

# MATH 575B Numerical Analysis

## Spring 2020 class notes

Misha Stepanov

Department of Mathematics and Program in Applied Mathematics  
University of Arizona, Tucson, AZ 85721, USA

### Part V

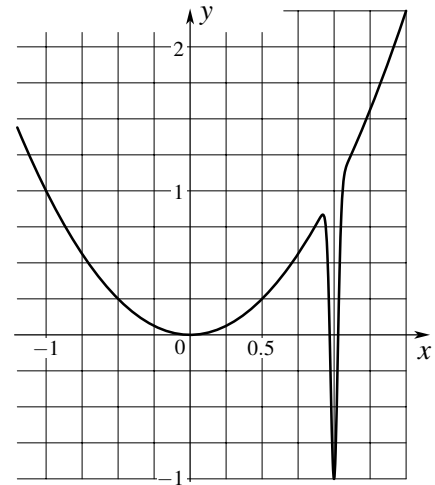
## Optimization

Suggested reading: [BoVa09, Chs. 9, 10, 11].

It is important to note that one can optimize *only one* function at once. Whenever you claim that you simultaneously optimize two (or more) functions, you are actually optimize a certain combination of those.

**Example V.1:** Consider a function with a narrow minimum,  $f(x) = x^2 - 2\exp(-(x-1)^2/2\sigma^2)$ , with  $\sigma$  being small, e.g.,  $\sigma = 1/40$ . It has a local minimum near  $x = 0$  and a global minimum near  $x = 1$ . If one doesn't test the values of the function near  $x = 1$ , one may not even realize the existence of the narrow peak pointing downward. Whenever you don't assume anything about the function you need to optimize, there is *no method* (other than brute-force exhaustive search within the whole feasible region) that will find a *global* optimum in a *guaranteed* way.

Maximizing  $f(\mathbf{x})$  is the same as minimizing  $-f(\mathbf{x})$ .



## 16 Least squares problem

Suggested reading: [TrBa97, Lecs. 11, 18, 19].

Consider a quadratic function  $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\hat{\mathbf{A}}\mathbf{x} - \mathbf{x}^T\mathbf{b} + c$ . At its minimum (or just extremum)  $\mathbf{x}_*$  the derivative (or gradient) is zero:  $\nabla f(\mathbf{x}_*) = \mathbf{0}$ . Derivative of quadratic function is linear, thus  $\nabla f(\mathbf{x}_*) = \mathbf{0}$  is a system of linear equations for  $\mathbf{x}_*$ .

In the combination  $\mathbf{x}^T\hat{\mathbf{A}}\mathbf{x}$  any assymmetric part of  $\hat{\mathbf{A}}$  is killed, and [if the matrix  $\hat{\mathbf{A}}$  is not symmetric] one can substitute  $\hat{\mathbf{A}}$  by  $\frac{1}{2}(\hat{\mathbf{A}} + \hat{\mathbf{A}}^T)$ . Matrix  $\hat{\mathbf{A}}$  then is diagonalizable with real eigenvalues. If at least one of the eugenvalues is strictly negative, then there is no lower bound for the values of  $f(\mathbf{x})$ . Consider some eigenvalue of  $\hat{\mathbf{A}}$  are zero, with  $\mathbf{y}$  being the corresponding eigenvector. If  $\mathbf{y} \cdot \mathbf{b} = \mathbf{y}^T\mathbf{b} \neq 0$ , then there is no lower bound for the values of  $f(\mathbf{x})$ .

The equation  $\nabla f(\mathbf{x}_*) = \mathbf{0}$  for the position of minimum  $\mathbf{x}_*$  reads as  $\hat{\mathbf{A}}\mathbf{x}_* = \mathbf{b}$ . The solution can be found by standard methods like Gaussian elimination, QR factorization, or (as  $\hat{\mathbf{A}}$  is real symmetric) using Cholesky factorization  $\hat{\mathbf{A}} = \hat{\mathbf{R}}^T\hat{\mathbf{R}}$ , where  $\hat{\mathbf{R}}$  is upper triangular. The latter can be done two times faster than standard LU factorization. See, e.g., [TrBa97, Lec. 23].

A least squares problem in linear algebra is the best “solution” to an overdetermined system of linear equations  $\hat{A}\mathbf{x} = \mathbf{b}$ , where  $\hat{A}$  is an  $m \times n$  matrix with  $m > n$ . We have  $m$  equations for  $n$  unknowns, and unless the equations are redundant, one doesn’t expect a solution to exist. Instead we look for the vector  $\mathbf{x}$  that minimizes the norm of the residual:

$$\mathbf{x}_* := \arg \min_{\mathbf{x}} \|\hat{A}\mathbf{x} - \mathbf{b}\|_2 = \arg \min_{\mathbf{x}} \left( \mathbf{x}^T \hat{A}^T \hat{A} \mathbf{x} - \mathbf{x}^T \hat{A}^T \mathbf{b} - \mathbf{b}^T \hat{A} \mathbf{x} + \mathbf{b}^T \mathbf{b} \right)$$

The minimized function is clearly bounded from below by 0, and if the matrix  $\hat{A}$  is of full rank, *i.e.*,  $\text{rank } \hat{A} = n < m$ , then the solution is unique:  $\mathbf{x}_* = (\hat{A}^T \hat{A})^{-1} \hat{A}^T \mathbf{b}$ . Of course, numerically the solution is found without forming the matrix  $\hat{A}^T \hat{A}$ , as  $\kappa(\hat{A}^T \hat{A}) = \kappa^2(\hat{A})$ . If  $\hat{A} = \hat{Q}\hat{R}$  and  $\hat{A} = \hat{U}\hat{\Sigma}\hat{V}^\dagger$  are the QR factorization and SVD of  $\hat{A}$ , respectively, then  $\mathbf{x}_* = \hat{R}^{-1} \hat{Q}^\dagger \mathbf{b} = \hat{V} \hat{\Sigma}^{-1} \hat{U}^\dagger \mathbf{b}$  (the action by  $\hat{R}^{-1}$  and by  $\hat{\Sigma}^{-1}$  is done by back substitution or by dividing by the diagonal matrix elements of  $\hat{\Sigma}$ ).

## Problems and exercises

1. Consider the problem of fitting the cloud of points  $(x_i, y_i)$ ,  $1 \leq i \leq N$ , by a linear function  $y = ax + b$ . The fit minimizes the sum of squares  $\sum_{i=1}^N (ax_i + b - y_i)^2$ . Write down explicit formulas for  $a$  and  $b$ . When the solution for  $a$  and  $b$  is not unique?

2. Fit the cloud of points  $\mathcal{S}_5 = \{(-2, -3), (-1, -1), (0, 5), (2, 5), (3, 1)\}$  by  $y = ax + b$  line using the least squares method,<sup>1</sup> *i.e.*, solve the following least squares problem

$$\begin{bmatrix} a_* \\ b_* \end{bmatrix} := \arg \min_{a, b} \left\| \begin{bmatrix} -2 & 1 \\ -1 & 1 \\ 0 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} - \begin{bmatrix} -3 \\ -1 \\ 5 \\ 5 \\ 1 \end{bmatrix} \right\|_2$$

## 17 Descent methods

Consider a continuously differentiable function  $f(\mathbf{x})$ . The gradient vector  $\nabla f(\mathbf{x})$  is the direction of the fastest growth of the function  $f(\mathbf{x})$ , locally we have  $f(\mathbf{x} + \boldsymbol{\epsilon}) \approx f(\mathbf{x}) + (\nabla f(\mathbf{x})) \cdot \boldsymbol{\epsilon}$ . If we move against its gradient,  $d\mathbf{x}/dt := -\nabla f(\mathbf{x})$ , then the function  $f(\mathbf{x}(t))$  is non-increasing:  $df(\mathbf{x})/dt = \nabla f(\mathbf{x}) \cdot d\mathbf{x}/dt = -\|\nabla f\|^2 \leq 0$ . This gives an idea how to compute the position  $\mathbf{x}_*$  of the minimum:

**Algorithm**  $\dot{\mathbf{x}} = -\nabla f$ : The minimum  $\mathbf{x}_*$  is estimated as  $\mathbf{x}(t)$  at sufficiently large  $t$ . The trajectory  $\mathbf{x}(t)$  is computed by some ODE solver, one needs to supply the initial condition  $\mathbf{x}(0)$ .

Some example functions that are going to be used:

$$G_2(x, y) = -x^2 - y^2 - x(x+y)^2 + (x^2 + y^2)^2$$

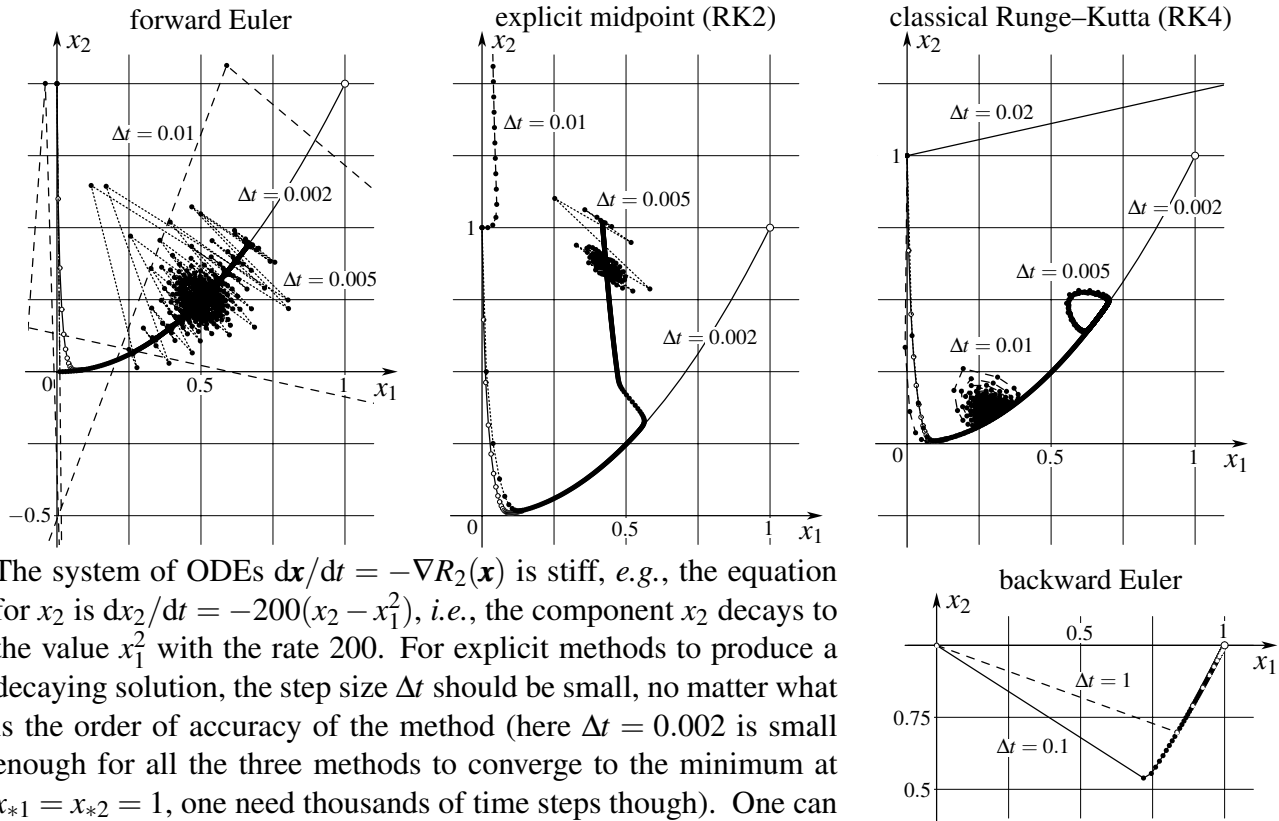
$$\text{Rosenbrock function}^2 R_n(x_1, x_2, \dots, x_n) = \sum_{i=1}^n (x_i - 1)^2 + A \sum_{i=1}^{n-1} (x_{i+1} - x_i^2)^2, \quad A \text{ is large, e.g., } A = 100$$

Here the lower index indicates the number of variables.

<sup>1</sup> See also the [Anscombe’s quartet](#) data.

<sup>2</sup> H. H. Rosenbrock, *An automatic method for finding the greatest or least value of a function*, The Computer Journal **3** (3) 175–184 (1960).

This algorithm may work not too well. Here is it being applied to the 2-dimensional Rosenbrock function  $R_2(x_1, x_2)$  with  $x_1(0) = 0$  and  $x_2(0) = 1$ :



The system of ODEs  $\dot{\mathbf{x}} = -\nabla R_2(\mathbf{x})$  is stiff, *e.g.*, the equation for  $x_2$  is  $\dot{x}_2 = -200(x_2 - x_1^2)$ , *i.e.*, the component  $x_2$  decays to the value  $x_1^2$  with the rate 200. For explicit methods to produce a decaying solution, the step size  $\Delta t$  should be small, no matter what is the order of accuracy of the method (here  $\Delta t = 0.002$  is small enough for all the three methods to converge to the minimum at  $x_{*1} = x_{*2} = 1$ , one need thousands of time steps though). One can try a method with stiff decay property, *e.g.*, backward Euler, but that would result in solving a system of [non-linear] equation in each time step. The whole minimization problem is more or less equivalent to the system of equations  $\nabla f(\mathbf{x}_*) = \mathbf{0}$ , so solving a system at each time step seems to be too expensive. Each step of backward Euler can be considered as solving the following minimization problem:

$$\mathbf{x}(t + \Delta t) := \arg \min_{\mathbf{X}} \left( \frac{\|\mathbf{X} - \mathbf{x}(t)\|^2}{2\Delta t} + f(\mathbf{X}) \right)$$

The function in brackets is equal to 0 at  $\mathbf{X} = \mathbf{x}(t)$ , and  $\|\mathbf{X} - \mathbf{x}(t)\| \geq 0$ , so it is guaranteed that  $f(\mathbf{x}(t + \Delta t)) \leq f(\mathbf{x}(t))$ . Still Algorithm  $\dot{\mathbf{x}} = -\nabla f$  (with backward Euler as a method of solving the system of ODEs) at best could be viewed as an application of continuation method to solve the system  $\nabla f(\mathbf{x}_*) = \mathbf{0}$ , which one may want to try, *e.g.*, due to no good initial guess for  $\mathbf{x}_*$ .

