \sum_{i=1}^{n} \lambda_i \\ &= \sum_{i=1}^{n} \lambda_i - \frac{1}{2}\|\boldsymbol{w}\|^2 \end{aligned} \tag{1.10}$$

これを λ_i それぞれについて分解すると次のようになる．

$$F(\lambda_i) = \lambda_i - \frac{1}{2}\sum_{i,j=1}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^t \boldsymbol{x}_j \tag{1.11}$$

結局，最適な $\boldsymbol{\lambda}$ を求めることは，「関数 $F(\lambda_i)$ を $\sum_{i=1}^{n} \lambda_i y_i = 0$ （式(1.8)）および $\lambda_i \geq 0$ の条件下で最大化する問題」と置き換えることができる．

ここで式(1.9)を見ると，$\lambda_i = 0$ となるような学習データ $\boldsymbol{x}_i$ はパラメータ $\boldsymbol{w}$ の決定には一切関与していないことがわかる．つまり $\lambda_i > 0$ となる学習データ $\boldsymbol{x}_i$ のみで識別関数が決定されるわけで，このような「識別関数の決定に関与するデータ(ベクトル)」を「**サポートベクトル**」(Support Vector, SV) と呼ぶ.
$F(\lambda_i)$ の最大化がなされると，サポートベクトルは互いのクラスの境界に最も近いデータのみが選択されようになる．これにより，式(1.2)にサポートベクトル $\boldsymbol{x}_S$ を代入することでパラメータ b が求まる．
$b = y_S - \boldsymbol{w}^t \boldsymbol{x}_S$ (1.12)

さて，ここまではいろんな解説記事に書いてあるんだけど，実際に $F(\lambda_i)$ を最大化する方法が書いてない記事ばっかり．で，これではコーディングできないので，とにかく最急降下法でこれを解く方法を考えてみる．

$F(\lambda_i)$ に関する最急降下法は次のように表される．

$$\lambda_i(t+1) = \lambda_i(t) + \eta \frac{\partial F(\lambda_i)}{\partial \lambda_i} \quad (1.13)$$

式(11)を λ_i で編微分すると，

$$\frac{\partial F(\lambda_i)}{\partial \lambda_i} = 1 - \sum_{j=1}^{n} \lambda_j y_i y_j \boldsymbol{x}_i^t \boldsymbol{x}_j \quad (1.13)$$

よって式(13)は次のようになる．

$$\lambda_i(t+1) = \lambda_i(t) + \eta \left(1 - \sum_{j=1}^{n} \lambda_j y_i y_j \boldsymbol{x}_i^t \boldsymbol{x}_j\right) \quad (1.14)$$

η はいわゆる学習係数で，小さな正の値とする．これにより，ヒューリスティックに最適な λ_i を算出することができる．

7　Anaconda を利用した Python のインストール

(Windows) (adsbygoogle = window.adsbygoogle || []).push({});

本サイトでは、Anaconda を利用して Python のインストールを行う手順を説明します。

Anaconda (アナコンダ) とは

Anaconda は、Continuum Analytics 社によって提供されている、Python 本体に加え、科学技術、数学、エンジニアリング、データ分析など、よく利用される Python パッケージ（2016 年 2 月時点で 400 以上）を一括でインストール可能にしたパッケージです。面倒なセットアップ作業が効率よく行えるため、Python 開発者の間で広く利用されています。なお、Anaconda は商用目的にも利用可能です。

Python のバージョン

2016 年 2 月現在、Python には、2.x 系のバージョンと 3.x 系のバージョン (現時点では、Python 3.5）が存在します。Python Wiki によると、”Python 2.x は遺産※で、Python 3.0 が Python 言語の現在と未来である。” と述べられています。

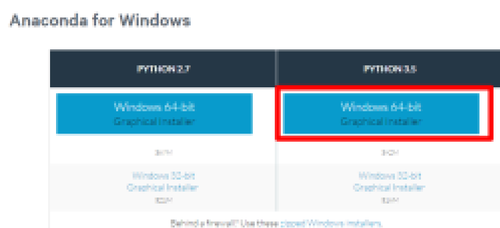
※ レガシー（遺産）とは、コンピュータ用語でいう過去のバージョンとの互換性などの問題から新しいバージョンに移行するのが困難になり、古いバージョンを使わざるを得なくなっている状況のことを指します。

ですので、新しく Python を始められる方には、Python 3.x 系を選択すればよいでしょう。本サイトでも、Python 3.x 系を中心に手順を解説します。

Anaconda を利用した Python のインストール

本例では、Windows 10 に Anaconda をインストールする手順を紹介します。

Anaconda のダウンロード

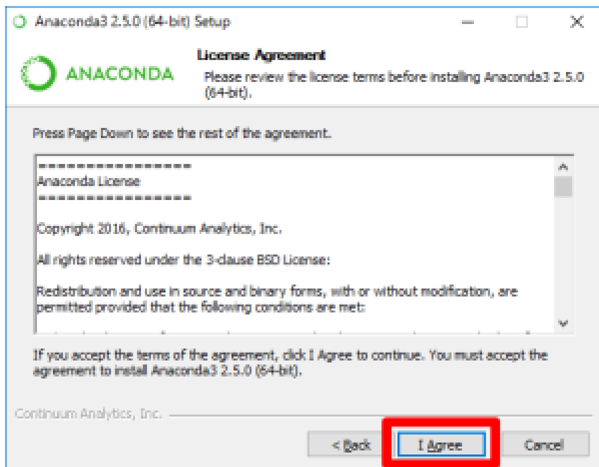


Anaconda のダウンロードページから、Python 3.5 の 64 bit のインストーラをダウンロードします。

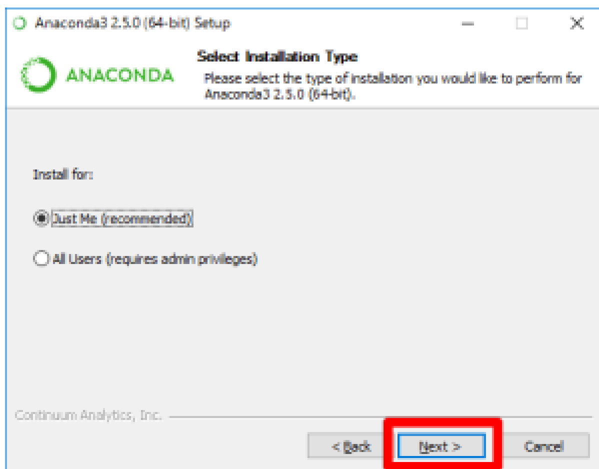
Anaconda のインストール
ダウンロードしたファイルを開き、インストーラを起動します。

「Next」を押して次に進みます。

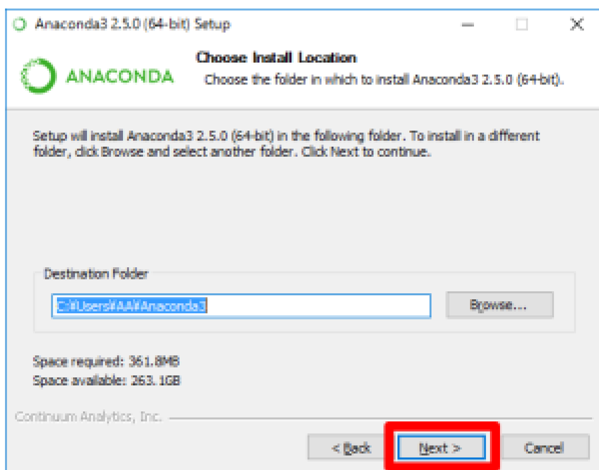
ライセンスを確認し、「I Agree」を押して次に進みます。



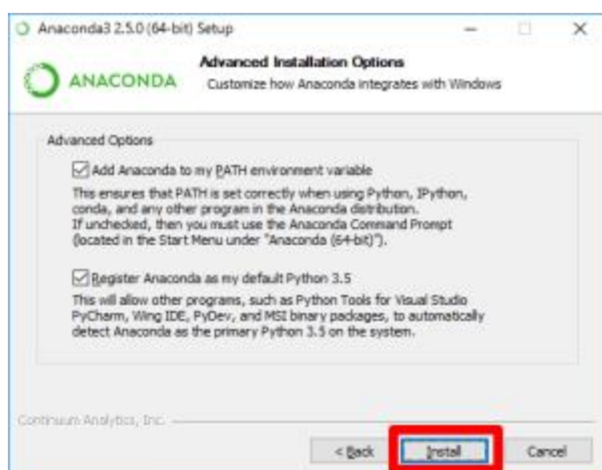
「Just Me」を選択されていることを確認し、「Next」を押して次に進みます。



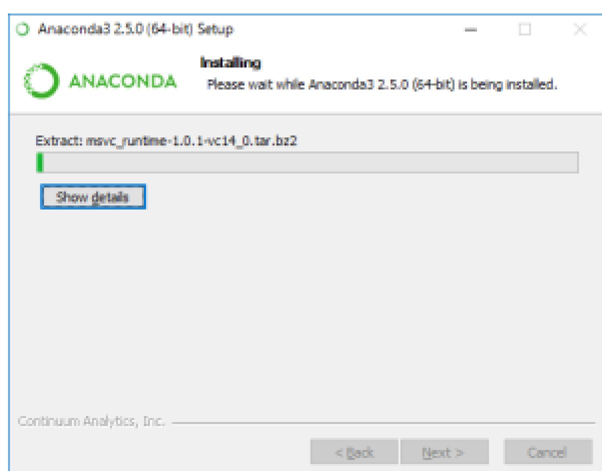
インストール先を尋ねられるので、「Next」を押して次に進みます。



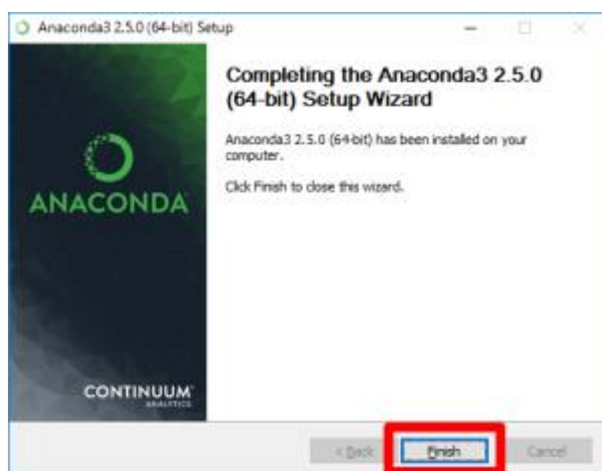
2 つのチェックボックス（環境変数 PATH への設定、Python 3.5 をデフォルトの Python として登録）にチェックが入っていることを確認し、「Next」を押して次に進みます。



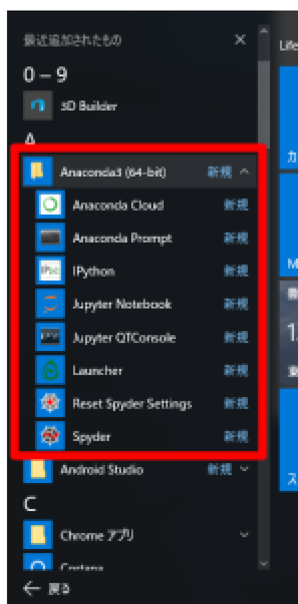
インストールが始まります。



「Finish」を押し、インストーラを閉じます。



スタートメニューに、Anaconda として、Python や関連ツールが一式格納されていることがわかります。



Anaconda を利用した Python のインストール方法は以上です。非常に簡単かつ便利なので、Python の開発環境を整備するには、Anaconda の利用がオススメです。

- See more at:

http://pythondatascience.plavox.info/python%E3%81%AE%E3%82%A4%E3%83%B3%E3%82%B9%E3%83%88%E3%83%BC%E3%83%AB/python%E3%81%AE%E3%82%A4%E3%83%B3%E3%82%B9%E3%83%88%E3%83%BC%E3%83%AB/python%E3%81%AE%E3%82%A4%E3%83%B3%E3%82%