Here is how the  $\nabla R_2(\mathbf{x}_*) = \mathbf{0}$  system [of two equations] is solved in GNU Octave<sup>3</sup>

```
octave:1> fsolve(@(x) [2. * (1. - x(1)) - 400. * x(1) * ((x(1))^2 - x(2)), 200.
* ((x(1))^2 - x(2))], [0., 1.])
ans =

    0.84715    0.71697
```

```
octave:2> format long; options = optimset('TolX', 1.e-13, 'MaxIter', 10000);
octave:3> fsolve(@(x) [2. * (1. - x(1)) - 400. * x(1) * ((x(1))^2 - x(2)), 200.
```

<sup>3</sup> MATLAB® is a commercial software, see [MathWorks MATLAB licensing for UA Faculty, Staff & Students](#). GNU Octave is one of several (less effective) free alternatives to MATLAB, with mostly compatible syntax.

```

* ((x(1))^2 - x(2)), [0., 1.], options)
ans =

    9.999996886399101e-01    9.999993759833326e-01

octave:4> newton(@(x) [2. * (1. - x(1)) - 400. * x(1) * ((x(1))^2 - x(2)); 200.
* ((x(1))^2 - x(2))], [0.; 1.])'
counter =    7
ans =

    1    1

```

Here `newton.m` implements the Newton–Raphson method, with the Jacobian matrix  $\nabla f$  being computed through finite differences. Solving the system  $\nabla f(\mathbf{x}_*) = \mathbf{0}$  by Newton’s method in order to minimize the function  $f$  is discussed in Sec. 18. Notice that for the built-in solver `fsolve` to produce an answer close to the exact  $x_{*1} = x_{*2} = 1$ , we had to tweak the solver parameters (mainly the maximal number of iterations allowed) using `optimset` command.

The trajectories  $\mathbf{x}(t)$  on “backward Euler” picture were obtained by the following MATLAB script:

```

X12 = @(x) (x(1))^2 - x(2);
RHS = @(x) [2. * (1. - x(1)) - 400. * x(1) * X12(x); 200. * X12(x)];
x0 = [0.; 1.]; dt = 0.1; diff = 1.; options = optimset('MaxIter', 10000);
while (diff > 1.e-6)
    x = fsolve(@(x) ((x - x0) / dt - RHS(x)), x0, options);
    diff = norm(x - x0); x0 = x
end

```

Here is the pseudo-code of a general descent method, with possible variants of its steps:

start with some initial guess  $\mathbf{x}$

while (stopping critidia is not met) do

$$\|\nabla f(\mathbf{x})\| \leq \text{some small number}$$

pick direction  $\Delta \mathbf{x}$

$$\text{gradient descent: } \Delta \mathbf{x} \propto -\nabla f(\mathbf{x})$$

$$\text{steepest descent: } \Delta \mathbf{x} := \arg \min_{\|\Delta \mathbf{x}\|=1} \nabla f \cdot \Delta \mathbf{x}$$

$$\|\cdot\| \text{ could be, e.g., } L^1\text{- or weighted norm}$$

line search: choose step size  $t$

exact line search:  $t := \arg \min_s f(\mathbf{x} + s\Delta \mathbf{x})$ , i.e., the step size  $t$  is found from *exact* one-dimensional minimization<sup>4</sup>

backtracking line search: start with some not too small  $t$ , then reduce  $t$  until  $f(\mathbf{x} + t\Delta \mathbf{x}) < f(\mathbf{x}) + \alpha \nabla f \cdot (t\Delta \mathbf{x})$ ,  $0 \leq \alpha < 1$

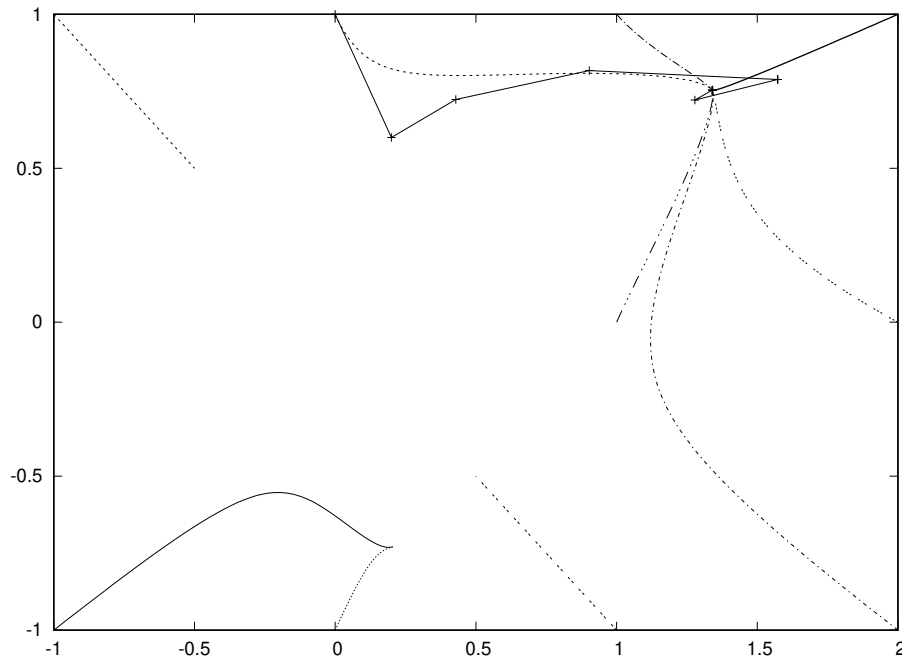
update:  $\mathbf{x} := \mathbf{x} + t\Delta \mathbf{x}$

return  $\mathbf{x}$

---

<sup>4</sup> This may be ...

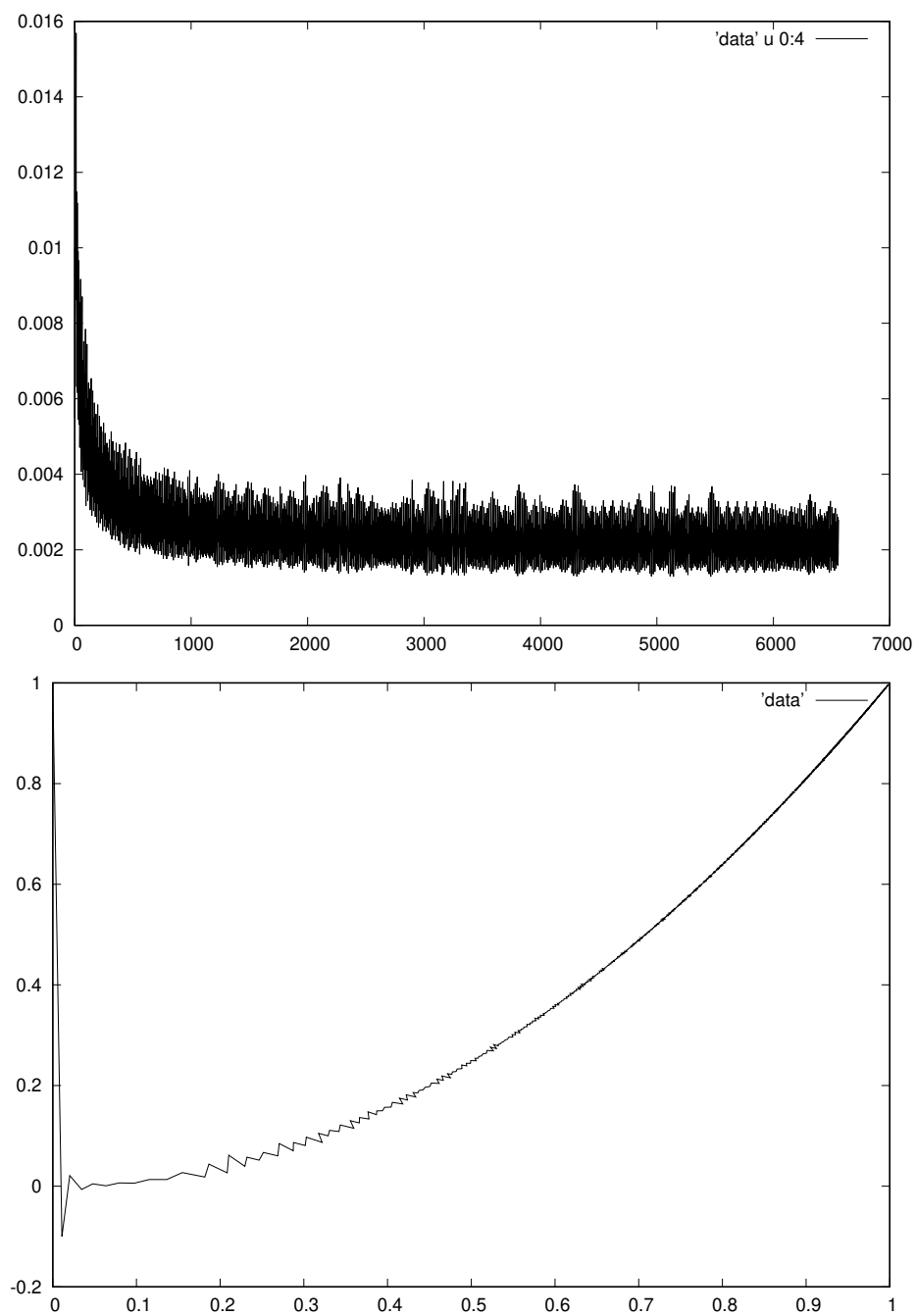
**Example 17.1:** Here is the gradient descent method with backtraching,  $\alpha = 0$ , applied to  $G_2(x,y)$ :

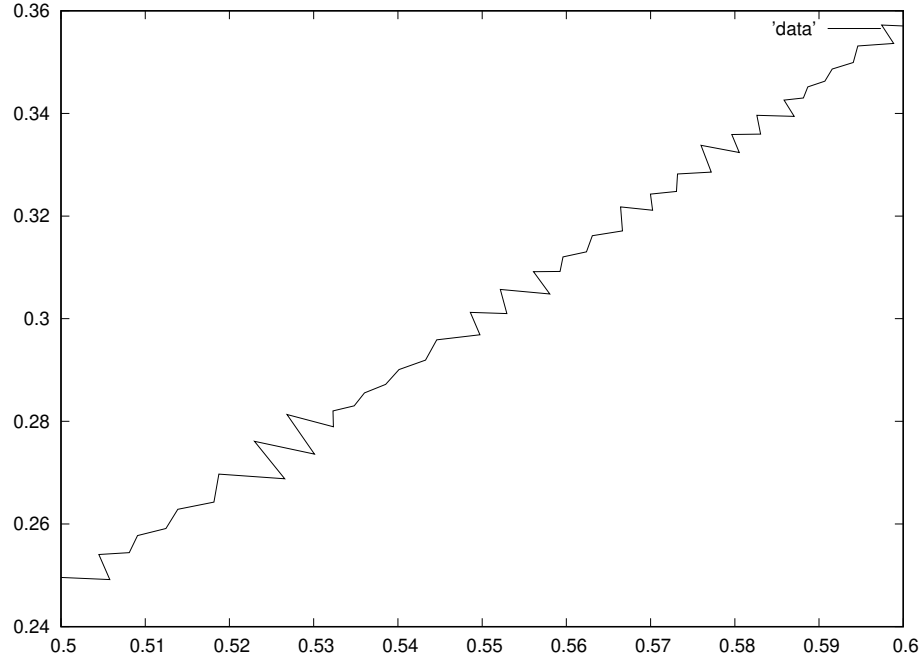


**Example 17.2:** Here is the gradient descent method with backtraching,  $\alpha = 0$ , applied to the Rosenbrock function  $R_2(x,y)$  (initial guess is  $\mathbf{x} = \begin{bmatrix} 0 & 1 \end{bmatrix}^T$ ):

```
import numpy as np
def f(x):
    return (x[0][0] - 1.0)**2 + 100. * (x[1][0] - (x[0][0])**2)**2
def grad_f(x):
    x21, grad_f = x[1][0] - (x[0][0])**2, np.array([[2. * (x[0][0] - 1.0)], [0.]])
    grad_f[0][0], grad_f[1][0] = grad_f[0][0] - 400. * x[0][0] * x21, 200. * x21
    return grad_f

x, old_F, F, dt = np.array([[0.], [1.]]), 0., 101., 0.01
while (abs(F - old_F) > 1.e-10):
    print(x[0][0], x[1][0], F, dt)
    old_x, old_F, F_x = x, F, grad_f(x)
    x, dt = x - dt * F_x, 1.1 * dt
    F = f(x)
    if (F > old_F):
        x, F, old_F, dt = old_x, old_F, old_F + 1., 0.5 * dt
```





**Example 17.3:** Consider  $f(x, y) = x^2 + Ay^2$ , with  $A \geq 1$ . Let us apply to it the gradient descent method with exact line search. One iteration of the descent method consists in the mapping

$$\nabla f(x, y) = 2 \begin{bmatrix} x \\ Ay \end{bmatrix}, \quad t_* = \arg \min_t \left( (x + tx)^2 + A(y + Aty)^2 \right) = -\frac{x^2 + A^2y^2}{x^2 + A^3y^2}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} \mapsto \begin{bmatrix} x + t_*x \\ y + At_*y \end{bmatrix} = \frac{(A-1)xy}{x^2 + A^3y^2} \begin{bmatrix} A^2y \\ -x \end{bmatrix} \mapsto \frac{(A-1)^2Ax^2y^2}{(x^2 + A^3y^2)(x^2 + Ay^2)} \begin{bmatrix} x \\ y \end{bmatrix}$$

In particular,

$$\begin{bmatrix} A \\ \pm 1 \end{bmatrix} C \mapsto \frac{A-1}{A+1} \begin{bmatrix} A \\ \mp 1 \end{bmatrix} C$$

The rate of decrease of the variables  $x, y$  (the minimum is at  $x_* = y_* = 0$ ) depends just on the ratio of  $x$  and  $y$ , and is minimal at  $x = \pm Ay$ . When  $A$  is large, the decrease of  $x$  and  $y$  is slow (as  $(A-1)/(A+1) \approx 1$ ).