８　PYTHON プログラム

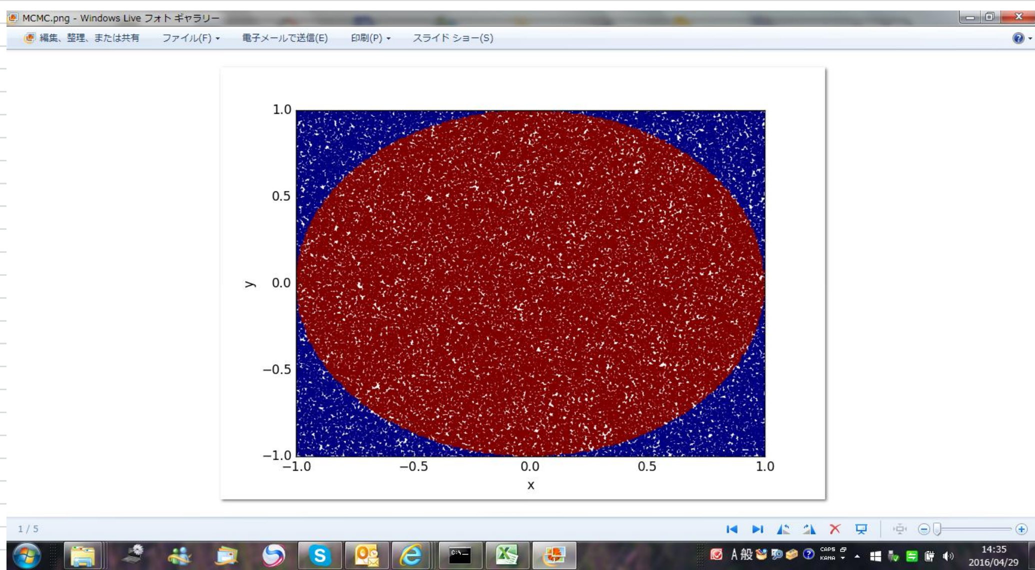
PYTHONプログラム

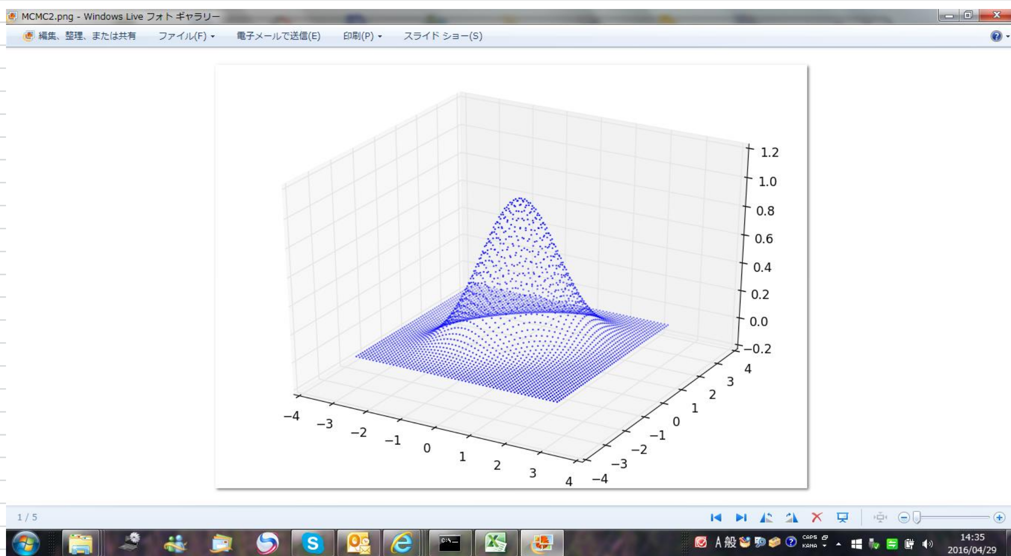
東京国際大学　渡辺信一

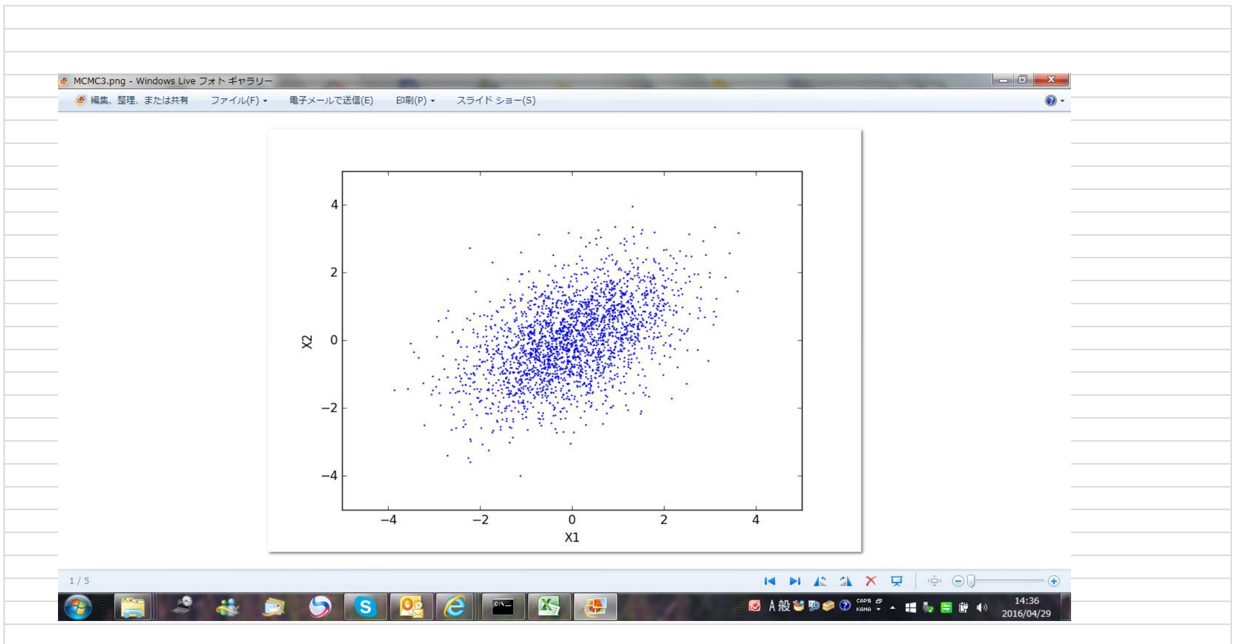
1	MCMC	MCMC（面積計算・3次元・ギブス・サンプリング）
2	BLM	ブラック＝リッターマン・モデル
3	GIBBS	ギブスサンプリング（2次元）
4	MH	メトロポリス・ヘイスティングス（3次元ギブス・サンプリング）
5	MCMC3	メトロポリス（2次元）
6	MH10	球体積
7	GAUSS	混合ガウス（メトロポリス）
8	GAUSS4	混合ガウス（パラレル・テンパリング）
9	MH21	メトロポリス法
10	MH22	ガウシアン混合モデル
11	CV	クロスバリデーション
12	BROWN	幾何ブラウン運動
13	MCMC25	MCMC（SUWA）

```
import numpy as np
import matplotlib.pyplot as plt

N = 100000

x = np.random.uniform(-1.0, 1.0, N)
y = np.random.uniform(-1.0, 1.0, N)

l = []

for i in xrange(N):
    if (x[i]**2 + y[i]**2) < 1.0:
        l.append(True)
    else:
        l.append(False)

print 2.0**2 * float(l.count(True)) / float(N)
# 3.13632

plt.scatter(x, y, c=l, s=5, edgecolor='None')
plt.xlim(-1.0, 1.0)
plt.ylim(-1.0, 1.0)
plt.xlabel('x')
plt.ylabel('y')
plt.show()



import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D

def range_ex(start, end, step):
    while start + step < end:
        yield start
        start += step

# P(x) : Target distribution
def P(x1, x2, b):
    return np.exp(-1/2 * (x1**2 - 2*b*x1*x2 + x2**2))

xs = []
ys = []
zs = []
b = 0.5

for i in range_ex(-3, 3, 0.1):
    for j in range_ex(-3, 3, 0.1):
        xs.append(i)
        ys.append(j)
        zs.append(P(i, j, b))

ax = Axes3D(plt.figure())
ax.scatter3D(xs, ys, zs, s=3, edgecolor='None')
plt.show()

def gibbs(N, thin):
    s = []
    x1 = 0.0
    x2 = 0.0
    for i in range(N):
        for j in range(thin):
            x1 = np.random.normal(b * x2, 1) # P(x1|x2)
            x2 = np.random.normal(b * x1, 1) # P(x2|x1)
        s.append((x1,x2))
    return np.array(s)

N = 3000
thin = 500
burn_in = 0.2
```

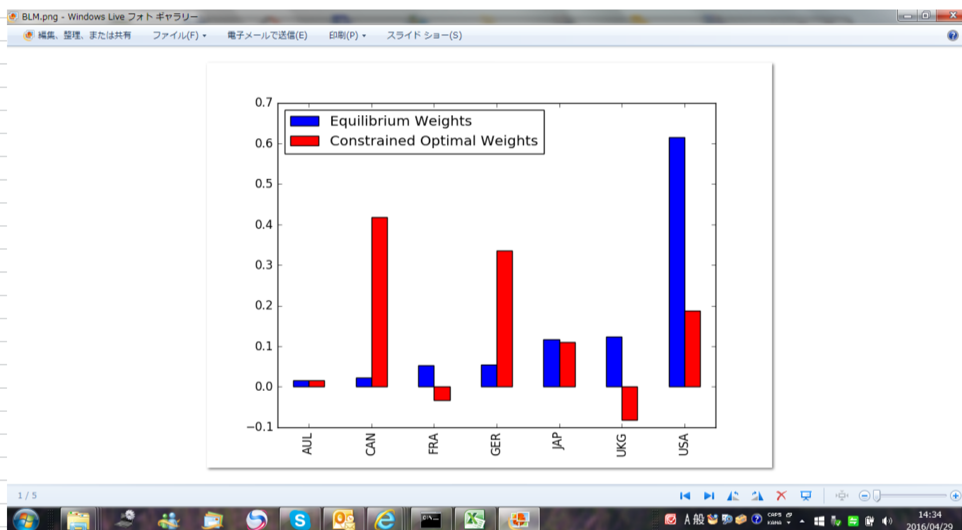
```
sample = gibbs(N, thin)

plt.scatter(
    sample[int(N * burn_in):,0],
    sample[int(N * burn_in):,1],
    s=3,
    edgecolor='None'
)
plt.xlabel('X1')
plt.ylabel('X2')
plt.show()
```



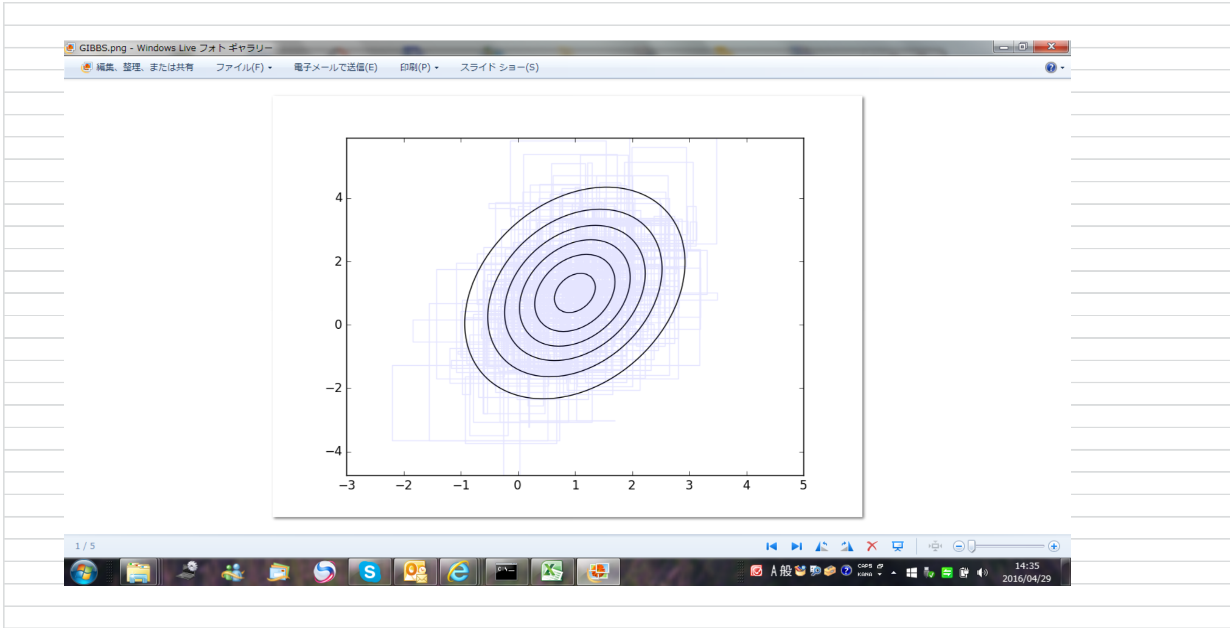



MCMC3.png - Windows Live フォト ギャラリー
編集、整理、または共有
ファイル(F)
電子メールで送信(E)
印刷(P)
スライド ショー(S)
4
2
0
−2
−4
X2
−4
−2
0
2
4
X1
1 / 5
14:36
2016/04/29

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# ウェイト
w = np.array([[0.016, 0.022, 0.052, 0.055, 0.116, 0.124, 0.615]]).T
# 相関行列
correlation = np.array([
        [1, 0.488, 0.478, 0.515, 0.439, 0.512, 0.491],
        [0.488, 1, 0.664, 0.655, 0.310, 0.608, 0.779],
        [0.478, 0.664, 1, 0.861, 0.355, 0.783, 0.668],
        [0.515, 0.655, 0.861, 1, 0.354, 0.777, 0.653],
        [0.439, 0.310, 0.355, 0.354, 1, 0.405, 0.306],
        [0.512, 0.608, 0.783, 0.777, 0.405, 1, 0.652],
        [0.491, 0.779, 0.668, 0.653, 0.306, 0.652, 1]])
# 標準偏差
std = np.array([[0.16, 0.203, 0.248, 0.271, 0.21, 0.2, 0.187]])
# 相関行列と標準偏差から共分散行列を計算
Sigma = correlation * np.dot(std.T, std)
# パラメータdelta  値はHe&Litterman(1999)に従う
delta = 2.5
# パラメータtau  値はHe&Litterman(1999)に従う
tau = 0.05
# 均衡リターンを求める(reverse optimization)
r_eq = delta * np.dot(Sigma, w)
P = np.array([
        [0,0,-0.295,1,0,-0.705,0],
        [0,1,0,0,0,0,-1]]) # 2x7 matrix (2: number of views, 7: number of assets)
Q = np.array([[0.05],[0.03]]) # 2-vector
Omega = np.array([
        [0.001065383332,0],
        [0,0.0008517381]])
# 均衡リターンに投資家のビューをブレンドする
r_posterior = r_eq + np.dot( np.dot( tau*np.dot(Sigma,P.T), np.linalg.inv(tau*np.dot(np.dot(P,Sigma),P.T)+Omega)), (Q-np.dot(P,r_eq)))
Sigma_posterior = Sigma + tau*Sigma - tau*np.dot( np.dot( np.dot(Sigma,P.T), np.linalg.inv(tau*np.dot(np.dot(P,Sigma),P.T)+Omega)), tau*np.dot(P,Sigma
# Forward Optimizationをして最適ウェイトを求める
w_posterior = np.dot(np.linalg.inv(delta*Sigma_posterior), r_posterior)
df = pd.DataFrame([w.reshape(7),w_posterior.reshape(7)],
                columns=['AUL','CAN','FRA','GER','JAP','UKG','USA'],
                index=['Equilibrium Weights','Constrained Optimal Weights'])
df.T.plot(kind='bar', color='br')
plt.show()
```

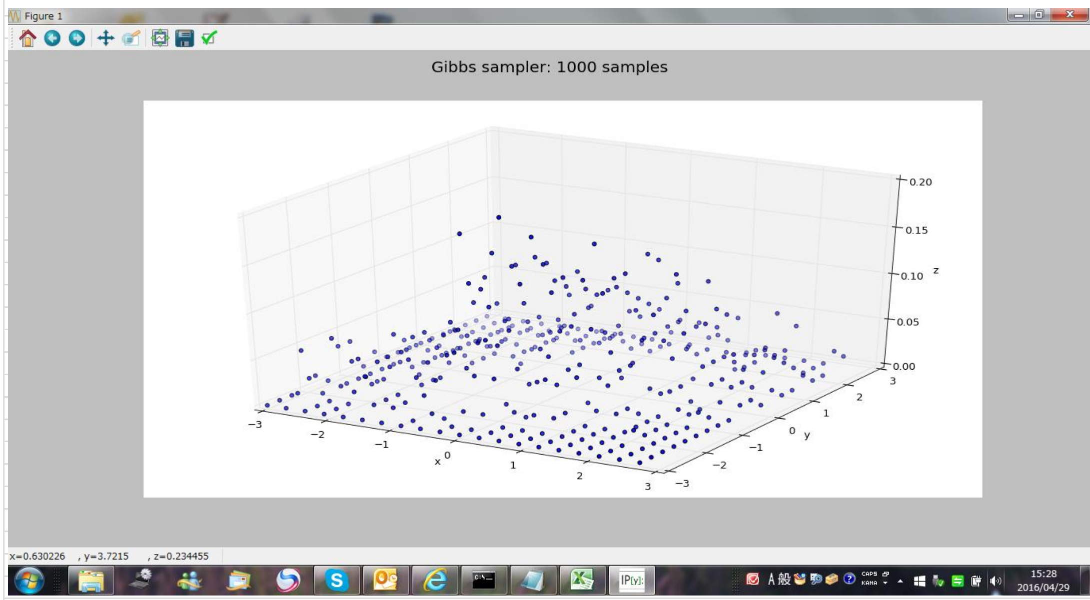


```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats

## 2dimention normal distribution
nu = np.ones((2))
covariance = np.array([[1,0.5],[0.5,3]])
# la:   固有値array
# v:    固有ベクトル array
la,v = np.linalg.eig(covariance)
avr_sigma = np.average(la)
#あとでプロット範囲を決めるときの指標に

def gibssampling(nu,cov,sample_size):
    """
    Gibbs sampling !
    @nu      :average vector
    @cov     :covariance matrix
    @sample_size     :size of sample
    return type      :numpy.array
    length           :sample_size
    """
    samples = []
    dim = len(nu)
    # start point of sampling
    start = [0,0]
    samples.append(start)
    search_dim = 0
    for i in range(sample_size):
        if search_dim == dim-1:
            """
            search dimension select is cyclic
            it can replace randomly
            """
            search_dim = 0
        else:
            search_dim = search_dim +1
        #new-sampling

        prev_sample = samples[-1][:] # previous sample
        A = cov[search_dim][search_dim-1] / float(cov[search_dim-1][search_dim-1])
# A*Σ_yy = Σ_xy
        _y = prev_sample[search_dim-1] # other dimention's previous values

        # p(x|y) ~ N(x|nu[x]+A(_y-nu[y]),Σ_zz)
        #  Σ_zz = Σ_xx - A0*Σ_yx

        mean = nu[search_dim] + A*(_y-nu[search_dim-1])
        sigma_zz = cov[search_dim][search_dim] -A*cov[search_dim-1][search_dim]

        sample_x = np.random.normal(loc=mean,scale=np.power(sigma_zz,.5),size=1)
        prev_sample[search_dim] = sample_x[0]

        samples.append(prev_sample)
    return np.array(samples)

sample = gibssampling(nu,covariance,1000)
plt.plot(sample[:,0],sample[:,1],alpha=.1)

#答え合わせ
malti_normal = stats.multivariate_normal(mean=nu,cov=covariance)
X,Y = np.meshgrid(np.linspace(nu[0]-avr_sigma*2,nu[0]+avr_sigma*2,100),
                  np.linspace(nu[1]-avr_sigma*2,nu[1]+avr_sigma*2,100))
Pos = np.empty(X.shape + (2,))
Pos[:,:,0]=X
Pos[:,:,1]=Y
Z=malti_normal.pdf(Pos)
plt.contour(X,Y,Z,colors="k")
plt.show()
```


GIBBS.png - Windows Live フォト ギャラリー
編集、整理、または共有
ファイル(F)
電子メールで送信(E)
印刷(P)
スライド ショー(S)
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
4
5
1 / 5
14:35
2016/04/29

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import random
import math

# 2D Gibbs Sampler
class Gibbs:
    def __init__(self, init_x, init_y):
        self.x         = init_x
        self.y         = init_y
        self.t         = 0

    def update(self):
        if(self.t == 0):
            self.x = Gibbs.rand_norm(0.5 * self.y, 1.0)
        else:
            self.y = Gibbs.rand_norm(0.5 * self.x, 1.0)
        self.t = (self.t + 1) % 2

    @staticmethod
    def rand_norm(mu, sigma):
        r1 = random.random()
        r2 = random.random()
        z = math.sqrt(-2.0 * math.log(r1)) * math.sin(2.0 * math.pi * r2)
        return mu + sigma * z

# 2D Histogram
class Hist2D:
    def __init__(self, minx, miny, maxx, maxy, nbins):
        self.minx = minx
        self.miny = miny
        self.maxx = maxx
        self.maxy = maxy
        self.nbins = nbins
        self.spanx = (maxx - minx) / nbins
        self.spany = (maxy - miny) / nbins
        self.bins  = [[0] * nbins for i in range(nbins)]

    def set_value(self, x, y):
        bx = int((x - self.minx) / self.spanx)
        by = int((y - self.miny) / self.spany)
        if bx >=0 and by >= 0 and bx < self.nbins and by < self.nbins:
            self.bins[bx][by] += 1

    def get(self, x, y):
        return self.bins[x][y]

# Gibbs sampler test
def gibbs_test(init_x, init_y, trial):
    gibbs = Gibbs(init_x, init_y)
    burn = int(trial / 10)
    for i in range(burn):
        gibbs.update()

    nbins = 20
    minx = -3.0
    miny = -3.0
    maxx = 3.0
    maxy = 3.0
    hist = Hist2D(minx, miny, maxx, maxy, nbins)
    for i in range(trial):
        hist.set_value(gibbs.x, gibbs.y)
        gibbs.update()

    xs = [0.0] * nbins * nbins
    ys = [0.0] * nbins * nbins
    zs = [0.0] * nbins * nbins
    fig = plt.figure()
    ax  = fig.add_subplot(111, projection='3d')
    for y in range(nbins):
        for x in range(nbins):
            i = y * nbins + x
            xs[i] = x * hist.spanx + hist.minx
            ys[i] = y * hist.spany + hist.miny
            zs[i] = hist.get(x, y) / (trial * hist.spanx * hist.spany)
```

```
    ax.scatter3D(xs, ys, zs)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.set_xlim3d(minx, maxx)
    ax.set_ylim3d(miny, maxy)
    ax.set_zlim3d(0.0, 0.2)
    plt.suptitle('Gibbs sampler: %d samples' % trial, size='18')
    plt.savefig('gibbs_%d.png' % trial)
    plt.show()

if __name__=='__main__':
    for t in [1000, 5000, 10000, 50000, 100000, 500000]:
        gibbs_test(0.0, 0.0, t)
```



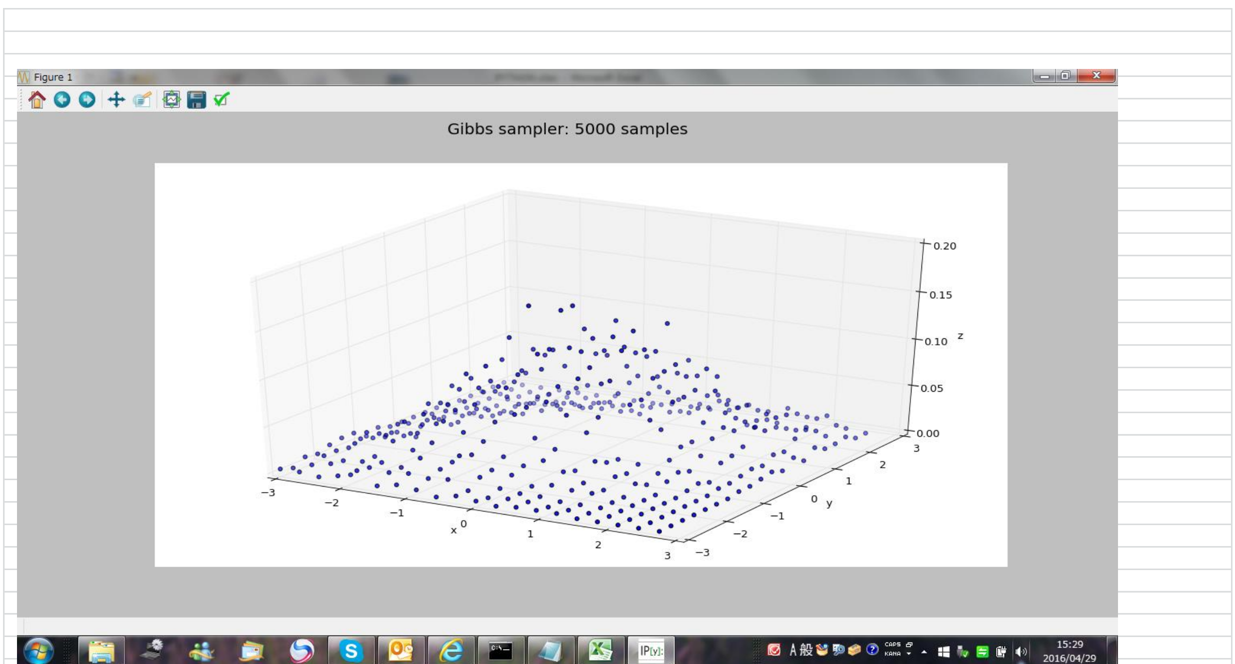


Figure 1
Gibbs sampler: 5000 samples
0.20
0.15
0.10
0.05
0.00
z
x
y
15:29
2016/04/29