## Problems and exercises

1. Consider a function  $H_2(x, y) = 50\sqrt{g(y^2 - x^2)} + (x - 10)^2 + y^2$ , where  $g(x) = \sqrt{x^2 + 1} + x$ . Find the minimum of  $H_2$  by the gradient descent method, starting from  $(x, y) = (-50, 40)$ .
2. Consider a function  $V_2(x, y) = (x + 3)^2 + y^2 e^{-2x}$ . Find the minimum of (a)  $V_2(x, y)$  and (b)  $W_2(x, z) := V_2(x, y = z/20)$  by the gradient descent method, starting from  $(x, y) = (0, 1)$  or  $(x, z) = (0, 20)$ . (c) The part (b) can be considered as an application of the steepest descent method to  $V_2$ . What norm  $\|\Delta \mathbf{x}\|$  is used?

## 18 Newton's method

Consider a twice continuously differentiable function  $f : \mathbf{R}^n \rightarrow \mathbf{R}$  that we want to minimize. At the position of minimum  $\mathbf{x}_*$  we have  $\nabla f(\mathbf{x}_*) = \mathbf{0}$ . We may think about this equation as a system of equations for the components of vector  $\mathbf{x}$ , and then try to solve it by Newton–Raphson method

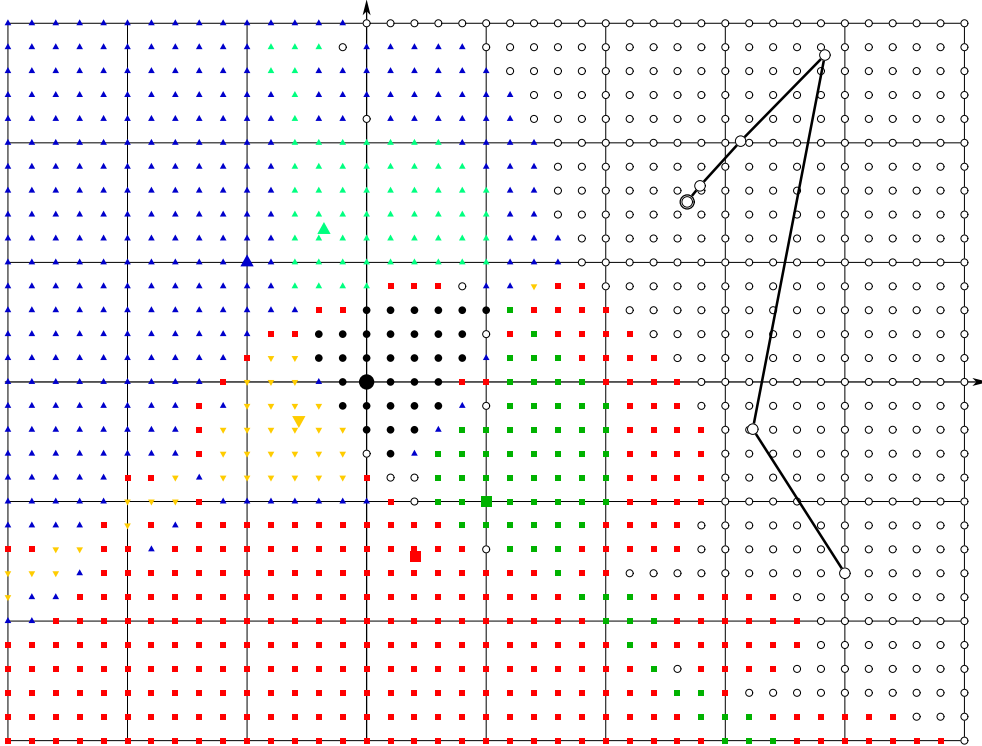
$$\mathbf{x} \mapsto \mathbf{x} - (\nabla^2 f)^{-1} \nabla f$$

Such updates of the position  $\mathbf{x}$  is so called *pure Newton* method.

Interpretations: 1) solution of the system  $\nabla f(\mathbf{x}_*) = \mathbf{0}$  by Newton–Raphson method; 2) minimization of local quadratic approximation; 3) steepest descent method with weighted norm  $\|\hat{W}\mathbf{x}\|_2$ , where  $\hat{W}^T \hat{W} = \nabla^2 f$ .

As the solutions of  $\nabla f(\mathbf{x}_*) = \mathbf{0}$  are not necessarily local minima, the pure Newton method could converge to local maxima, saddle points, *etc.*

**Example 18.1:** Here is the pure Newton method applied to  $G_2(x, y)$ . In the plot below it is shown for a grid of initial vectors  $\begin{bmatrix} x & y \end{bmatrix}^T$  where, if starting from them, the pure Newton method converges to. The method more or less converges to the closest point where  $\nabla G_2 = \mathbf{0}$ , and this point could be local minimum (open circle, blue triangle, and red square), maximum (black circle), saddle point (cyan and orange triangles, and green square). Larger shapes show the position of the corresponding points with  $\nabla G_2 = \mathbf{0}$ , while small shapes correspond to positions starting from which the Newton's method converges to a corresponding zero gradient point.



The pure Newton method is invariant under affine transformations: Let the vectors  $\mathbf{x}$  and  $\mathbf{y}$  be connected by  $\mathbf{x} = \hat{T}\mathbf{y}$ . Consider a function  $f(\mathbf{x})$  and its deformation  $g(\mathbf{y}) = f(\hat{T}\mathbf{y})$ . We have

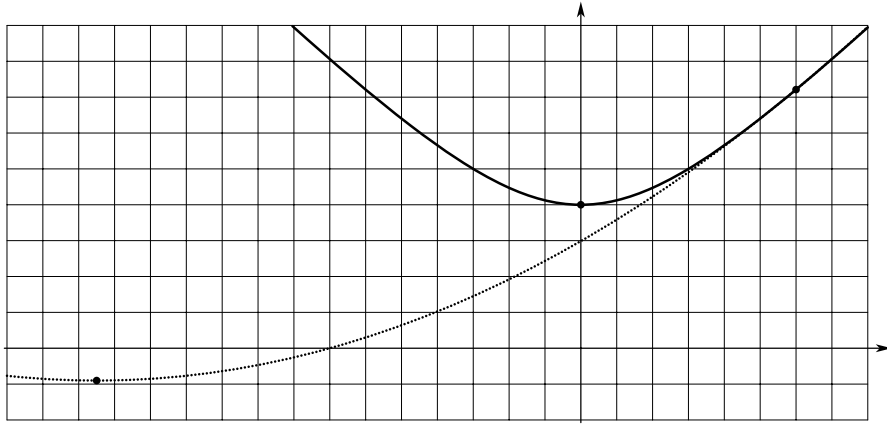
$$\frac{\partial g(\mathbf{y})}{\partial y_i} = \sum_j \frac{\partial f(\mathbf{x})}{\partial x_j} \bigg|_{\mathbf{x}=\hat{T}\mathbf{y}} \cdot \underbrace{\frac{\partial x_j}{\partial y_i}}_{T_{ji}} \quad \text{or} \quad \nabla_{\mathbf{y}} g(\mathbf{y}) = \hat{T}^T \nabla_{\mathbf{x}} f(\mathbf{x})$$

$$\frac{\partial g(\mathbf{y})}{\partial y_i \partial y_j} = \sum_k \frac{\partial f(\mathbf{x})}{\partial x_k \partial x_l} \bigg|_{\mathbf{x}=\hat{T}\mathbf{y}} \cdot \underbrace{\frac{\partial x_k}{\partial y_i}}_{T_{ki}} \underbrace{\frac{\partial x_l}{\partial y_j}}_{T_{lj}} \quad \text{or} \quad \nabla_{\mathbf{y}}^2 g(\mathbf{y}) = \hat{T}^T \nabla_{\mathbf{x}}^2 f(\mathbf{x}) \hat{T}$$

$$\mathbf{x} = \hat{T}\mathbf{y} \mapsto \hat{T} \left( \mathbf{y} - (\nabla_{\mathbf{y}}^2 g(\mathbf{y}))^{-1} \nabla_{\mathbf{y}} g(\mathbf{y}) \right) = \mathbf{x} - \hat{T} \left( \hat{T}^T \nabla_{\mathbf{x}}^2 f(\mathbf{x}) \hat{T} \right)^{-1} \hat{T}^T \nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{x} - (\nabla_{\mathbf{x}}^2 f(\mathbf{x}))^{-1} \nabla_{\mathbf{x}} f(\mathbf{x})$$

**Example 18.2:** Consider  $f(x) = \sqrt{x^2 + 1}$ . Then the pure Newton updates would be

$$f'(x) = \frac{x}{\sqrt{x^2 + 1}}, \quad f''(x) = \frac{1}{(x^2 + 1)^{3/2}}, \quad x \mapsto x - \frac{x/\sqrt{x^2 + 1}}{1/(x^2 + 1)^{3/2}} = x - (x^2 + 1)x = -x^3$$



Whenever  $|x| > 1$ , the next iteration of the pure Newton method is going to drive  $x$  further from the minimum of  $f$  at  $x_* = 0$ .

In order to improve the reliability of the Newton method, ...

*Damped Newton method:*  $\mathbf{x} \mapsto \mathbf{x} - t(\nabla^2 f)^{-1} \nabla f$ , where  $t$  is obtained from line search. It is a descent method, there the direction of search is obtained from the Newton method:  $\Delta \mathbf{x} = -(\nabla^2 f)^{-1} \nabla f$ .

*Levenberg–Marquardt algorithm:*  $\mathbf{x} \mapsto \mathbf{x} - t(\nabla^2 f + \mu \hat{I})^{-1} \nabla f$ . In the limit  $\mu \rightarrow 0 / +\infty$  we reproduce Newton / gradient descent methods.

**Example 18.3:** Consider  $f(x) = ax + by + (cx^2 + y^2)/2$ . At  $x = y = 0$  we have<sup>5</sup>

$$\begin{aligned} \nabla f(x) &= \begin{bmatrix} a \\ b \end{bmatrix}, & \nabla^2 f(x) &= \begin{bmatrix} c & 0 \\ 0 & 1 \end{bmatrix}, & -(\nabla^2 f)^{-1} \nabla f &= - \begin{bmatrix} c & 0 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} a \\ b \end{bmatrix} = - \begin{bmatrix} a/c \\ b \end{bmatrix} \\ (\nabla f) \cdot \left( -(\nabla^2 f)^{-1} \nabla f \right) &= -\frac{a^2}{c} - b^2, & (\nabla f) \cdot \left( -(\nabla^2 f + \mu \hat{I})^{-1} \nabla f \right) &= -\frac{a^2}{\mu + c} - \frac{b^2}{1 + \mu} \end{aligned}$$

<sup>5</sup> You may think about this example as follows: The Hessian  $\nabla^2 f$  is symmetric, so it can be diagonalized. Here  $x$ - and  $y$ -axis are the directions of  $\nabla^2 f$ 's eigenvalues. Here all the Taylor series terms beyond quadratic ones are dropped. By rescaling the coordinates, we make the coefficient at  $y^2$  being  $1/2$ , i.e.,  $\partial^2 f / \partial y^2 = 1$ .

If the function  $f$  is non-convex,  $c$  could be negative, and then  $-a^2/c - b^2$  could end up being positive. The direction of search is wrong, as locally moving in this direction increases the function.

## 18.1 Quasi-Newton methods

Consider  $f : \mathbf{R} \rightarrow \mathbf{R}$ , and we want to find such  $x_*$  that  $f(x_*) = 0$ . The updates according to Newton–Raphson method are  $x_{k+1} := x_k - f(x_k)/f'(x_k)$ .

Imagine we don't want to calculate the derivative of function  $f$ . We can estimate it through finite differences. We can consider an update rule  $x_{k+1} = x_k - f(x_k) \cdot (x_k - x_{k-1}) / (f(x_k) - f(x_{k-1}))$ . Geometrically this corresponds to forming a line that goes through the points  $(x_{k-1}, f(x_{k-1}))$  and  $(x_k, f(x_k))$ , and then find where this line crosses zero. A method of finding a root of a function with this update rule is called *secant method*.

**Example 18.4** Consider we want to compute  $\sqrt{2}$ , so we construct the function  $f(x) = x^2 - 2$  and then find its root(s). Let us apply Newton–Raphson method and secant method, starting with  $x_0 = 0.4$  and  $x_1 = 2.7$ :

	Newton–Raphson	secant
$x_0$	0.4	0.4
$x_1$	$0.4 - (0.4^2 - 2)/0.8 = 2.7$	2.7
$x_2$	$929/540 \approx 1.720370370370370$	$154/155 \approx 0.9935483870967741$
$x_3$	$1446241/1003320 \approx 1.441455368177650$	$7258/5725 \approx 1.267772925764192$
$x_4$	$1.414470981367771$	$1446241/1003320 \approx 1.441455368177650$
$x_5$	$1.414213585796884$	$1.412741073918240$
$x_6$	$1.414213562373095$	$1.414199508244253$
$x_7$		$1.414213569693568$
$x_8$		$1.414213562373059$
$x_9$		$1.414213562373095$
$\sqrt{2}$	$1.414213562373095048801688724209698\dots$	

**Algorithm BFGS** (Broyden–Fletcher–Goldfarb–Shanno algorithm): Quasi-Newton algorithm with low rank updates of the Hessian approximation at each step.

start with  $k = 0$ , some initial guess  $\mathbf{x}_0$  and  $\hat{C}_0$  (e.g.,  $\hat{C}_0 = \hat{I}$ )

while  $(\|\nabla f(\mathbf{x}_k)\| > \epsilon)$  do

    pick direction  $\Delta \mathbf{x}_k := -\hat{C}_k \nabla f(\mathbf{x}_k)$

    line search: choose step size  $t$

        backtracking line search: start with some not too small  $t$ , then reduce  $t$   
        (e.g.,  $t \leftarrow \beta t$ ) until  $f(\mathbf{x}_k + t\Delta \mathbf{x}_k) < f(\mathbf{x}_k) + \alpha \nabla f(\mathbf{x}_k) \cdot (t\Delta \mathbf{x}_k)$ ,  $0 \leq \alpha < 1$

    update:  $\mathbf{x}_{k+1} := \mathbf{x}_k + (d_k := t\Delta \mathbf{x}_k)$

$\mathbf{g}_k := \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$

$\hat{C}_{k+1} := \left( \hat{I} - \frac{\mathbf{d}_k \mathbf{g}_k^T}{\mathbf{g}_k^T \mathbf{d}_k} \right) \hat{C}_k \left( \hat{I} - \frac{\mathbf{g}_k \mathbf{d}_k^T}{\mathbf{g}_k^T \mathbf{d}_k} \right) + \frac{\mathbf{d}_k \mathbf{d}_k^T}{\mathbf{g}_k^T \mathbf{d}_k}$

$\hat{C}$  is an approximation of inverse Hessian,  $\hat{C} \approx (\nabla^2 f)^{-1}$

        and  $\hat{C}_{k+1}$  is chosen from the condition  $\mathbf{d}_k = \hat{C}_{k+1} \mathbf{g}_k$ , which is  
        an approximation of  $\nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k) \approx (\nabla^2 f) \cdot (\mathbf{x}_{k+1} - \mathbf{x}_k)$

$k \leftarrow k + 1$

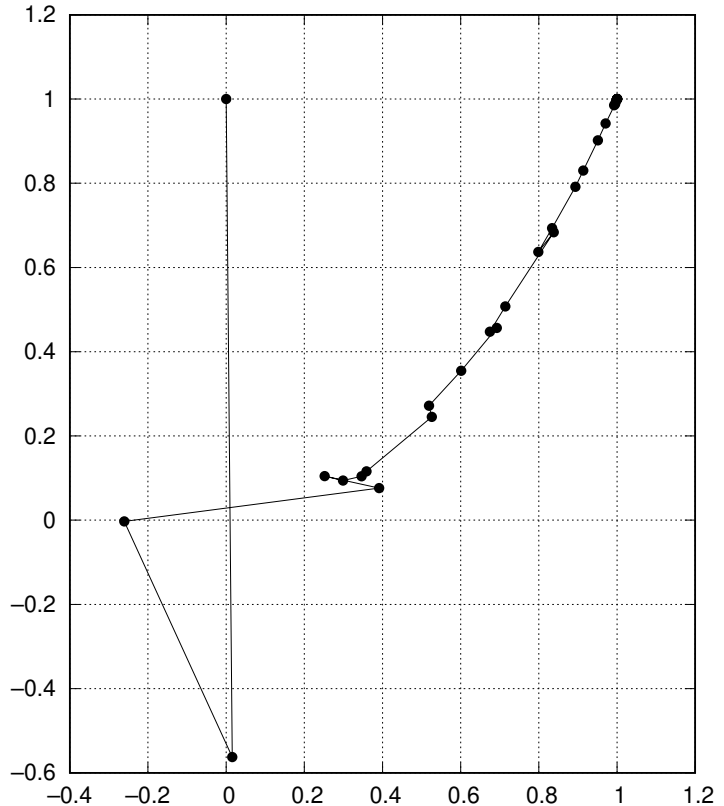
return  $\mathbf{x}_{\text{last}}$

**Example 18.5** Let us apply BFGS method to  $R_2(x,y)$ . Here is a MATLAB script:

```
function [x] = BFGS_R2(x)
    f = @(x) ((x(1) - 1)^2 + 100 * (x(2) - (x(1))^2)^2);
    df = @(x) (2 * [x(1) - 1; 0] + 200 * (x(2) - (x(1))^2) * [-2 * x(1); 1]);
    F = f(x); dF = df(x); C = eye(length(x));
    while (norm(dF) > 1.e-12)
        d = -C * dF;
        while (f(x + d) >= F)
            d = d / 2.;
        if (norm(d) < eps)
            C = eye(length(x)); d = -dF;
        end
        end
        x = x + d; F = f(x); new_dF = df(x); g = new_dF - dF; dF = new_dF;
        rho = 1. / (g' * d); mu = rho * (1. + rho * (g' * C * g));
        C = C - rho * (d * (g' * C) + (C * g) * d') + mu * d * d';
    end
```

```
octave:1> BFGS_R2([0; 1])
```

```
0.          1.
0.015625    -0.5625
-0.2605890939040998 -0.002902255146133292
0.3913828287588037  0.07627178330541139
0.2521830924078697  0.1044103708834347
0.2988661772597083  0.09405969456641125
0.3466003737396083  0.1040695448518871
0.3592647314084498  0.1156911711064965
0.5263173496617095  0.2453304005070866
0.5188182427260010  0.2716690732076990
0.6011469854117857  0.3548731818406570
0.6925269268918063  0.4563841342293742
0.6747491664659666  0.4475149568539881
0.7143015319640732  0.5073274307022437
0.8381443235933718  0.6837312358530869
0.7986209536630781  0.6368555689833639
0.8340051975976166  0.6934920772241859
0.8937025247897291  0.7914764619245512
0.9136520114636923  0.8303169726010279
0.9509644199894306  0.9019550045848869
0.9705682730518248  0.9421854653766993
0.9950326701317903  0.9879595713889101
0.9926616882035864  0.9851558833421766
0.9977406839316127  0.9954588419091560
0.9999291150286614  0.9998429509407731
0.9999963589854302  0.9999925797940143
1.000000030472313   1.000000059038727
0.9999999999259571  0.9999999998718994
0.999999999998126   0.999999999995955
1.000000000000000   1.000000000000001
```

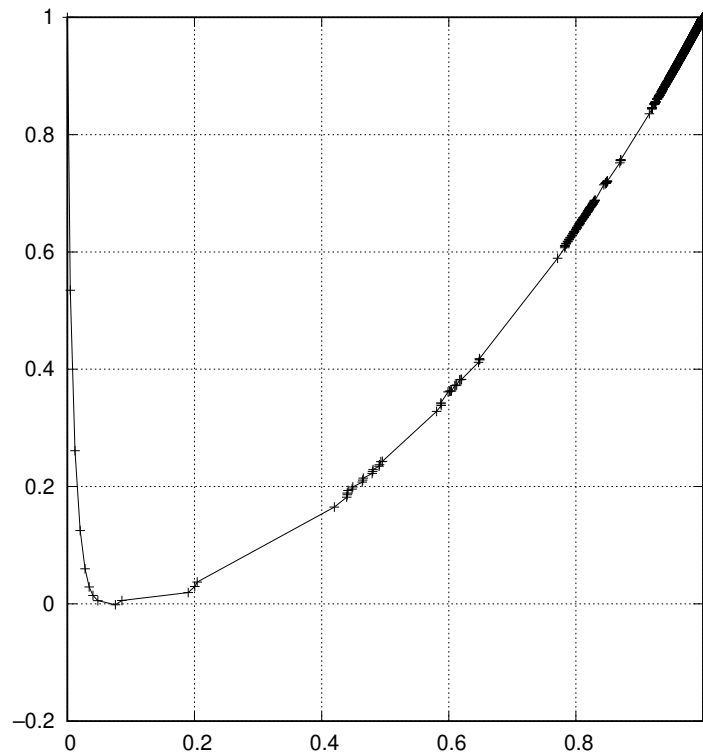


```
import numpy as np
def f(x):
    return (x[0] - 1.)**2 + 100. * (x[1] - (x[0])**2)**2
def grad_f(x):
    x21, grad_f = x[1] - (x[0])**2, np.array([2. * (x[0] - 1.), 0.])
    grad_f[0], grad_f[1] = grad_f[0] - 400. * x[0] * x21, 200. * x21
    return grad_f
```

```

x, alpha, mu = np.array([0., 1.]), 0.75, 0.1
old_x, grad = x, grad_f(x)
while (max(abs(grad[0]), abs(grad[1])) > 1.e-10):
    print(x[0], x[1], f(x))
    old_x, x = x, x + mu * (x - old_x)
    grad = grad_f(x)
    d = -grad
    while (f(x) + alpha * np.inner(grad, d) < f(x + d)):
        d = 0.7 * d
    x = x + d

```



## Problems and exercises

1. Consider a function  $H_2(x, y) := \exp(8x - 13y + 21) + \exp(21y - 13x - 34) + 0.0001 \exp(x + y)$ . Is it convex/strictly convex? Minimize it, *i.e.*, find  $(x_*, y_*) = \arg \min_{(x, y)} H_2(x, y)$ .
2. Minimize a function  $J_2(x, y) := 3xy - 2y + 1000(x^2 + y^2 - 1.1) \exp(10(x^2 + y^2 - 1))$ .
3. Consider a 99-dimensional vector  $\mathbf{x}$  with components  $x_1, x_2, \dots, x_{99}$ . For convenience, the dummy components  $x_0 = -1$  and  $x_{100} = 1$  are introduced, but  $x_0$  and  $x_{100}$  are not variables in the optimization problem below. Minimize a function

$$E_{99}(\mathbf{x}) := \frac{1}{2} \sum_{i=0}^{99} (x_{i+1} - x_i)^2 + \frac{1}{16} \sum_{i=1}^{99} (1 - x_i^2)^2, \quad \mathbf{x}_* = \arg \min_{\mathbf{x}} E_{99}(\mathbf{x})$$

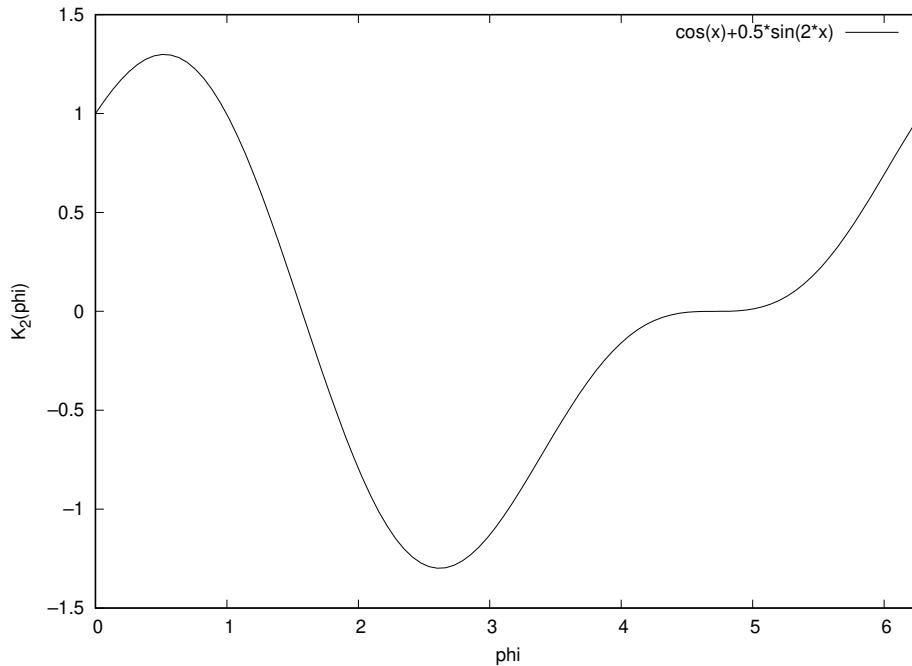
Find  $E_{99}(\mathbf{x}_*)$ . Plot the vector  $\mathbf{x}_*$  (how the component  $x_{*i}$ ,  $0 \leq i \leq 100$ , changes with  $i$ ).

## 19 Equality constrained minimization

Consider we want to minimize a function  $f(\mathbf{x})$ ,  $\mathbf{x} \in \mathbf{R}^n$ , subject to a set of equality constraints  $h_i(\mathbf{x}) = 0$ ,  $i = 1, 2, \dots, n_{\text{eq}}$ .<sup>6</sup>

One method is to parametrize the set of points  $\mathbf{x}$  satisfying the equality constraints by  $n - n_{\text{eq}}$  coordinates  $\mathbf{y} \in \mathbf{R}^{n-n_{\text{eq}}}$ , and then solve unconstrained optimization problem in  $\mathbf{y}$  variable.

**Example 19.1:** Let us minimize  $K_2(x, y) = x + xy$  subject to  $h_1(x, y) = x^2 + y^2 - 1 = 0$ . We can parametrize the set [of points satisfying the constraint]  $x^2 + y^2 = 1$  as  $(\cos \varphi, \sin \varphi)$ ,  $0 \leq \varphi < 2\pi$ . Then we have  $K_2(x, y) \rightarrow K_2(\varphi) = \cos \varphi + \cos \varphi \sin \varphi = \cos \varphi + \frac{1}{2} \sin 2\varphi$ . The minimum of  $K_2(\varphi)$  is at  $\varphi_* = 5\pi/6$ , which corresponds to  $x_* = -\sqrt{3}/2$ ,  $y_* = 1/2$ ,  $p_* = K_2(x_*, y_*) = -3\sqrt{3}/4$ .