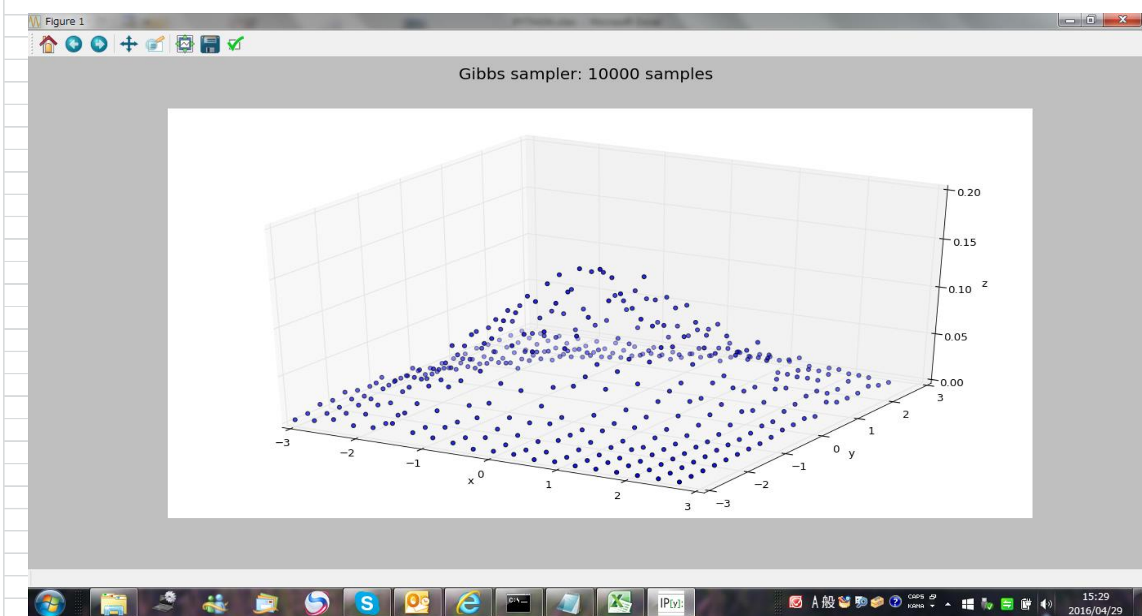


Figure 1
Gibbs sampler: 10000 samples
0.20
0.15
0.10
0.05
0.00
z
x
y
15:29
2016/04/29

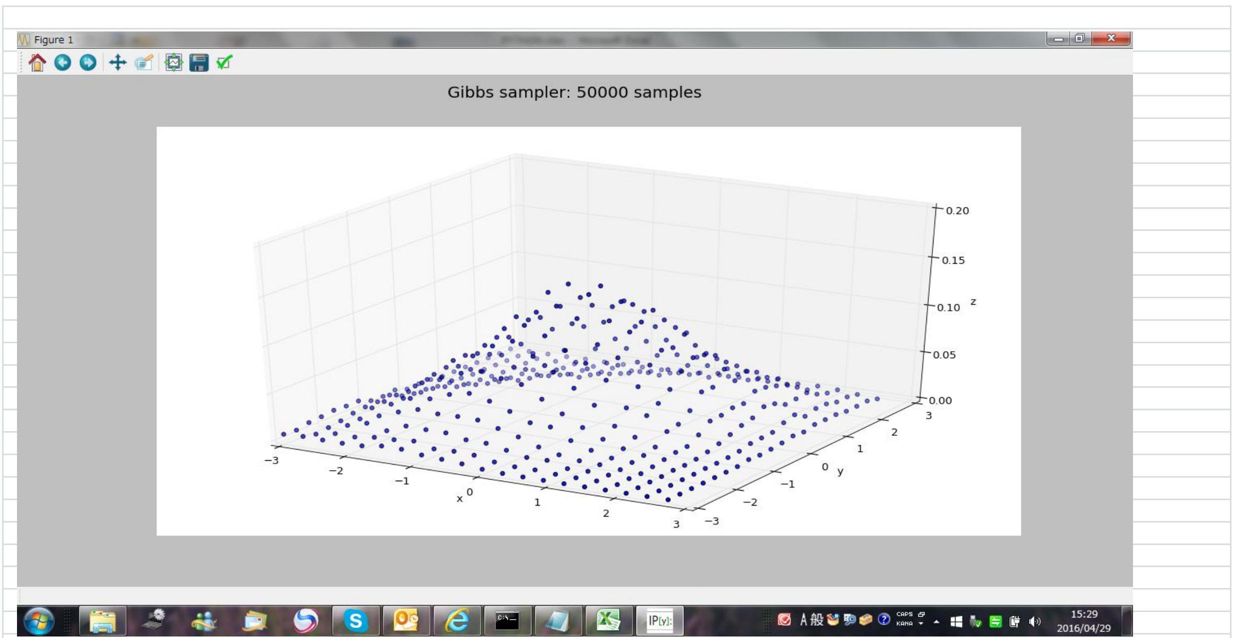


Figure 1
Gibbs sampler: 50000 samples
0.20
0.15
0.10
0.05
0.00
z
x
y
−3
−2
−1
0
1
2
3
15:29
2016/04/29

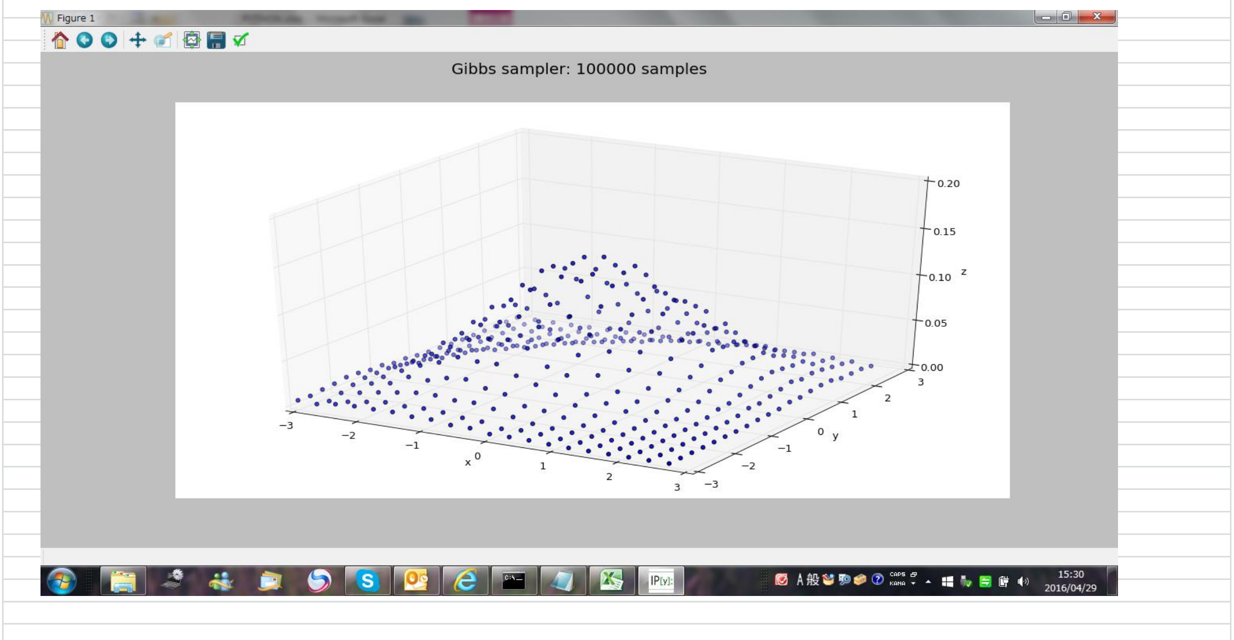


Figure 1
Gibbs sampler: 100000 samples
0.20
0.15
0.10
0.05
0.00
z
x
y
−3
−2
−1
0
1
2
3
15:30
2016/04/29

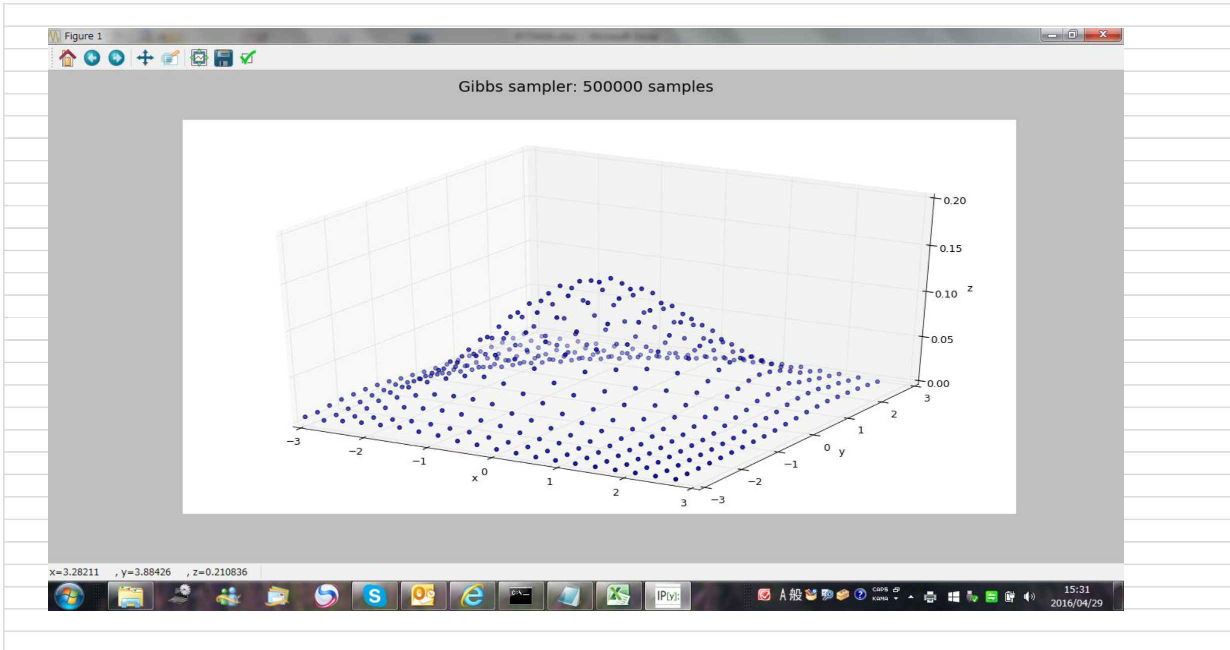


Figure 1
Gibbs sampler: 500000 samples
x=3.28211 , y=3.88426 , z=0.210836

```
#!/usr/bin/env python
# coding: utf-8

import copy
import numpy as np
import matplotlib.pyplot as plt

# P(x) : Target distribution
def P(x1, x2, b):
    return np.exp(-0.5 * (x1**2 - 2*b*x1*x2 + x2**2))

# Q(x) : Proposal distribution
def Q(c, mu1, mu2, sigma):
    return (c[0] + np.random.normal(mu1, sigma), c[1] + np.random.normal(mu2, sigma))

def metropolis(N):
    current = (10, 10)
    sample = []
    sample.append(current)
    accept_ratio = []

    for i in range(N):
        candidate = Q(current, mu1, mu2, sigma)

        T_prev = P(current[0], current[1], b)
        T_next = P(candidate[0], candidate[1], b)
        a = T_next / T_prev

        if a > 1 or a > np.random.uniform(0, 1):
            # Update state
            current = copy.copy(candidate)
            sample.append(current)
            accept_ratio.append(i)

    print 'Accept ratio :', float(len(accept_ratio)) / N
    return np.array(sample)

b = 0.5
mu1 = 0
mu2 = 0
sigma = 1

N = 30000
burn_in = 0.2

sample = metropolis(N)

plt.scatter(
    sample[int(len(sample) * burn_in):, 0],
    sample[int(len(sample) * burn_in):, 1],
    alpha=0.3,
    s=5,
    edgecolor='None'
)
plt.title('MCMC (Metropolis)')
plt.show()

fig = plt.figure(figsize=(15, 6))

ax = fig.add_subplot(121)
plt.hist(sample[int(N * burn_in):,0], bins=30)
plt.title('x')

ax = fig.add_subplot(122)
plt.hist(sample[int(N * burn_in):,1], bins=30)
plt.title('y')
plt.show()

print 'x :', np.mean(sample[int(len(sample) * burn_in):,0]), np.var(sample[int(len(sample) * burn_in):,0])
# x : -0.00252259614386 1.26378688755
print 'y :', np.mean(sample[int(len(sample) * burn_in):,1]), np.var(sample[int(len(sample) * burn_in):,1])
# y : -0.0174372516771 1.24832585103
```