**Example 19.1** continued: Let us introduce the Lagrangian and the dual function

$$\mathcal{L}(x, y; \mathbf{v}) = x + xy + \mathbf{v}(x^2 + y^2 - 1), \quad g(\mathbf{v}) = \inf_{x, y} \mathcal{L}(x, y; \mathbf{v}) = \begin{cases} -4\mathbf{v}^3/(4\mathbf{v}^2 - 1), & \mathbf{v} > 1/2 \\ -\infty, & \mathbf{v} \leq 1/2 \end{cases}$$

```
[...]/teaching/2020-1/math_575b/notes/Octave$ cat Lagrangian_K_2.m
function [grad_L] = Lagrangian_K_2(x)
    grad_L = [1 + x(2); x(1); 0.] + x(3) * [2. * x(1); 2. * x(2); 0.];
    grad_L(3) = (x(1))^2 + (x(2))^2 - 1.;

[...]/teaching/2020-1/math_575b/notes/Octave$ octave-cli
GNU Octave, version 4.4.1
[... copyright notice and links ...]
octave:1> format long; format compact
octave:2> newton(@Lagrangian_K_2, [0.2; 0.; 0.])'
counter = 34
ans =
```

<sup>6</sup>If we want the optimization problem to be convex, then  $f$  should be convex, while the set of points satisfying the equality constraints should be convex too, *i.e.*, it should be flat, and constraints could be written as linear ones.

```

8.660254037844387e-01    5.0000000000000001e-01    -8.660254037844385e-01

octave:3> newton(@Lagrangian_K_2, [-0.3; 0.; 0.])'
counter = 32
ans =
-8.660254037844431e-01    5.00000000000000023e-01    8.660254037844236e-01

octave:4> newton(@Lagrangian_K_2, [0.; 0.; 0.])'
warning: matrix singular to machine precision
warning: called from
    newton at line 14 column 7
counter = 2
ans =
1.192092909718666e-08    -1.0000000000000000e+00    5.960464637411177e-09

octave:5> newton(@Lagrangian_K_2, [-0.8; 0.3; 0.])'
counter = 5
ans =
-8.660254039596367e-01    5.0000000001235305e-01    8.660254033776313e-01

```

The maximum of  $g(\mathbf{v})$  happens at  $\mathbf{v} = \sqrt{3}/2$ , with  $d_* = g(\mathbf{v}_*) = -2(\sqrt{3}/2)^3 = -3\sqrt{3}/4 = p_*$ .

**Example 19.2:** Let us minimize  $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\hat{Q}\mathbf{x} - \mathbf{x}^T\mathbf{r}$  with the condition  $\hat{A}\mathbf{x} = \mathbf{b}$ . We form Lagrangian  $\mathcal{L}(\mathbf{x}, \mathbf{v}) = \frac{1}{2}\mathbf{x}^T\hat{Q}\mathbf{x} - \mathbf{x}^T\mathbf{r} + \mathbf{v}^T(\hat{A}\mathbf{x} - \mathbf{b})$ . Equating the gradient of  $\mathcal{L}$  with respect to  $\mathbf{x}$  and  $\mathbf{v}$  to zero, we get

$$\begin{array}{l} \hat{Q}\mathbf{x} + \hat{A}^T\mathbf{v} = \mathbf{r} \\ \hat{A}\mathbf{x} = \mathbf{b} \end{array} \quad \begin{array}{|c|c|} \hline \hat{Q} & \hat{A}^T \\ \hline \hat{A} & \hat{O} \\ \hline \end{array} \begin{array}{|c|} \hline \mathbf{x} \\ \hline \mathbf{v} \\ \hline \end{array} = \begin{array}{|c|} \hline \mathbf{r} \\ \hline \mathbf{b} \\ \hline \end{array}$$

It is an  $(n + n_{\text{eq}}) \times (n + n_{\text{eq}})$  system of linear equations, solving which would give the position of optimum  $\mathbf{x}_*$  that automatically satisfy the condition  $\hat{A}\mathbf{x}_* = \mathbf{b}$ , due to the lower part of the system.

**Example 19.3:** Consider  $R_2(x, y)$  with the condition  $y = 2x - 1$ .

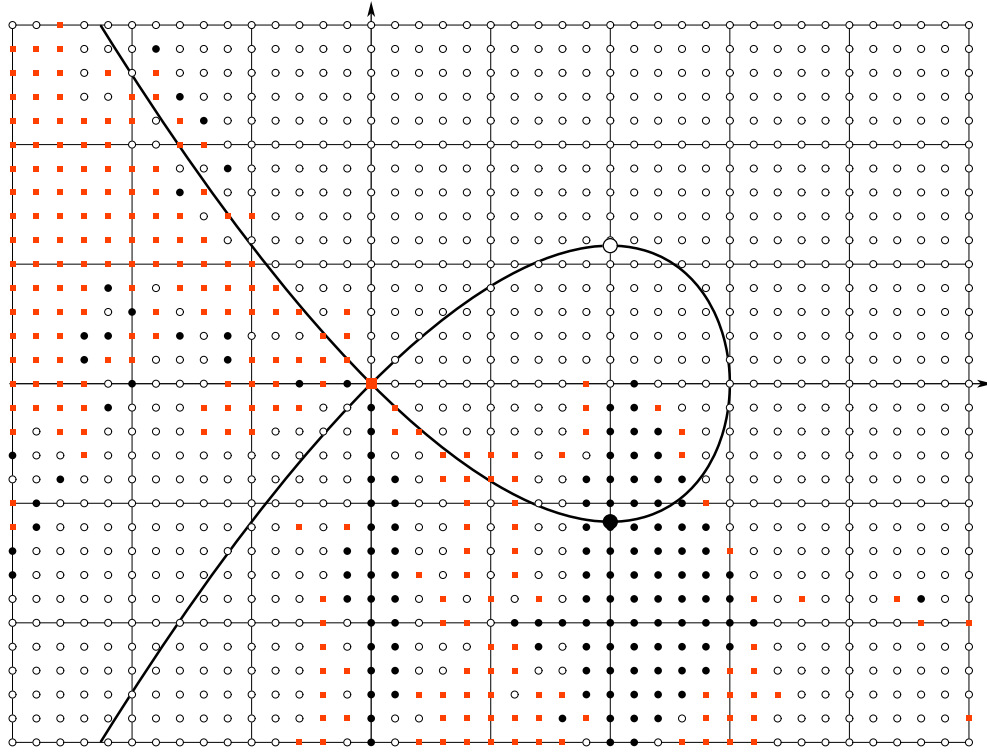
```

function [x] = eq_R2(x)
    flag = 1;
    while (flag > 0)
        G = 2 * [x(1) - 1; 0; 0] + 200 * (x(2) - (x(1))^2) * [-2 * x(1); 1; 0];
        % Hessian of R_2(x, y) % + nu * (y - 2x - 1)
        H = zeros(3);
        H(1, 1) = 2. - 400. * x(2) + 1200 * (x(1))^2;
        H(1, 2) = -400 * x(1); H(2, 1) = H(1, 2);
        H(2, 2) = 200.;
        H(3, 1) = -2.; H(1, 3) = -2.;
        H(3, 2) = 1.; H(2, 3) = 1.;
        xold = x; x = x - H \ G; x(3) = 0.;
        x'
        if (norm(x - xold) < 1.e-8)
            flag = 0;
        end
    end
end

```

**Example 19.4:** Consider the function  $L_2(x,y) = (x-1)^2 + (y-1)^2$  subject to condition  $y^2 = x^2 - 2x^3/3$ .

The plot below is the map where the primal-dual Newton method (which is an application of Newton method to the Lagrangian  $\mathcal{L}(\mathbf{x}, \mathbf{v})$ ) converges to. There are 3 points (shown by larger shapes) at which the gradient of  $L_2$  along the zero level curve of the equality constraint is zero:  $(1, \pm 3^{-1/2})$  and  $(0,0)$  (there are 2 directions of the level curve at this point, the gradient is zero along only one of the directions). The small shapes correspond to positions  $(x,y,\mathbf{v} = 0)$  starting from which the Newton's method converges to a corresponding zero gradient point.



Let us consider another idea to solve an equality constrained optimization problem. Consider we want to minimize  $f(\mathbf{x})$  subject to  $n_{\text{eq}}$  equality constraints  $\mathbf{h}(\mathbf{x}) = \mathbf{0}$ . We form a function

$$f_t(\mathbf{x}) := f(\mathbf{x}) + t \sum_{i=1}^{n_{\text{eq}}} h_i^2(\mathbf{x})$$

and minimize it for different values of  $t$ . Obviously, for the points on which  $\mathbf{h}(\mathbf{x}) = \mathbf{0}$  we have  $f_t(\mathbf{x}) = f(\mathbf{x})$ . When  $t$  is large, then any non-zero values in  $\mathbf{h}(\mathbf{x})$  would produce too large value of  $f_t(\mathbf{x})$ , so the large  $t$  is, the better the solution of unconstrained minimization of  $f_t(\mathbf{x})$  approximates the solution of the equality constrained minimization.

When  $t$  is large, the function  $f_t(\mathbf{x})$  has narrow valleys along the zero set of  $\mathbf{h}(\mathbf{x})$ , the fact needed to be taken into account.

**Example 19.4, continued:** We have  $f_t(x,y) = (x-1)^2 + (y-1)^2 + t(y^2 - x^2 + 2x^3/3)^2$ . Let us apply damped Newton method to it:

```
function [X] = newton_L2_eq(X, t)
```

```

h = @(x) ((x(2))^2 - (x(1))^2 * (1. - 2. * x(1) / 3.));
f = @(x) ((x(1) - 1.)^2 + (x(2) - 1.)^2 + t * (h(x))^2);
while (1 > 0)
    F = f(X);
    x = X(1); y = X(2); H = y^2 - x^2 + 2. * x^3 / 3.;
    dF = 2. * [x - 1.; y - 1.] + 4. * t * H * [x^2 - x; y];
    if (norm(dF) < 1.e-8)
        break;
    end
    ddF = [2. + 8. * t * (x^2 - x)^2 + 4. * t * (2. * x - 1.) * H, 0.; 0., 0.];
    ddF(1, 2) = 8. * t * (x^2 - x) * y; ddF(2, 1) = ddF(1, 2);
    ddF(2, 2) = 2. + 8. * t * y^2 + 4. * t * H;
    d = -(ddF \ dF); flag = 0;
    while (f(X + d) >= F)
        d = d / 2.;
        if (norm(d) < eps)
            if (flag == 1)
                break;
            end
            d = -dF; flag = 1;
        end
        end
    X = X + d;
    X'
end

```

## Problems and exercises

1. Consider the function  $M_2(x, y) = x^2 + y^2$ . Minimize and maximize it subject to the condition  $(6x + 29)^2(x - 1)^2 + 12(6x + 31)(x - 1)y^2 + 36y^4 = 0$ .

## 20 Example V.2: image restoration<sup>7</sup>

Consider you have an “image”  $v$  (it is either a function  $v(x)$  or a two-dimensional picture  $v(x, y)$ , *etc.*). The image is distorted by noise,  $w = v + \xi$ . Our task is to remove noise. We can pose it as a following optimization problem

$$u_* := \arg \min_u \left( \frac{1}{2} \|u - w\|_2^2 + \lambda \|\nabla u\|_p^v \right)$$

The 1<sup>st</sup> term tries to get most of the signal  $w$  that we have. The 2<sup>nd</sup> term tries to make the restored image “smooth”, as we think of the noise  $\xi$  to be high frequency, not correlated from pixel to pixel, *etc.* If  $\lambda = 0$ , then we just have  $u = w$ .

Consider the case  $p = v = 2$ . Then  $u$  is the result of minimization of quadratic function. In the case of one-dimensional signal we have

$$u_* := \arg \min_u \int dx \underbrace{\left( \frac{1}{2} (u - w)^2 + \lambda (u'(x))^2 \right)}_{\mathcal{L}(u, u')}$$

---

<sup>7</sup> Adapted from [Cal20, Sec. 3.6].

The Euler–Lagrange equation would be linear:  $(\partial \mathcal{L} / \partial u')' = \partial \mathcal{L} / \partial u$  or  $(2\lambda u')' = 2\lambda u'' = u - w$ . If we do the Fourier transform, we get  $\hat{w}(k) = (1 + 2\lambda k^2)\hat{u}(k)$  (even in higher dimensions), or  $u$  is the convolution of  $w$  with a certain kernel, namely  $\exp(-|x|/\sqrt{2\lambda})/\sqrt{8\lambda}$ .

Let us think about an image as a graph, with vertices being the pixels, while edges indicate that its endpoints are neighbors. The optimization problem looks simpler (*i.e.*, more local) for  $v = p$ :

$$u := \arg \min_u \left( \frac{1}{2} \sum_v (u_v - w_v)^2 + \lambda \sum_{e=v_1 v_2} |u_{v_1} - u_{v_2}|^p \right)$$

(It is possible to assign different values of  $\lambda$  to edges, if desired.)

The choice  $p[=v] = 1$  is better. Let us also smooth  $|u|$  as  $\sqrt{\varepsilon^2 + u^2}$  with some small  $\varepsilon$ . The problem with two pixels and one edge would look like

$$(u_{*1}, u_{*2}) := \arg \min_{(u_1, u_2)} \left( \frac{1}{2} (u_1 - w_1)^2 + \frac{1}{2} (u_2 - w_2)^2 + \lambda \sqrt{\varepsilon^2 + (u_1 - u_2)^2} \right) = \arg \min_{(u_1, u_2)} f(u_1, u_2)$$

If we do the minimization by ODE version of gradient, we get the equation of motion descent

$$\frac{du_1}{dt} = -\frac{\partial f(u_1, u_2)}{\partial u_1} = w_1 - u_1 - \lambda \frac{u_1 - u_2}{\sqrt{\varepsilon^2 + (u_1 - u_2)^2}} \approx w_1 - u_1 - \frac{\lambda}{\varepsilon} \cdot (u_1 - u_2)$$

The last approximate equality is written for the case when  $u_1$  and  $u_2$  are close,  $(u_1 - u_2) \ll \varepsilon$ . (Similar equation can be written for  $u_2$ , and one can even write a closed equation for the difference  $u := u_1 - u_2$ .) In that case  $-d(du/dt)/du = f''(u) = 2\lambda/\varepsilon$  is large, and we need small step size (not greater than  $\varepsilon/\lambda$  if we use forward Euler method) in order for an explicit scheme for solving the ODE for  $u$  to be in its region of absolute stability.

Let us add some inertia to the ODE we are trying to solve:

$$m \frac{d^2 u}{dt^2} + \frac{du}{dt} = -f'(u) = -\frac{2\lambda u}{\sqrt{\varepsilon^2 + u^2}}$$

You may imagine a particle of mass  $m$  moving in the potential  $f(u)$ , while the term  $du/dt$  provides a friction force. (In the limit  $m, \lambda \rightarrow \infty$  with  $\lambda/m$  being fixed we get a Hamiltonian system with no dissipation.) This is similar to applying Polyak's heavy ball or Nesterov's fast gradient methods, instead of just gradient descent. For small  $u$  we have the system  $m\ddot{u} + \dot{u} + \lambda u/\varepsilon = 0$ . Now instead of solution  $u \propto e^{\gamma t}$  with  $\gamma = -2\lambda/\varepsilon$  (which bounds the time step by  $\Delta t \lesssim \varepsilon/\lambda$ ), we get  $\gamma = \pm \sqrt{1/4m^2 - 2\lambda/\varepsilon m} - 1/2m \approx \pm i \sqrt{2\lambda/\varepsilon m}$ , so we need  $\Delta t \lesssim \sqrt{\varepsilon m/\lambda}$ , *i.e.*,  $\Delta t$  is not much greater than the inverse frequency of oscillations of the particle near the bottom of the potential  $f(u)$ .

If one still applies forward Euler method, then in order for  $\gamma$  to be in the region of absolute stability, we need  $|1 + \gamma \Delta t| < 1$ , or  $(1 - \Delta t/2m)^2 + (\Delta t)^2(2\lambda/\varepsilon m - 1/4m^2) = 1 - \Delta t/m + 2(\Delta t)^2\lambda/\varepsilon m < 1$ , which gives  $\Delta t < \varepsilon/2\lambda$ . In order to be stable near the bottoms of  $\sqrt{\varepsilon^2 + u^2}$  and have  $\Delta t \gg \varepsilon/\lambda$ , we need to use a method of at least 2<sup>nd</sup> order of accuracy. Even then, if we are just worried about the speed of [linear] convergence in the small vicinity of the minimum of the function, where the quadratic approximation would already work, to have the largest gain [in decrease of the function] per step one would choose  $\Delta t \sim \varepsilon/\lambda$ , regardless of the numerical method for solving the system of ODEs.

The benefit of introducing the mass  $m$  is that [before entering the small vicinity of the minimum of the function] we descend along the narrow (as  $\varepsilon$  is small) valleys in a more decisive fashion.

The system of ODEs for the pixels' values looks like

$$\text{for all pixels } v \quad m \frac{d^2 u_v}{dt^2} + \frac{du_v}{dt} = w_v - u_v - \lambda \sum_{v'} \frac{u_v - u_{v'}}{\sqrt{\epsilon^2 + (u_v - u_{v'})^2}}$$

where the summation over  $v'$  goes over the pixels neighboring the pixel  $v$ .

**Example V.2.1:** Let us denoise an  $64 \times 64$  image of letter “U”. The appropriate values of  $\lambda$  are related to the typical pixels' values (from 0 to 255) — the value of  $\lambda$  should not be smaller than the magnitude of noise. The explicit midpoint (RK2) method was used. With  $\epsilon = 0.1$  (*i.e.*, smaller than our resolution of pixel's values, which are integers) the value of  $m \approx 0.2$  works well and allows the time steps larger than  $\epsilon/\lambda$ .

The noise increases from left to right, and the values of  $\lambda$  that reasonably denoise the image (*e.g.*,  $\lambda = 16$  for the left image (where standard deviation of noise is 17) and  $\lambda = 128$  for the right image (standard deviation of noise is 102) increase too.

When  $\lambda$  is very large, any changes in the image contribute greatly to the function being minimized, because of  $\lambda \|\nabla u\|_1$  term. The fading of the image with  $\lambda$  is visible for, *e.g.*,  $\lambda \geq 64$ . Another thing to notice is the jump of the pixels' values in between the top parts of letter “U” sides. This is because the length between the sides is shorter than the boundary of inner part of letter “U”, and the “optimal” reconstruction makes the value of the function smaller by funneling part of the change in image to shorter segment, thus reducing  $\lambda \|\nabla u\|_1$ . (The area in between the sides of “U” is not becoming dark because of  $\|u - w\|_2$  term which forces the reconstruction  $u$  to resemble the original image  $w$ ).

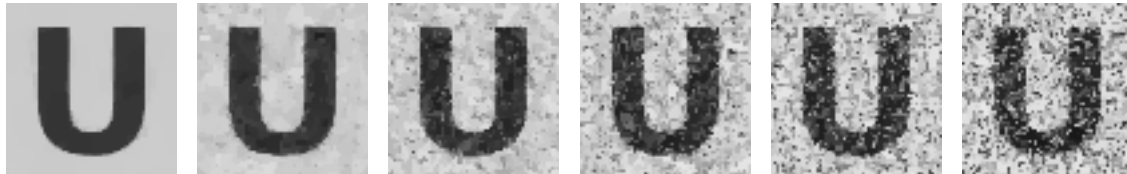
Original image (grayscale,  $64 \times 64$ , 8 bits color depth (values are from 0 to 255), letter “U”, 51 on the letter and 204 on the background), images with added noise (normally distributed with zero mean, independent from pixel to pixel, with standard deviation 17 (left), 34, 51, 68, 85, and 102 (right), if the value at the pixel after adding the noise becomes smaller than 0 or larger than 255, then it is clipped). Restorations for several values of  $\lambda$  are shown,  $\lambda = 4$  (top), 8, 16, 32, 64, 128, and 256 (bottom).



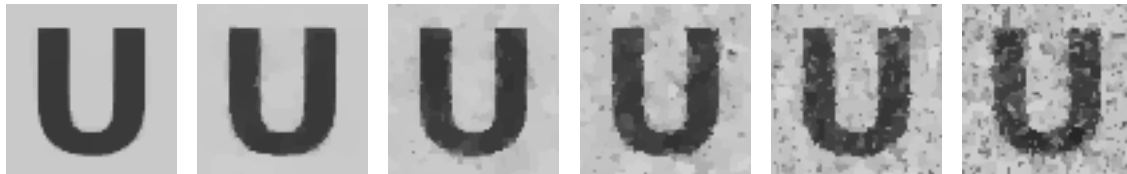
$\lambda = 4$



$\lambda = 8$



$\lambda = 16$



$\lambda = 32$



$\lambda = 64$



$\lambda = 128$



$\lambda = 256$

1. Consider the image below. Think of it as a matrix written on the right. If you use  $p = \epsilon = 0.1$ , and appropriate values of  $\alpha$  and  $\beta$ , you can produce the image below.



(The idea of barrier method is similar to the idea of introducing the auxiliary function  $f_t(\mathbf{x}) = f(\mathbf{x}) + t \sum_i h_i^2(\mathbf{x})$ .) Consider we want to minimize the function  $f(\mathbf{x})$  subject to inequality constraints  $g_i(\mathbf{x}) \leq 0$ . This problem is equivalent to [unconstrained] minimization of  $f(\mathbf{x}) + \sum_i I_-(g_i(\mathbf{x}))$ , where  $I_-(g)$  is an indicator function

Let us substitute the function  $I_-(g)$  by, e.g.,  $L_t(g) := -(1/t) \ln(-g)$  (and, e.g.,  $L_t(g) = +\infty$  if  $g \geq 0$ ) — the so called logarithmic barrier. The larger  $t$  is, the better  $L_t(g)$  resembles  $I_-(g)$ , so we can hope that the point of minimum of  $f_t(\mathbf{x}) := f(\mathbf{x}) + \sum_i L_t(g_i(\mathbf{x}))$  is close to the solution of our [constrained] optimization problem.

```
function [X] = newton_L2_ineq(X, t)
    g = @(x) ((x(2))^2 - (x(1))^2 * (1. - 2. * x(1) / 3.));
    f = @(x) ((x(1) - 1.)^2 + (x(2) - 1.)^2 - log(-g(x)) / t);
    while (1 > 0)
        F = f(X);
        x = X(1); y = X(2); G = y^2 - x^2 + 2. * x^3 / 3.;
        dF = 2. * [x - 1.; y - 1.] - 2. * [x^2 - x; y] / (t * G);
        if (norm(dF) < 1.e-8)
            break;
        end
        ddF = [0., 0.; 0., 2. + 4. * y^2 / (t * G^2) - 2. / (t * G)];
        ddF(1, 2) = 4. * (x^2 - x) * y / (t * G^2); ddF(2, 1) = ddF(1, 2);
        ddF(1, 1) = 2. + 4. * (x^2 - x)^2 / (t * G^2) - (4. * x - 2.) / (t * G);
        d = -ddF \ dF;
        while ((f(X + d) >= F) || (g(X + d) >= 0.))
            d = d / 2.;
        end
    end
end
```

```

    end
    X = X + d;
end

octave:1> format long; format compact
octave:2> newton_L2_ineq([0.5; 0.], 1.)'
ans =
    1.0000000000000117e+00    2.213803260348214e-01

octave:3> newton_L2_ineq([0.5; 0.], 10.)'
ans =
    1.0000000000000000e+00    4.879092568824754e-01

octave:4> newton_L2_ineq([0.5; 0.], 100.)'
ans =
    9.999999999999998e-01    5.659458736827376e-01

octave:5> newton_L2_ineq([1.; 0.5659458736827376], 100000.)'
ans =
    1.0000000000000000e+00    5.773384395149100e-01

octave:6> 1. / sqrt(3.)
ans =    5.773502691896258e-01

```

The larger is the value of  $t$ , the closer is the position of the minimum of  $f_t(x, y) = (x - 1)^2 + (y - 1)^2 - (1/t) \ln(x^2 - y^2 - 2x^3/3)$  to the exact position  $(x_*, y_*) = (1, 1/\sqrt{3})$ .

## Problems and exercises

1. Minimize the function  $L_2(x, y) = (x - 1)^2 + (y - 1)^2$  subject to  $y^2 + x^3 \leq 0$ .
2. Minimize the function  $C_{110}(x_1, x_2, \dots, x_{55}, y_1, y_2, \dots, y_{55}) := \sum_{i=1}^{56} (y_{i-1} + y_i)/2 = \frac{1}{2} + \sum_{i=1}^{55} y_i$  subject to 55 inequality constraints  $y_i \geq 0$ ,  $i = 1, 2, \dots, 55$ , and 56 equality constraints  $(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2 = 0.03^2$ ,  $i = 1, 2, \dots, 56$ . Here for convenience dummy [non-]variables  $x_0 = y_0 = 0$  and  $x_{56} = y_{56} = 1$  are introduced.<sup>8</sup>

## 22 Linear programming

**Example 22.1:** Consider a problem of minimizing a linear function, *e.g.*,  $-x - 2y$ , in the domain  $x \geq 0, y \geq 0, x + 3y \leq 9, x + y \leq 5$ .

```

[...]/teaching/2020-1/math_575b/notes/linear_programming$ cat example_1_simplex.c
#include <stdio.h>
#include <glpk.h>
int main(void) { glp_prob *lp; int ia[5], ja[5]; double x, y, ar[5];
    lp = glp_create_prob(); glp_set_obj_dir(lp, GLP_MIN);
    glp_add_rows(lp, 2); glp_add_cols(lp, 2);
    glp_set_col_bnds(lp, 1, GLP_LO, 0., 0.); /* 0 <= x */
    glp_set_col_bnds(lp, 2, GLP_LO, 0., 0.); /* 0 <= y */

```

<sup>8</sup> The problem is about the shape of a flexible chain of 56 segments with length 0.03 (so the chain's length is  $\sqrt{2} < L = 1.68 = 56 \cdot 0.03 < 2$ ) with its ends at the points (0,0) and (1,1) in  $xy$ -plane. The function  $C_{110}$  is the potential energy in the gravity field. The line  $y = 0$  is the ground level, and the chain can not go below it.

```

glp_set_obj_coef(lp, 1, -1.); glp_set_obj_coef(lp, 2, -2.); /* min -x - 2 y */

ia[1] = 1; ja[1] = 1; ar[1] = 1.; /* x + 3 y <= 9 */
ia[2] = 1; ja[2] = 2; ar[2] = 3.;
glp_set_row_bnds(lp, 1, GLP_UP, 0., 9.);
ia[3] = 2; ja[3] = 1; ar[3] = 1.; /* x + y <= 5 */
ia[4] = 2; ja[4] = 2; ar[4] = 1.;
glp_set_row_bnds(lp, 2, GLP_UP, 0., 5.);
glp_load_matrix(lp, 4, ia, ja, ar); glp_simplex(lp, NULL);

x = glp_get_col_prim(lp, 1); y = glp_get_col_prim(lp, 2);
printf("x = % 22.16e\ny = % 22.16e\n", x, y);
glp_delete_prob(lp); return 0; }
[...]/teaching/2020-1/math_575b/notes/linear_programming$ cc example_1_simplex.c
-lglpk ; ./a.out
GLPK Simplex Optimizer, v4.65
2 rows, 2 columns, 4 non-zeros
*      0: obj =   0.0000000000e+00 inf =   0.000e+00 (2)
*      2: obj =  -7.0000000000e+00 inf =   0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
x =   2.9999999999999996e+00
y =   2.0000000000000000e+00
[...]/teaching/2020-1/math_575b/notes/linear_programming$