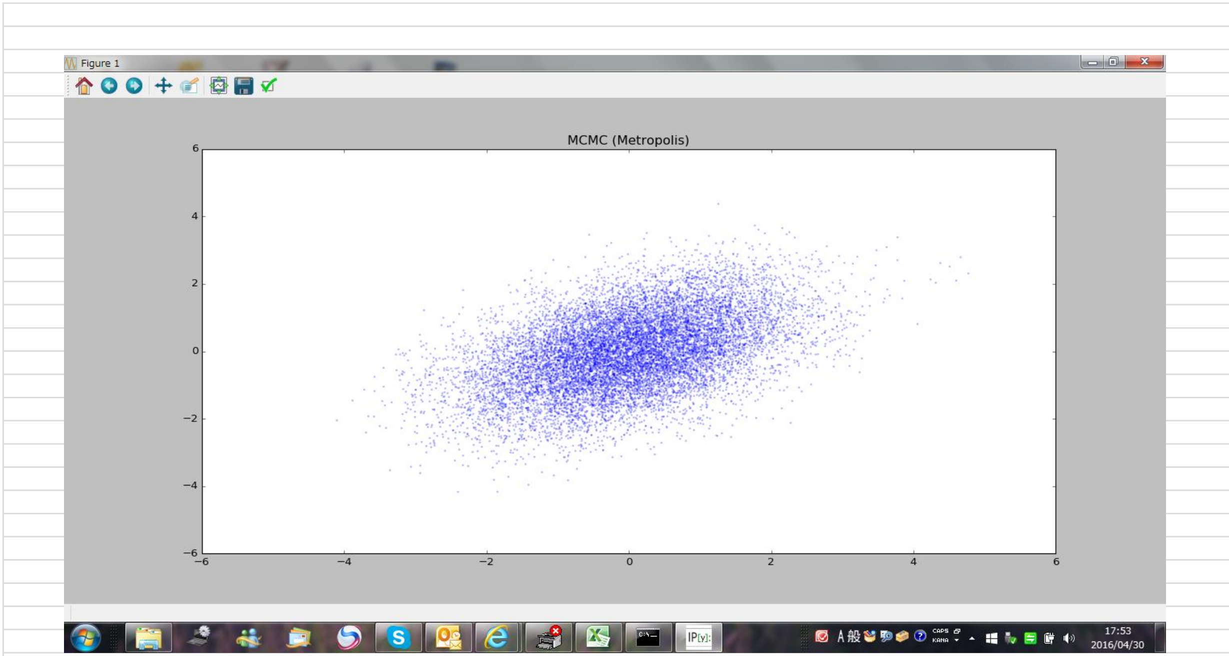


Figure 1
MCMC (Metropolis)
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6
17:53
2016/04/30

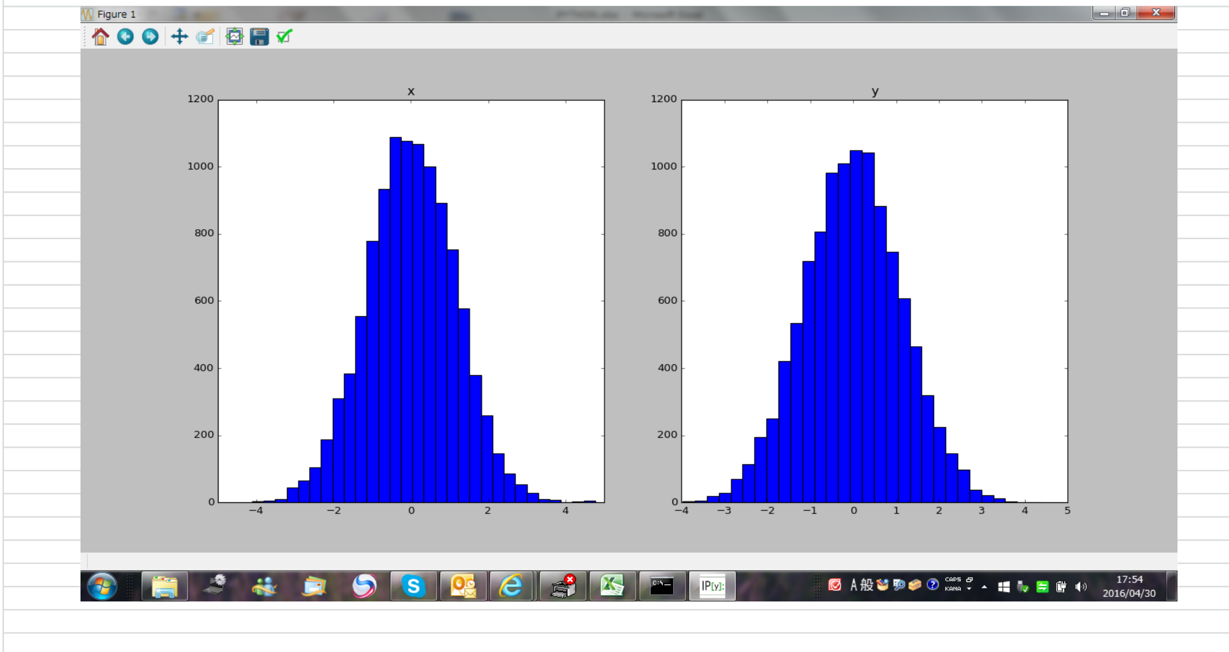
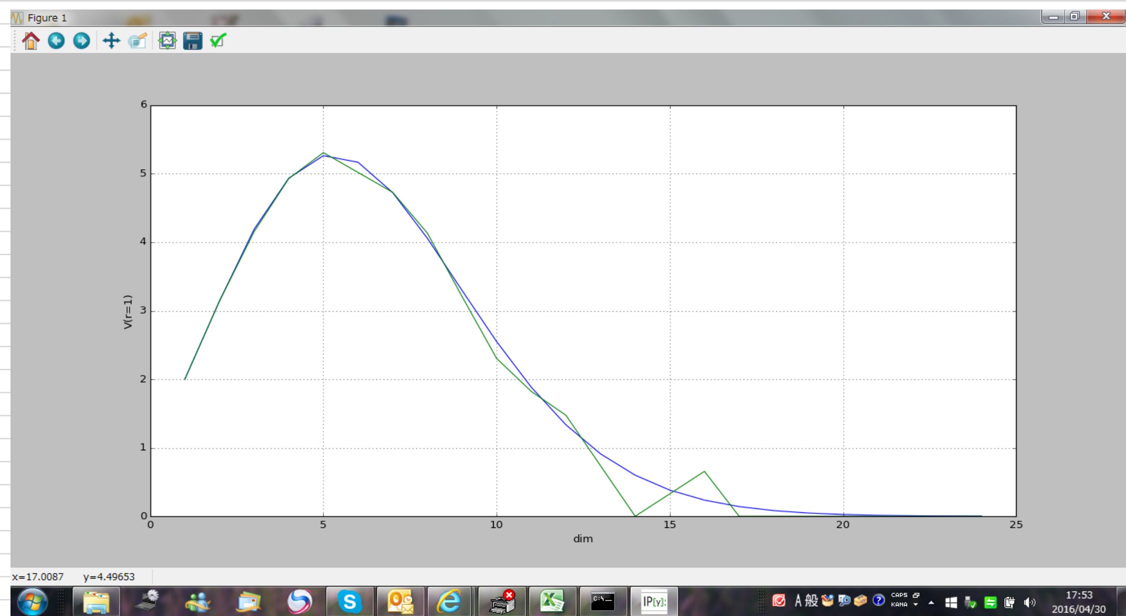


Figure 1
x
y
1200
1000
800
600
400
200
0
−4
−2
0
2
4
−4
−3
−2
−1
0
1
2
3
4
5
17:54
2016/04/30

```
mport math
import numpy as np
import matplotlib.pyplot as plt

def V(n):
    return math.pi**(n/2.0) / math.gamma(n/2.0+1.0)

def count_point(n):
    x = []
    count = 0

    for i in xrange(n):
        x.append(np.random.uniform(-1.0, 1.0, N))

    for i in xrange(N):
        r = 0.0
        for j in xrange(n):
            r += x[j][i]**2.0
        if r < 1.0:
            count += 1

    return count

tv = []
mc = []
N = 100000

for n in xrange(1,25):
    c = count_point(n)
    tv.append(V(n))
    mc.append(2.0**n * float(c) / float(N))

    print n, 'Dim'
    print 'Theoretical Value:', V(n)
    print 'Monte Carlo :', 2.0**n * float(c) / float(N)

x = np.arange(1, 25, 1)
plt.plot(x, tv)
plt.plot(x, mc)
plt.xlabel('dim')
plt.ylabel('V(r=1)')
plt.grid(True)
plt.show()
```



```
# -*- coding: utf-8 -*-
import numpy as np

#標準多変量正規分布
def standard_multivariate_normal_pdf(x):
    x = np.asarray(x)
    return 1.0/(2.0*np.pi)**(len(x)/2.0)*np.exp(-0.5*np.dot(x,x))

#定常分布（混合ガウシアン）
def equilibrium_distribution(x):
    x = np.asarray(x)
    #混合比率(3:7で混ぜる)
    theta = np.array((0.3, 0.7))
    #2つのガウシアンの確率密度（平均(1,1), (-1,-1)の2つ）
    p1 = standard_multivariate_normal_pdf(x-np.array(( 1,  1)))
    p2 = standard_multivariate_normal_pdf(x-np.array((-1, -1)))
    #混合分布の確率密度
    return np.dot(theta, np.array((p1,p2)))

#MCMC Generator by Metropolis Algorithm
def mcmc_metropolis(x):
    #次の位置の候補の作成
    x_proposed = x + 0.5*np.random.normal(size=len(x))
    #メトロポリスアルゴリズムに従って候補の棄却・採用判定を行う
    if(np.random.uniform(size=1) < equilibrium_distribution(x_proposed)/equilibrium_distribution(x)):
        return x_proposed
    else:
        return x

if __name__ == "__main__":
    #初期スタート位置
    x = np.array([[0,0]])
    #10000回のシミュレーション
    for i in range(10000):
        x = np.vstack((x, mcmc_metropolis(x[-1])))
    #サンプルの散布図 ＆ 真の確率密度の等高線
    import matplotlib.pyplot as plt
    #散布図
    plt.scatter(x[:,0],x[:,1],c="black",s=10,edgecolors="None",alpha=0.5)
    #等高線（真の確率密度から計算）
    axis_range = np.arange(-3.0, 3.0, 0.05)
    X,Y = np.meshgrid(axis_range, axis_range)
    Z = np.zeros(np.shape(X))
    for i in xrange(np.size(X,0)):
        for j in xrange(np.size(X,1)):
            Z[i,j] = equilibrium_distribution((X[i,j],Y[i,j]))
    CS = plt.contour(X,Y,Z,linewidths=5)
    plt.clabel(CS, fontsize=25)
    #各ガウシアンの平均値が見やすいように適用に線を入れる
    plt.axvline(x= 1,c='r',ls='--')
    plt.axvline(x=-1,c='b',ls='--')
    plt.axhline(y= 1,c='r',ls='--')
    plt.axhline(y=-1,c='b',ls='--')
    plt.show()
```

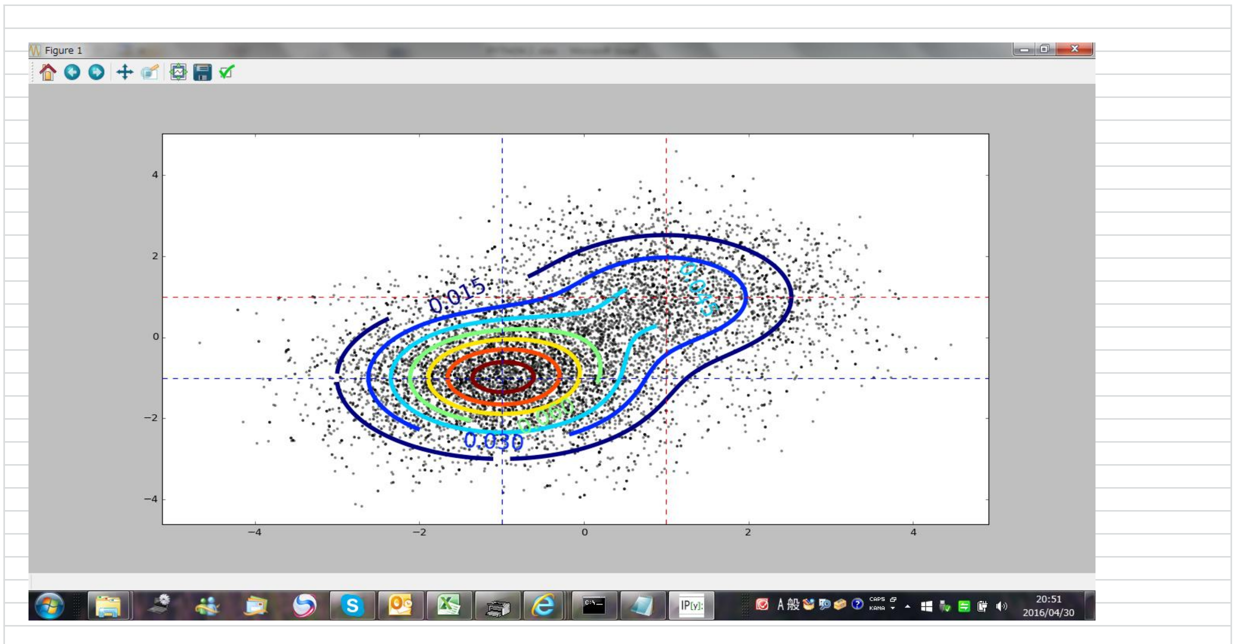
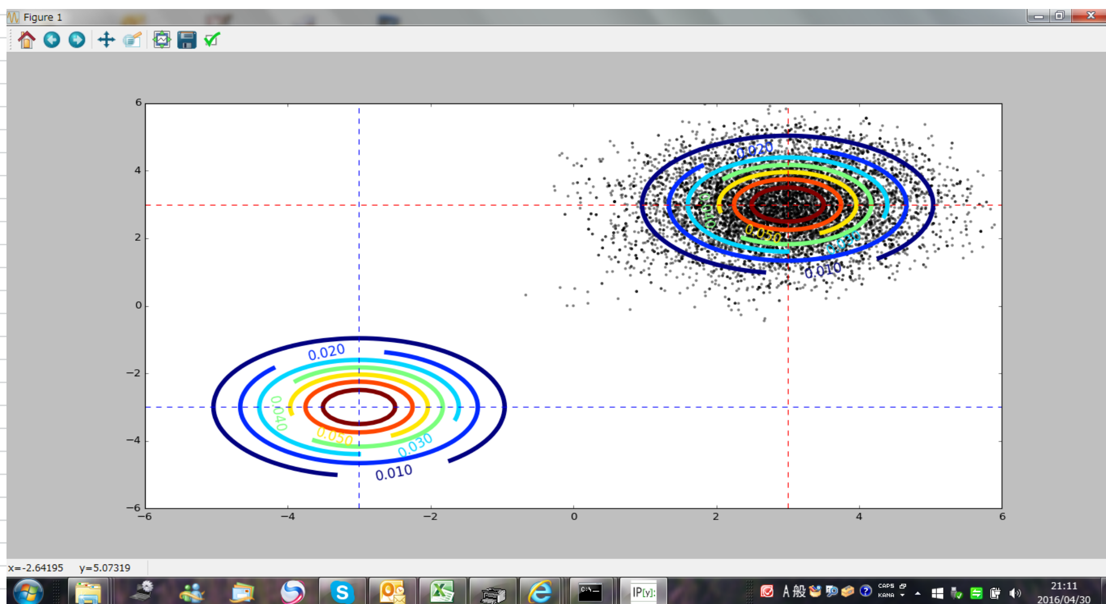