```

Here is the same linear program solved by `scipy.optimize.linprog` from [SciPy](#):

```

[...]/teaching/2020-1/math_575b/notes/linear_programming$ cat example_1.py
from scipy.optimize import linprog
print(linprog([-1., -2.], A_ub = [[1., 3.], [1., 1.]], b_ub = [9., 5.]))
[...]/teaching/2020-1/math_575b/notes/linear_programming$ python3 example_1.py
con: array([], dtype=float64)
fun: -7.0
message: 'Optimization terminated successfully.'
nit: 3
slack: array([0., 0.])
status: 0
success: True
x: array([3., 2.])
[...]/teaching/2020-1/math_575b/notes/linear_programming$

```

**Example 22.2:** Consider fitting the cloud of 5 points  $\mathcal{S}_5 = \{(-2, -3), (-1, -1), (0, 5), (2, 5), (3, 1)\}$  by  $y = ax + b$  line, with  $\max_i |ax_i + b - y_i|$  being minimized. The problem of fitting in  $L^\infty$  sense could be written as a linear program

minimize  $d$  subject to  $-d \leq ax_i + b - y_i \leq d$  for all  $i$

```

[...]/teaching/2020-1/math_575b/notes/linear_programming$ cat example_2.c
#include <stdio.h>
#include <glpk.h>
#define MAT(I, J, A) ia[m] = I; ja[m] = J; ar[m] = A; m++;
int main(void) { glp_prob *lp; int i, k, m, ia[31], ja[31]; double a, b, ar[31];
double xy[5][2] = {{-2., -3.}, {-1., -1.}, {0., 5.}, {2., 5.}, {3., 1.}};
lp = glp_create_prob(); glp_set_obj_dir(lp, GLP_MIN);

```

```

glp_add_rows(lp, 10); glp_add_cols(lp, 3);
for (i = 1; i <= 3; i++) glp_set_col_bnds(lp, i, GLP_FR, 0., 0.);
glp_set_obj_coef(lp, 1, 0.); glp_set_obj_coef(lp, 2, 0.);
glp_set_obj_coef(lp, 3, 1.);

for (m = 1, i = 0; i < 5; i++) {
    k = 2 * i + 1; MAT(k, 1, xy[i][0]); MAT(k, 2, 1.); MAT(k, 3, -1.);
    glp_set_row_bnds(lp, k, GLP_UP, 0., xy[i][1]);
    k = 2 * i + 2; MAT(k, 1, xy[i][0]); MAT(k, 2, 1.); MAT(k, 3, 1.);
    glp_set_row_bnds(lp, k, GLP_LO, xy[i][1], 0.); }
glp_load_matrix(lp, m - 1, ia, ja, ar); glp_simplex(lp, NULL);

a = glp_get_col_prim(lp, 1); b = glp_get_col_prim(lp, 2);
printf("a = % 22.16e\nb = % 22.16e\n", a, b);
glp_delete_prob(lp); return 0; }
[...]/teaching/2020-1/math_575b/notes/linear_programming$ cc example_2.c -lglpk
[...]/teaching/2020-1/math_575b/notes/linear_programming$ ./a.out
GLPK Simplex Optimizer, v4.65
10 rows, 3 columns, 28 non-zeros
    0: obj = 0.000000000e+00 inf = 1.500e+01 (5)
    5: obj = 3.200000000e+00 inf = 0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
a = 8.00000000000000016e-01
b = 1.8000000000000000e+00
[...]/teaching/2020-1/math_575b/notes/linear_programming$

```

**Example 22.3:** Consider fitting the cloud of 5 points  $\mathcal{S}_5$  by  $y = ax + b$  line, with  $\sum_i |ax_i + b - y_i|$  being minimized. The problem of fitting in  $L^1$  sense could be written as a linear program

$$\text{minimize } \sum_i d_i \text{ subject to } -d_i \leq ax_i + b - y_i \leq d_i \text{ for all } i$$

```

[...]/teaching/2020-1/math_575b/notes/linear_programming$ diff -tyW 156 --suppre
ss-common-lines example_2.c example_3.c
glp_add_rows(lp, 10); glp_add_cols(lp, 3); |
glp_add_rows(lp, 10); glp_add_cols(lp, 7); |
for (i = 1; i <= 3; i++) glp_set_col_bnds(lp, i, GLP_FR, 0., 0.); |
for (i = 1; i <= 7; i++) glp_set_col_bnds(lp, i, GLP_FR, 0., 0.); |
glp_set_obj_coef(lp, 3, 1.); |
for (i = 3; i <= 7; i++) glp_set_obj_coef(lp, i, 1.); |
    k = 2 * i + 1; MAT(k, 1, xy[i][0]); MAT(k, 2, 1.); MAT(k, 3, -1.); |
    k = 2 * i + 1; MAT(k, 1, xy[i][0]); MAT(k, 2, 1.); MAT(k, i + 3, -1.); |
    k = 2 * i + 2; MAT(k, 1, xy[i][0]); MAT(k, 2, 1.); MAT(k, 3, 1.); |
    k = 2 * i + 2; MAT(k, 1, xy[i][0]); MAT(k, 2, 1.); MAT(k, i + 3, 1.); |
[...]/teaching/2020-1/math_575b/notes/linear_programming$ cc example_3.c -lglpk
[...]/teaching/2020-1/math_575b/notes/linear_programming$ ./a.out
GLPK Simplex Optimizer, v4.65
10 rows, 7 columns, 28 non-zeros
    0: obj = 0.000000000e+00 inf = 1.500e+01 (5)
    4: obj = 1.000000000e+01 inf = 0.000e+00 (0)
*    7: obj = 1.000000000e+01 inf = 0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
a = 2.0000000000000000e+00
b = 1.0000000000000000e+00
[...]/teaching/2020-1/math_575b/notes/linear_programming$

```

## Part VI

# Applied probability

## 23 Generating pseudo-random numbers

linear congruential generators, RANDU (bad pseudo-random numbers generator)

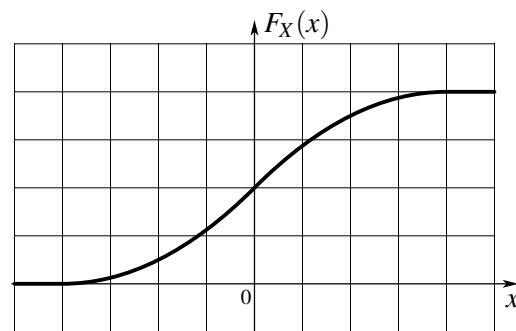
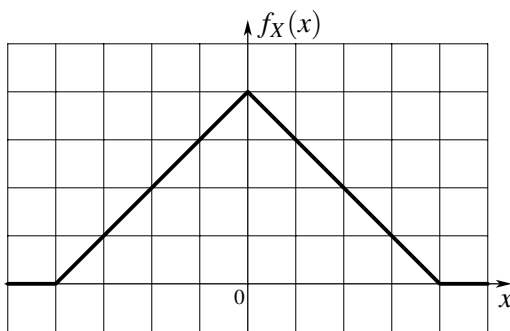
TestU01, KISS: a bit too simple

GNU GSL RNGs

From now on let us assume that we have an access to a generator of uniformly distributed on  $[0, 1)$  random numbers. Different trials of this generator are assumed to be independent.

**Example 23.1:** Let us construct a generator of random numbers with distribution function

$$f_X(x) = \begin{cases} 1+x, & -1 \leq x \leq 0 \\ 1-x, & 0 \leq x \leq 1 \\ 0, & \text{otherwise} \end{cases} \quad F_X(x) = \begin{cases} 0, & x \leq -1 \\ (1+x)^2/2, & -1 \leq x \leq 0 \\ 1 - (1-x)^2/2, & 0 \leq x \leq 1 \\ 1, & 1 \leq x \end{cases}$$



The 1<sup>st</sup> idea is to use the expression for  $F_X^{-1}$ :

$$x := \begin{cases} -1 + \sqrt{2u}, & u \leq 1/2 \\ 1 - \sqrt{2(1-u)}, & u \geq 1/2 \end{cases}$$

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
int main(void) { gsl_rng * RNG; int i; double u, x;
  RNG = gsl_rng_alloc(gsl_rng_ranlux389);

  for (i = 0; i < 2000000; i++) { u = gsl_rng_uniform(RNG);
    if (u < 0.5) x = sqrt(2. * u) - 1.; else x = 1. - sqrt(2. * (1 - u));
    printf("% 22.16e\n", x); }

  gsl_rng_free(RNG); return 0; }
```

The 2<sup>nd</sup> idea is to notice that  $Y := \min(u_1, u_2)$  is distributed according to  $F_Y(y) = 2(1 - y)$  for  $0 \leq y \leq 1$ . We set  $x := \text{sign}(2u_3 - 1) \cdot \min(u_1, u_2)$ . This way we use three uniform numbers  $u_1, u_2, u_3$  to form one  $x$ , but the functions are simpler.

```

#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
int main(void) { gsl_rng * RNG; int i, j; double u[3], x;
  RNG = gsl_rng_alloc(gsl_rng_ranlux389);

  for (i = 0; i < 1000; i++) {
    for (j = 0; j < 3; j++) u[j] = gsl_rng_uniform(RNG);
    x = u[0]; if (u[1] < x) x = u[1]; x = copysign(x, 2. * u[2] - 1.);
    printf("% 22.16e\n", x); }

  gsl_rng_free(RNG); return 0; }

```

$$f_{X+Y}(z) = \int dx dy f_X(x) f_Y(y) \delta(x+y-z) = (f_X * f_Y)(z)$$

The 3<sup>rd</sup> idea is to notice that  $u_1 + u_2 - 1$  (like the sum of two dice rolls) would be distributed in the desired way. This can also be rewritten as  $u_1 - (1 - u_2)$ , and as  $u_2$  and  $1 - u_2$  have same distribution, let us generate  $x$  as  $x := u_1 - u_2$ :

```

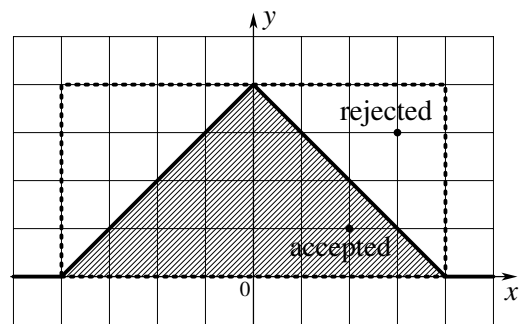
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
int main(void) { gsl_rng * RNG; int i, j; double u[2], x;
  RNG = gsl_rng_alloc(gsl_rng_ranlux389);

  for (i = 0; i < 2000000; i++) {
    for (j = 0; j < 2; j++) u[j] = gsl_rng_uniform(RNG);
    x = u[0] - u[1]; printf("% 22.16e\n", x); }

  gsl_rng_free(RNG); return 0; }

```

The 4<sup>th</sup> idea is the rejection method due to John von Neumann. Consider we have the desired density distribution function  $f_X(x)$  confined in a rectangle, with the share of the area under the  $f_X$  being not small. We can throw a point uniformly distributed in the rectangle by throwing two uniform numbers  $u_1$  and  $u_2$ . We then check whether the point in the rectangle is below the density  $f_X(x)$ , and if not, then we reject it and try again. In the case of success the horizontal coordinate is  $x$ .



```

#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
double f_X(double x) { if ((x <= -1.) || (x >= 1.)) return 0.; else
  { if (x <= 0.) return 1. + x; else return 1. - x; } }

int main(void) { gsl_rng * RNG; int i, j; double u[2], x;
  RNG = gsl_rng_alloc(gsl_rng_ranlux389);

```

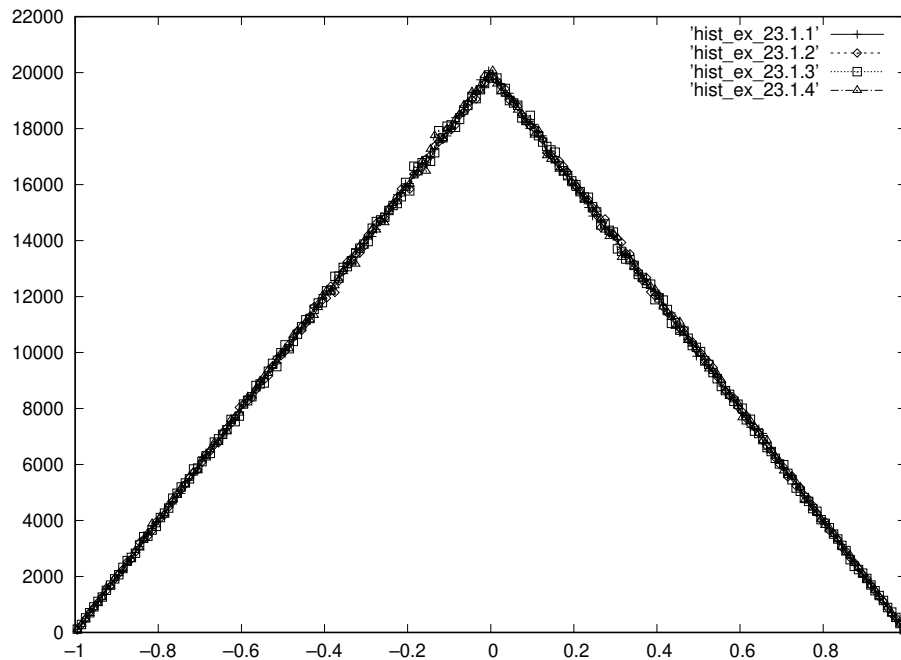
```

for (i = 0; i < 2000000; i++) {
    for (;;) { for (j = 0; j < 2; j++) u[j] = gsl_rng_uniform(RNG);
        x = 2. * u[0] - 1.; if (u[1] < f_X(x)) break; }
    printf("% 22.16e\n", x); }

gsl_rng_free(RNG); return 0; }

```

All the 4 methods give the same distribution of the random variable  $X$ :



**Example 23.2:** Let us construct a generator of random numbers with standard normal distribution.

The 1<sup>st</sup> idea is to use the inverse cumulative distribution function. Here  $F_X(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x dy e^{-y^2/2}$  is related to a so called error function. There is a built-in [inverse error function](#) in MATLAB (and GNU Octave), but not in C's `math.h` or even GNU GSL. Let us invert it using the bisection method (it is not going to be too fast):