Figure 1
4
2
0
−2
−4
0.015
0.030
20:51
2016/04/30

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
#標準多変量正規分布
def standard_multivariate_normal_pdf(x):
    x = np.asarray(x)
    return 1.0/(2.0*np.pi)**(len(x)/2.0)*np.exp(-0.5*np.dot(x,x))
#混合ガウシアンの平均値を指定して生成する関数
def generate_equilibrium_distribution(mu):
    #定常分布（混合ガウシアン）
    def equilibrium_distribution(x):
        x = np.asarray(x)
        #混合比率(5:5で混ぜる)
        theta = np.array((0.5, 0.5))
        #2つのガウシアンの確率密度（平均(mu,mu), (-mu,-mu)の2つ）
        p1 = standard_multivariate_normal_pdf(x-np.array(( mu,  mu)))
        p2 = standard_multivariate_normal_pdf(x-np.array((-mu, -mu)))
        #混合分布の確率密度
        return np.dot(theta, np.array((p1,p2)))
    return equilibrium_distribution
#MCMC(メトロポリス法）による次候補生成
def next_with_metropolis_algorithm(x,equilibrium_distribution):
    #次の位置の候補の作成
    x_proposed = x + 0.5*np.random.normal(size=len(x))
    #メトロポリスアルゴリズムに従って候補の棄却・採用判定を行う
    if(np.random.uniform(size=1) < equilibrium_distribution(x_proposed)/equilibrium_distribution(x)):
        return x_proposed
    else:
        return x
#通常のマルコフ連鎖モンテカルロ法による分布生成
def mcmc_normal(mu, size_simulation):
    #定常分布の平均値((-mu,-mu),(mu,mu))設定
    equilibrium_distribution = generate_equilibrium_distribution(mu)
    #初期スタート位置
    x = np.array([[0,0]])
    for i in range(size_simulation):
        x = np.vstack((x, next_with_metropolis_algorithm(x[-1],equilibrium_distribution)))
    return x
#レプリカ交換モンテカルロ法による分布生成
def mcmc_replica_exchange(mu, size_simulation, size_replica, frequency_exchange):
    #初期スタート位置
    x = [[np.random.uniform(-mu,mu,2)] for i in range(size_replica)]
    #適当に間を切って定常分布”族”生成
    equilibrium_distributions = [generate_equilibrium_distribution(var) for var in np.linspace(0,mu,num=size_replica)]
    #シミュレーション
    for i in range(size_simulation):
        #通常のMCMCシミュレーション（メトロポリス法）
        for index_replica in range(size_replica):
            x_next = next_with_metropolis_algorithm(x[index_replica][-1],equilibrium_distributions[index_replica])
            x[index_replica] = np.vstack((x[index_replica], x_next))
        #適当な交換頻度に応じたレプリカの交換
        if(i % frequency_exchange == 0):
            index_exchange = int(np.random.uniform(0,size_replica-1))
            x1 = x[index_exchange  ][-1]
            x2 = x[index_exchange+1][-1]
            eq_dist1 = equilibrium_distributions[index_exchange]
            eq_dist2 = equilibrium_distributions[index_exchange+1]
            #メトロポリスアルゴリズムに従って状態交換の採択・棄却判定
            if(np.random.uniform(size=1) < (eq_dist1(x2)*eq_dist2(x1))/(eq_dist1(x1)*eq_dist2(x2))):
                x[index_exchange][-1], x[index_exchange+1][-1] = np.copy(x2),np.copy(x1)
    return x[size_replica-1]
    plt.show()
```

```
#サンプルの散布図 & 真の確率密度の等高線
def plot_result(x, mu):
    equilibrium_distribution = generate_equilibrium_distribution(MU)
    axis_range = mu+3.0
    #散布図
    plt.scatter(x[:,0],x[:,1],c="black",s=10,edgecolors="None",alpha=0.5)
    #等高線(真の確率密度から計算)
    mesh = np.arange(-axis_range,axis_range,0.1)
    X,Y = np.meshgrid(mesh, mesh)
    Z = np.zeros(np.shape(X))
    for i in xrange(np.size(X,0)):
        for j in xrange(np.size(X,1)):
            Z[i,j] = equilibrium_distribution((X[i,j],Y[i,j]))
    CS = plt.contour(X,Y,Z,linewidths=5)
    plt.clabel(CS,fontsize=15)
    #各ガウシアンの平均値が見やすいように適用に線を入れる
    plt.axvline(x= mu,c='r',ls='--')
    plt.axvline(x=-mu,c='b',ls='--')
    plt.axhline(y= mu,c='r',ls='--')
    plt.axhline(y=-mu,c='b',ls='--')
    plt.axis([-axis_range, axis_range,-axis_range, axis_range])
    plt.show()
#
if __name__ == "__main__":
    #定常分布にしたい混合ガウシアンの平均値(2つのガウシアンにおいて(mu,mu), (-mu,-mu)が平均となる)
    MU = 3.0
    #レプリカの個数
    SIZE_REPLICA = 5
    #何ステップに一回状態交換を計算するか
    FREQUENCY_EXCHANGE = 20
    #シミュレーション回数
    SIZE_SIMULATION= 10000

    #通常のマルコフ連鎖モンテカルロ法による混合ガウス分布生成
    x = mcmc_normal(MU,SIZE_SIMULATION)
    plot_result(x, MU)
    print np.mean(x, axis=0)

    #レプリカ交換モンテカルロ法による混合ガウス分布生成
    x = mcmc_replica_exchange(MU,SIZE_SIMULATION,SIZE_REPLICA,FREQUENCY_EXCHANGE)
    plot_result(x, MU)
    print np.mean(x, axis=0)
```



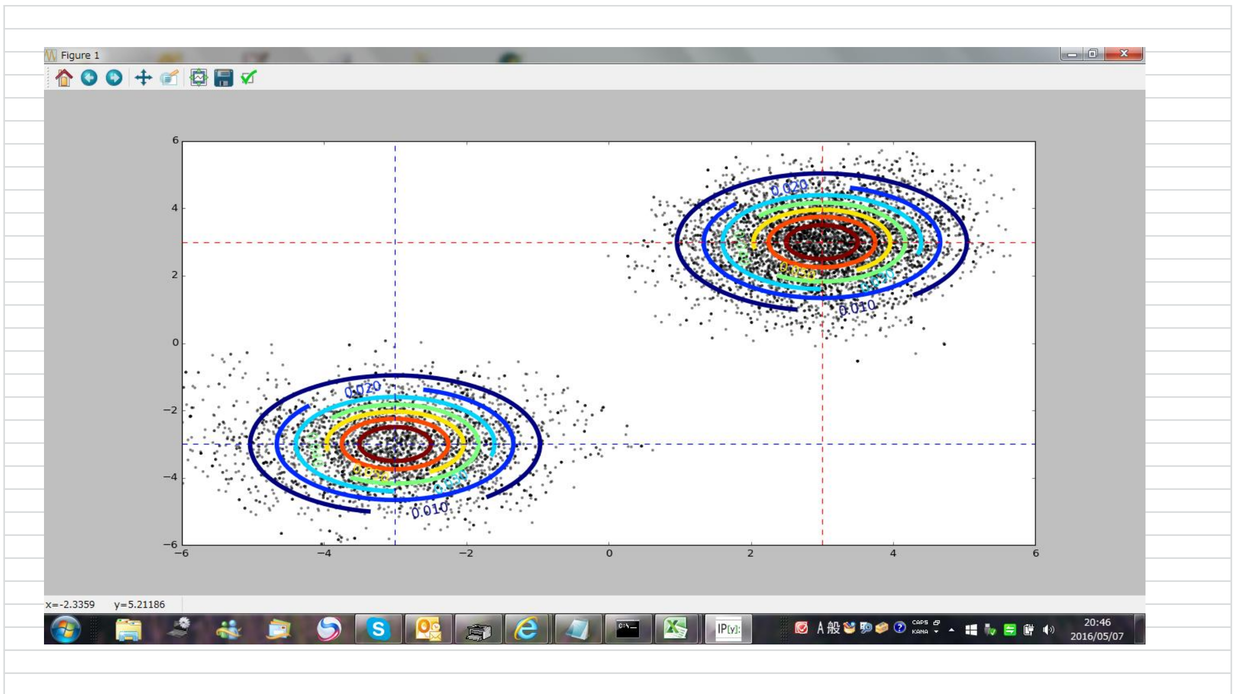
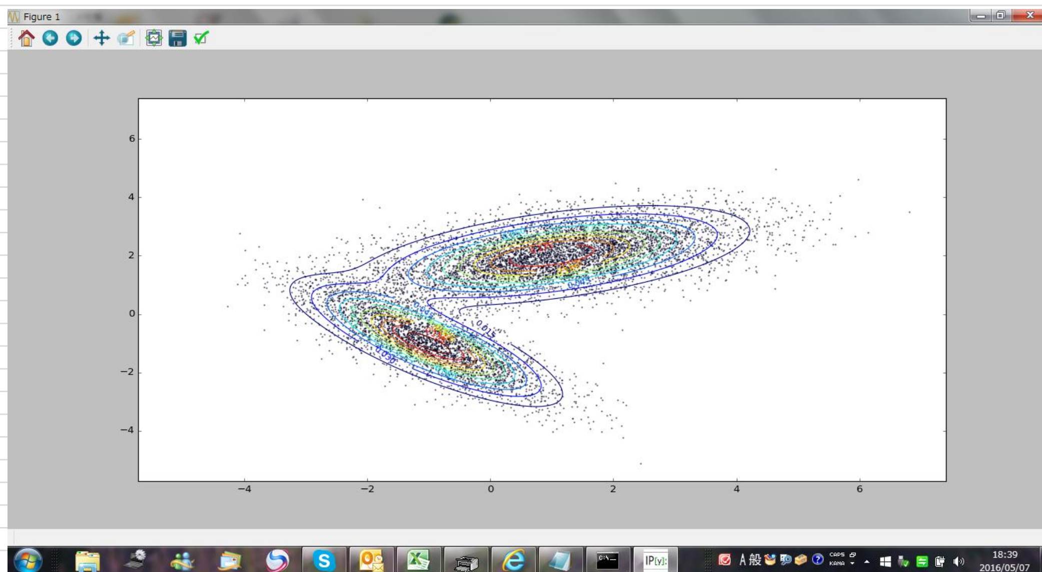


Figure 1
0.020
0.010
x=-2.3359 y=5.21186
20:46
2016/05/07

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

def q(x, y):
    g1 = mlab.bivariate_normal(x, y, 1.0, 1.0, -1, -1, -0.8)
    g2 = mlab.bivariate_normal(x, y, 1.5, 0.8, 1, 2, 0.6)
    return 0.6*g1+28.4*g2/(0.6+28.4)

'''Metropolis Hastings'''
N = 100000
s = 10
r = np.zeros(2)
p = q(r[0], r[1])
print p
samples = []
for i in xrange(N):
    rn = r + np.random.normal(size=2)
    pn = q(rn[0], rn[1])
    if pn >= p:
        p = pn
        r = rn
    else:
        u = np.random.rand()
        if u < pn/p:
            p = pn
            r = rn
    if i % s == 0:
        samples.append(r)

samples = np.array(samples)
plt.scatter(samples[:, 0], samples[:, 1], alpha=0.5, s=1)

'''Plot target'''
dx = 0.01
x = np.arange(np.min(samples), np.max(samples), dx)
y = np.arange(np.min(samples), np.max(samples), dx)
X, Y = np.meshgrid(x, y)
Z = q(X, Y)
CS = plt.contour(X, Y, Z, 10)
plt.clabel(CS, inline=1, fontsize=10)
plt.show()
```



```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
from mpl_toolkits.mplot3d import Axes3D
from sklearn import mixture

def q(x, y):
g1 = mlab.bivariate_normal(x, y, 1.0, 1.0, -1, -1, -0.8)
g2 = mlab.bivariate_normal(x, y, 1.5, 0.8, 1, 2, 0.6)
return 0.6*g1+28.4*g2/(0.6+28.4)

def plot_q():
fig = plt.figure()
ax = fig.gca(projection='3d')
X = np.arange(-5, 5, 0.1)
Y = np.arange(-5, 5, 0.1)
X, Y = np.meshgrid(X, Y)
Z = q(X, Y)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=plt.get_cmap('coolwarm'),
linewidth=0, antialiased=True)
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.savefig('3dgauss.png')
plt.clf()


'''Metropolis Hastings'''
N = 10000
s = 10
r = np.zeros(2)
p = q(r[0], r[1])
print p
samples = []
for i in xrange(N):
rn = r + np.random.normal(size=2)
pn = q(rn[0], rn[1])
if pn >= p:
p = pn
r = rn
else:
u = np.random.rand()
if u < pn/p:
p = pn
r = rn
if i % s == 0:
samples.append(r)

samples = np.array(samples)
plt.scatter(samples[:, 0], samples[:, 1], alpha=0.5, s=1)

'''Plot target'''
dx = 0.01
x = np.arange(np.min(samples), np.max(samples), dx)
y = np.arange(np.min(samples), np.max(samples), dx)
X, Y = np.meshgrid(x, y)
Z = q(X, Y)
CS = plt.contour(X, Y, Z, 10, alpha=0.5)
plt.clabel(CS, inline=1, fontsize=10)
plt.savefig("samples.png")
return samples

def fit_samples(samples):
gmix = mixture.GMM(n_components=2, covariance_type='full')
gmix.fit(samples)
print gmix.means_
colors = ['r' if i==0 else 'g' for i in gmix.predict(samples)]
ax = plt.gca()
ax.scatter(samples[:,0], samples[:,1], c=colors, alpha=0.8)
plt.savefig("class.png")

if __name__ == '__main__':
plot_q()
s = sample()
fit_samples(s)

plt.show()
```

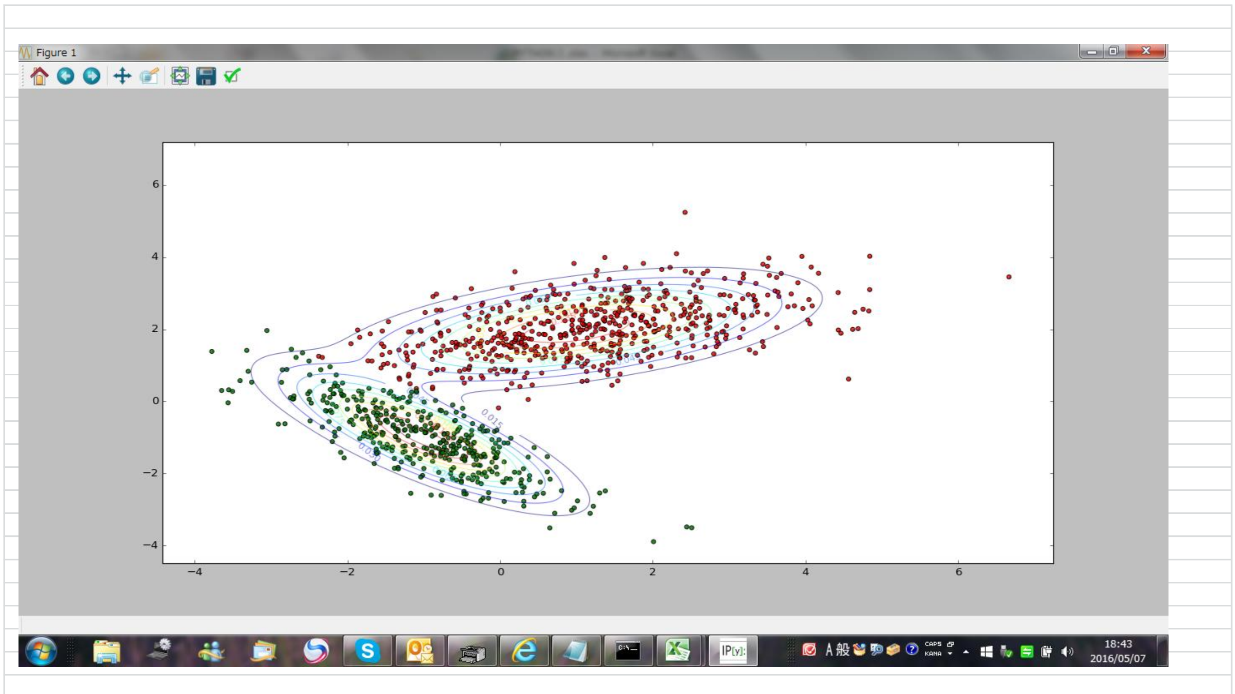


Figure 1
6
4
2
0
−2
−4
−4
−2
0
2
4
6
18:43
2016/05/07

```
import numpy as np
import matplotlib.pyplot as plt
# X座標を適当にサンプリングして指定、それに合わせてy = sin(x) + noiseを生成
np.random.seed(1)
x = np.sort(np.random.uniform(-np.pi, np.pi,100))
y = np.sin(x) + 0.1*np.random.normal(size=len(x))
# scikit-learnに突っ込むためにフォーマット変更
x = x.reshape((len(x),1))
# 全データの描画
plt.plot(x, y, 'o')
plt.show()

from sklearn import svm
from sklearn import cross_validation
#データを6割をトレーニング、4割をテスト用とする
x_train, x_test, y_train, y_test = cross_validation.train_test_split(x, y, test_size=0.4)
# 線でつないでplotする用にx_test・y_testをx_testの昇順に並び替える
index = x_test.argsort(0).reshape(len(x_test))
x_test = x_test[index]
y_test = y_test[index]
# サポートベクトル回帰を学習データ使って作成
reg = svm.SVR(kernel='rbf', C=1).fit(x_train, y_train)
# テストデータに対する予測結果のPLOT
plt.plot(x_test, y_test, 'bo-', x_test, reg.predict(x_test), 'ro-')
plt.show()


# 決定係数R^2
print reg.score(x_test, y_test)

from sklearn.grid_search import GridSearchCV
#RBFカーネルのパラメーターγと罰則Cを複数個作ってその中で(スコアの意味で)良い物を探索(カーネルもパラメーターとして使用可能)
tuned_parameters = [{'kernel': ['rbf'], 'gamma': [10**i for i in range(-4,0)], 'C': [10**i for i in range(1,4)]}]
gscv = GridSearchCV(svm.SVR(), tuned_parameters, cv=5, scoring="mean_squared_error")
gscv.fit(x_train, y_train)
#一番スコア悪い&良い奴を出す
params_min,_,_ = gscv.grid_scores_[np.argmin([x[1] for x in gscv.grid_scores_])]
reg_min = svm.SVR(kernel=params_min['kernel'], C=params_min['C'], gamma=params_min['gamma'])
reg_max = gscv.best_estimator_
#全トレーニングデータを使って再推計
reg_min.fit(x_train, y_train)
reg_min.fit(x_train, y_train)
#正答(青)&良い(赤)&悪い(緑)の結果をPLOT
plt.plot(x_test, y_test, 'bo-',x_test, reg_max.predict(x_test), 'ro-',x_test, reg_min.predict(x_test), 'go-')
plt.show()
```

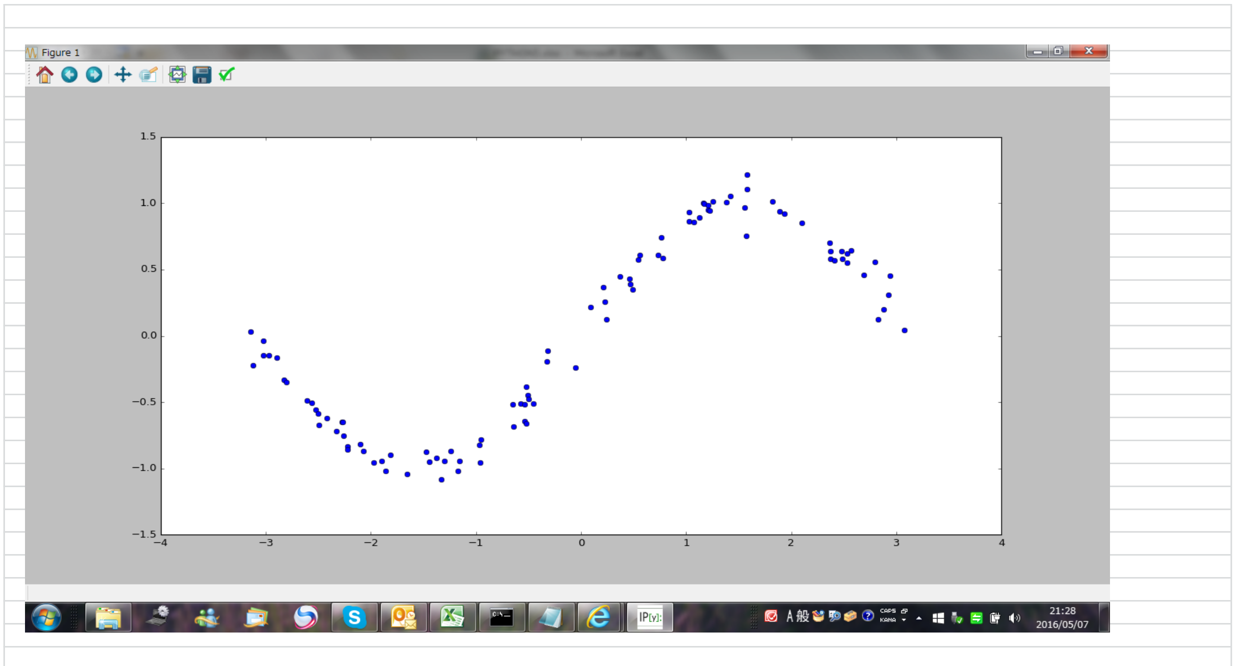


Figure 1
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−4
−3
−2
−1
0
1
2
3
4
21:28
2016/05/07

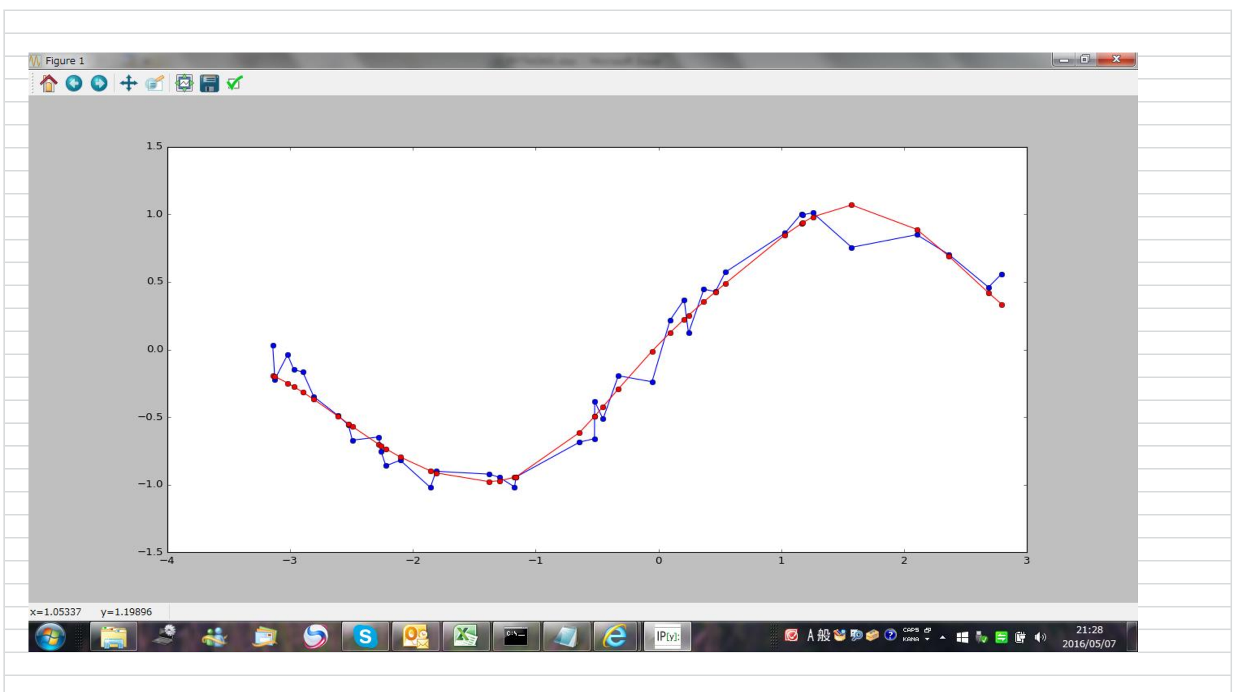


Figure 1
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−4
−3
−2
−1
0
1
2
3
x=1.05337 y=1.19896
21:28
2016/05/07

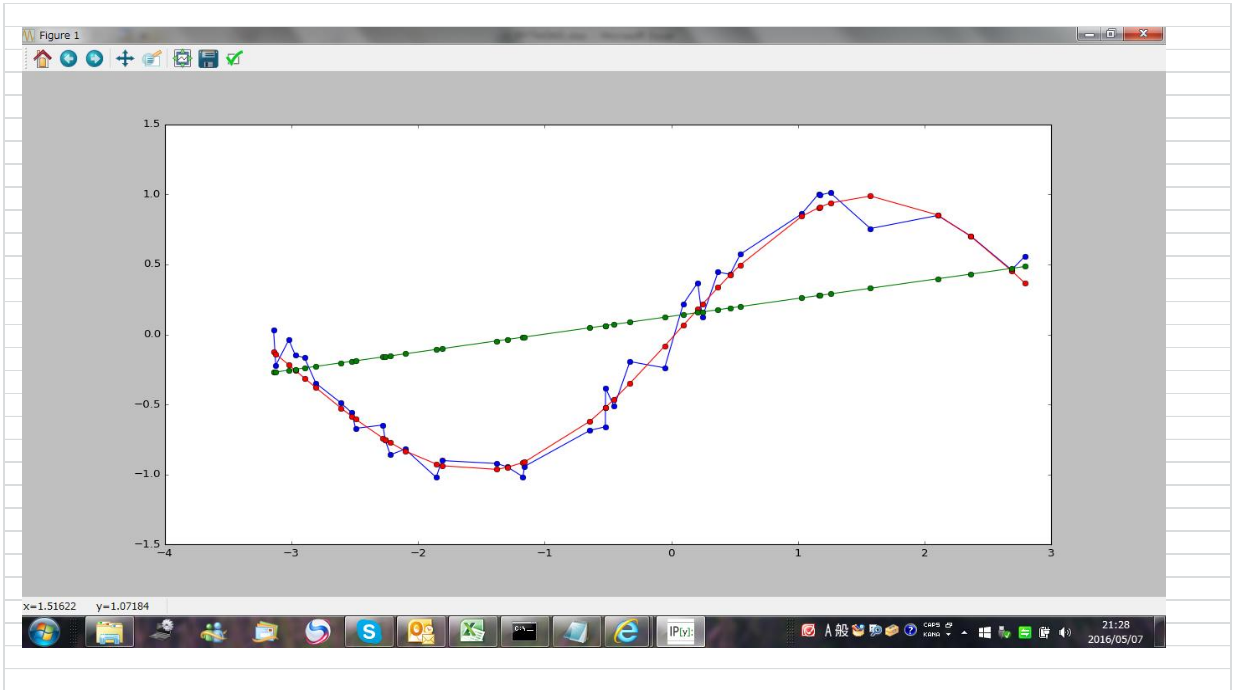
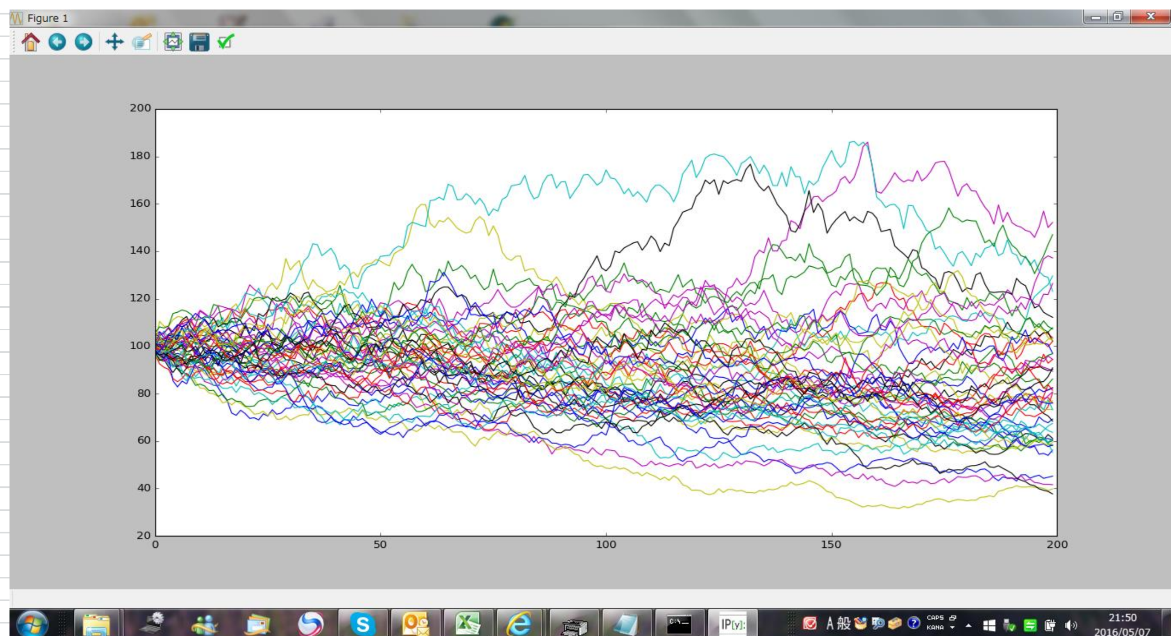


Figure 1
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−4
−3
−2
−1
0
1
2
3
x=1.51622
y=1.07184
21:28
2016/05/07

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt

#株価の確率微分方程式に従うサンプルパスを生成
#dS/S = r * dt + vol * dW_t
def generate_path(S0, T, r, vol, N, M):
    dt = T/M
    w = np.cumsum(np.reshape(np.random.standard_normal(N*M), (N,M)), 1) * (dt**0.5)
    t = np.cumsum(np.ones((N,M)), 1)*dt
    return S0 * np.exp((r-0.5*vol**2)*t + vol*w)

if __name__ == "__main__":
    #株価・満期・瞬間金利・ボラティリティ
    S0  = 100.0
    T   = 3.0
    R   = 0.01
    VOL = 0.2
    #パス数・時間刻み数
    N = 50
    M = 200
    #パスの生成＆描画
    x = generate_path(S0, T, R, VOL, N, M)
    plt.plot(x.T)
    plt.show()
```



```
# -*- coding: utf-8 -*-
import numpy
import matplotlib.pyplot
#不変分布(比率のみ，最初の要素に一番ウェイト高い奴を持ってくる)
INVARIANT_WEIGHT = [5.0] + [float(i+1) for i in range(4)]
#↑から計算される適当なグローバル変数(累積和・状態数)
INVARIANT_WEIGHT_CUM = numpy.cumsum(INVARIANT_WEIGHT)
NUM_OF_STATE = len(INVARIANT_WEIGHT)
#乱数生成機(seedは適当に設定)
GENERATOR_RAND = numpy.random.mtrand.RandomState(100)

#メトロポリス法で次の状態計算する関数
def Metropolis(current):
    prob        = GENERATOR_RAND.random_sample()
    candidate   = GENERATOR_RAND.randint(0, NUM_OF_STATE)
    weight_candidate = INVARIANT_WEIGHT[candidate]
    weight_current   = INVARIANT_WEIGHT[current]
    if prob < min(1.0,  weight_candidate / weight_current):
        return candidate
    else:
        return current
#Suwa-Todoらによる手法で次の状態を計算する関数,逆変換法を使ってる
def SuwaTodo(current):
    prob = GENERATOR_RAND.random_sample()
    prob_sum = 0.0
    for candidate in range(NUM_OF_STATE):
        prob_sum += (v_SuwaTodo(current, candidate) / INVARIANT_WEIGHT[current])
        if prob < prob_sum:
            return candidate
#Suwa-Todoらの手法で使用されている遷移確率×不変分布の値。論文ではv_ijとなってる
def v_SuwaTodo(i, j) :
    weight_i = INVARIANT_WEIGHT[i]
    weight_j = INVARIANT_WEIGHT[j]
    if j ==0 :
        invariant_cum_j = INVARIANT_WEIGHT_CUM[-1]
    else:
        invariant_cum_j = INVARIANT_WEIGHT_CUM[j - 1]
    delta = INVARIANT_WEIGHT_CUM[i] - invariant_cum_j + INVARIANT_WEIGHT[0]
    return max(0, min(delta, weight_i + weight_j - delta, weight_i, weight_j))


if __name__ == '__main__':
    #モンテカルロステップ数。列方向の次元
    NUM_OF_STEP = 40
    #MCMCの鎖の数。行方向の次元
    NUM_OF_CHAIN = 1000
    #初期状態
    initial_state = 3
    #MCMCの各ステップ・各鎖ごとのデータをためこむための行列
    metropolis = numpy.zeros((NUM_OF_CHAIN, NUM_OF_STEP))
    suwatodo   = numpy.zeros((NUM_OF_CHAIN, NUM_OF_STEP))
    #MCMCやる
    for row in range(NUM_OF_CHAIN):
        state_metropolis = initial_state
        state_suwatodo   = initial_state
        for col in range(NUM_OF_STEP):
            state_metropolis = Metropolis(state_metropolis)
            metropolis[row, col] = state_metropolis
            state_suwatodo = SuwaTodo(state_suwatodo)
            suwatodo[row, col]   = state_suwatodo

    #各ステップでの状態ごとの確率分布
    probability_series_metropolis = numpy.zeros((NUM_OF_STATE, NUM_OF_STEP))
    probability_series_suwatodo   = numpy.zeros((NUM_OF_STATE, NUM_OF_STEP))
    for col in range(NUM_OF_STEP):
        for state in range(NUM_OF_STATE):
            probability_series_metropolis[state, col] = len([i for i in metropolis[:, col] if i == state]) / float(NUM_OF_CHAIN)
            probability_series_suwatodo[state, col]   = len([i for i in suwatodo[:, col]   if i == state]) / float(NUM_OF_CHAIN)

    #ある特定の状態が出現する確率のステップ(時系列)推移
    matplotlib.pyplot.plot(probability_series_suwatodo[3].T, 'r',  label='Suwa-Todo',linewidth=3)
    matplotlib.pyplot.plot(probability_series_metropolis[3].T, 'b',  label='Metropolis',linewidth=3)
    matplotlib.pyplot.legend()
    matplotlib.pyplot.show()
```

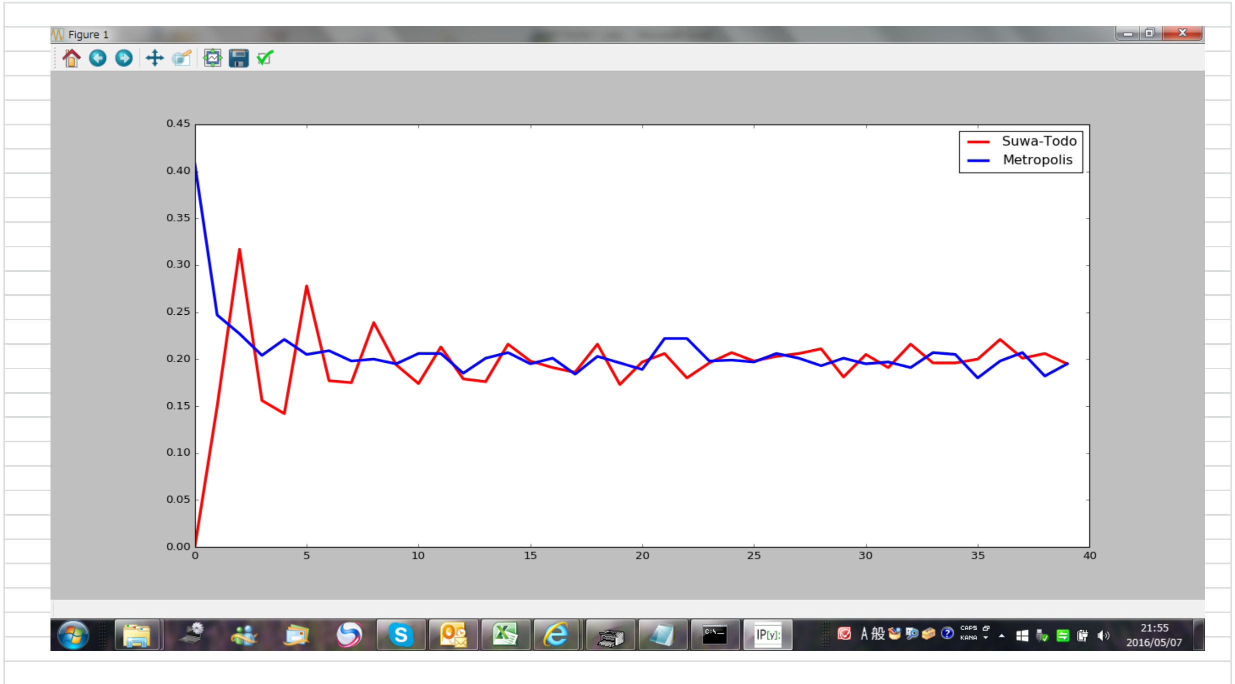


Figure 1
Suwa-Todo
Metropolis
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
5
10
15
20
25
30
35
40
21:55
2016/05/07