```

#include <stdio.h>
#include <math.h>
#include <gsl/gsl_sf_erf.h>
#include <gsl/gsl_rng.h>
int main(void) { gsl_rng * RNG; int i; double u, x, xl, xr;
    RNG = gsl_rng_alloc(gsl_rng_ranlux389);

    for (i = 0; i < 1000000; i++) { u = gsl_rng_uniform(RNG);
        for (xl = -20., xr = 20.; xr - xl > 1.e-15;)
            { x = 0.5 * (xl + xr); if (gsl_sf_erf_Q(x) < u) xr = x; else xl = x; }
        printf("% 22.16e\n", x); }

    gsl_rng_free(RNG); return 0; }

```

The 2<sup>nd</sup> idea uses the Central Limit Theorem. Let us sum 100 (again, this is not too fast) uniformly distributed numbers, and then shift and rescale the result so that we get a number that is [almost] normally distributed, with zero mean and standard deviation being equal to 1:

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
int main(void) { gsl_rng * RNG; int i, j; double su, x;
  RNG = gsl_rng_alloc(gsl_rng_ranlux389);

  for (i = 0; i < 1000000; i++) {
    for (su = 0., j = 0; j < 100; j++) su += gsl_rng_uniform(RNG);
    x = (su - 50.) / sqrt(100. * (1. / 12.)); printf("% 22.16e\n", x); }

  gsl_rng_free(RNG); return 0; }
```

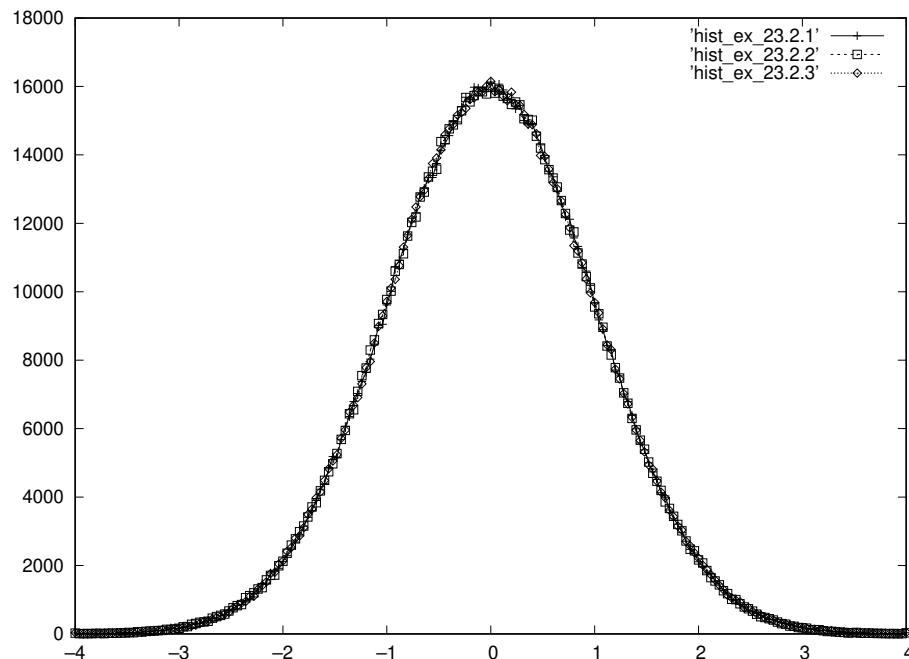
The 3<sup>rd</sup> idea is exploiting  $\frac{1}{2\pi}e^{-(x^2+y^2)/2}dxdy = \frac{1}{2\pi}e^{-r^2/2}rdrd\theta = e^{-s}ds\frac{d\theta}{2\pi}$ , where  $r = \sqrt{x^2+y^2}$  and  $s = r^2/2$ . We have exponential distribution for  $s$  and uniform one for  $\theta$ . The code is

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
int main(void) { gsl_rng * RNG; int i, j; double u[2], r, x;
  RNG = gsl_rng_alloc(gsl_rng_ranlux389);

  for (i = 0; i < 500000; i++) {
    for (j = 0; j < 2; j++) u[j] = gsl_rng_uniform(RNG);
    r = sqrt(-2. * log(u[0]));
    x = r * cos(2. * M_PI * u[1]); printf("% 22.16e\n", x);
    x = r * sin(2. * M_PI * u[1]); printf("% 22.16e\n", x); }

  gsl_rng_free(RNG); return 0; }
```

All the 3 methods give the same distribution of the random variable  $X$ :



## Problems and exercises

1. Write a random number generator that produces numbers distributed according to density distribution function

$$f(x) = \begin{cases} \exp(x)/(e-1), & 0 \leq x \leq 1; \\ 0, & x < 0 \text{ or } x > 1. \end{cases}$$

## 24 Monte Carlo method

Imagine you have a stochastic differential equation  $dx/dt = \xi(t)$ , where  $\xi(t)$  is a white noise with correlation function  $\langle \xi(t_1)\xi(t_2) \rangle = 2D\delta(t_1 - t_2)$ . Then the density distribution function  $\rho(t, x)$  of the random variable  $x(t)$  satisfies the so called Fokker–Planck equation  $\partial\rho(t, x)/\partial t = D\partial^2\rho(t, x)/\partial x^2$ . The quantity  $D$  is often called the diffusion coefficient.

One of the solutions of this partial differential equation is  $\rho(t, x) = \exp(-x^2/4Dt)/\sqrt{4\pi Dt}$ , with  $\lim_{t \rightarrow 0^+} \rho(t, x) = \delta(x)$ . The solution gives the density of  $x(t)$  conditioned by  $x(0) = 0$ . Notice that  $\langle x^2(t) \rangle = 2Dt$ , i.e.,  $x \sim \sqrt{Dt}$ .

Imagine we discretized time (with the time step being  $\tau$ ), and  $x(t)$ ,  $\xi(t)$  are substituted by their grid functions  $x_i$ ,  $\xi_{i+1/2}$ . We would like to write down some kind of update rule  $x_{i+1} = x_i + \tau \xi_{i+1/2}$ . What values the noise  $\xi_{i+1/2}$  does take? The correlation function  $\langle \xi(t_1)\xi(t_2) \rangle = 2D\delta(t_1 - t_2)$  is now  $\langle \xi_i \xi_j \rangle = 2D\delta_{ij}/\tau$ , i.e., all  $\xi_i$  are independent normally distributed random variables, with standard deviation  $\sqrt{2D/\tau}$ . Thus the update rule looks like  $x_{i+1} = x_i + \sqrt{2D\tau} \cdot \zeta$ , where  $\zeta$  on each step is independently chosen according to the standard normal distribution,  $f(\zeta) = \exp(-\zeta^2/2)/\sqrt{2\pi}$ . Note that  $x_{i+1} - x_i \sim \sqrt{\tau} \gg \tau$ . This can also be obtained from

$$\langle (x(t+\tau) - x(t))^2 \rangle = \int_t^{t+\tau} dt_1 \int_t^{t+\tau} dt_2 \langle \xi(t_1)\xi(t_2) \rangle = \int_t^{t+\tau} dt_1 \int_t^{t+\tau} dt_2 2D\delta(t_1 - t_2) = 2D\tau$$

The decay of information about the distant past in random process  $X(t)$  is often measured by autocorrelation or autocovariance function

$$K_{XX}(t_1, t_2) = E\left((X(t_1) - EX(t_1))(X(t_2) - EX(t_2))\right)$$

For stationary process  $X(t)$  the quantity  $EX(t)$  does not depend on  $t$ , while  $K_{XX}(t_1, t_2)$  depends just on time difference  $t_1 - t_2$ . If  $X(t_1)$  and  $X(t_2)$  are independent, then  $K_{XX}(t_1, t_2) = 0$ .<sup>9</sup> The smaller  $K_{XX}(\tau = t_1 - t_2)$ , the less dependence we expect between the values of  $X$  separated by  $\tau$  in time.

**Example 24.1:** Consider a stochastic differential equation  $dx(t)/dt = -x(t) + \xi(t)$ , where  $\xi(t)$  is white noise,  $\langle \xi(t_1)\xi(t_2) \rangle = \delta(t_1 - t_2)$ . The quantity  $x(t)$  is stirred by  $\xi$ , otherwise  $x(t)$  exponentially

<sup>9</sup> The reverse is not true. Consider, e.g.,  $X$  and  $Y$  to be uniformly distributed on the circle  $X^2 + Y^2 = 1$ . Then  $EX = EY = EXY = 0$  (so [linear correlation coefficient](#) between  $X$  and  $Y$  is zero), but  $X$  and  $Y$  are not independent, they are even functionally dependent.

decays with rate 1. We have  $x(t) = \int_{-\infty}^t dt' e^{-(t-t')} \xi(t')$ , and  $Ex(t) = 0$ , while

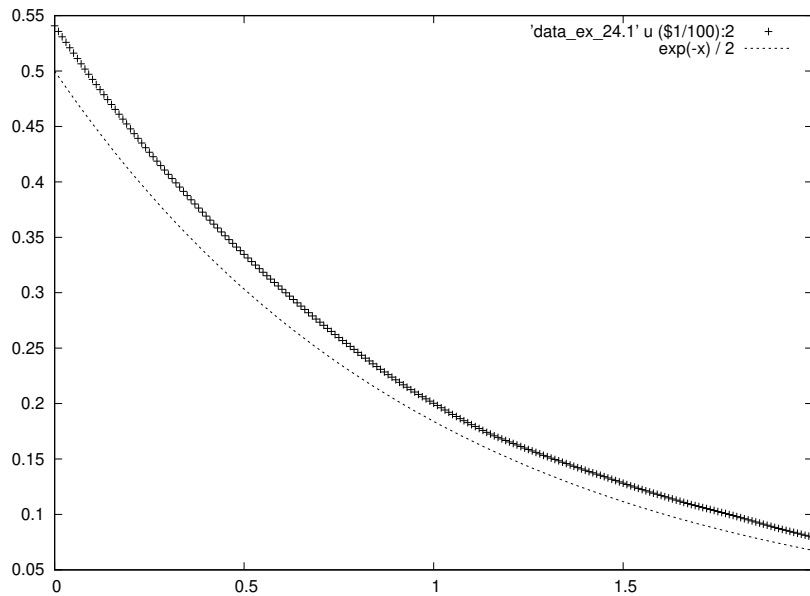
$$\begin{aligned} K_{xx}(t_1, t_2) &= Ex(t_1)x(t_2) = \int_{-\infty}^{t_1} dt'_1 \int_{-\infty}^{t_2} dt'_2 \exp(-(t_1 - t'_1) - (t_2 - t'_2)) \underbrace{E\xi(t'_1)\xi(t'_2)}_{\delta(t'_1 - t'_2)} \\ &= \int_{-\infty}^{\min(t_1, t_2)} dt' \exp(2t' - t_1 - t_2) = \frac{1}{2} \exp(-|t_1 - t_2|) \end{aligned}$$

The process  $x(t)$  is stationary, and  $K_{xx}(t_1, t_2)$  depends just on  $t_1 - t_2$ .<sup>10</sup>

The equation can be solved numerically using the update rule  $x_{i+1} = e^{-\tau}x_i + \sqrt{\tau}\zeta_{i+1/2}$ , here  $\tau$  is the time step. Here index in  $\zeta$  just distinguished the independent trials of standard normal distribution. The statistics of  $-\zeta$  is the same as of  $\zeta$ , and  $Ex_i = 0$ . As  $E\zeta = 0$ , we have  $Ex_{i+1}x_i = e^{-\tau}Ex_i^2$ , or  $K(\tau) = e^{-\tau}K(0)$ . We have  $K(2) = Ex_{i+2}x_i = E(e^{-\tau}x_{i+1} + \sqrt{\tau}\zeta_{i+3/2})x_i = e^{-\tau}K(1) = e^{-2\tau}K(0)$ . It can be shown that  $K(t) = e^{-t}K(0)$ , *i.e.*, correlations between different time values of  $x$  decay with rate 1.

```
import numpy as np; from random import normalvariate
x, dt, corr, N = np.zeros(100200), 0.01, np.zeros(201), 0
x[0], sigma = 0., np.sqrt(dt)
for i in range(1, 100200):
    x[i] = (1. - dt) * x[i - 1] + normalvariate(0., sigma)
    if (i >= 200):
        for j in range(0, 201):
            corr[j] += x[i] * x[i - j]
        N += 1

print('# N =', N)
for j in range(0, 201):
    corr[j] /= N
    print(j, corr[j])
```



<sup>10</sup> The reverse is not true. It could be that  $K_{XX}(t_1, t_2) = K_{XX}(t_1 - t_2)$ , but the process  $X(t)$  is non-stationary.

**Example 24.2:** Consider a stochastic differential equation  $dx(t)/dt = x(t)\xi(t)$ , where  $\xi(t)$  is white noise,  $\langle \xi(t_1)\xi(t_2) \rangle = 2D\delta(t_1 - t_2)$ . We would like to solve them numerically, producing realizations of the process  $x(t)$ . One can come up with, *e.g.*, the following update rules:

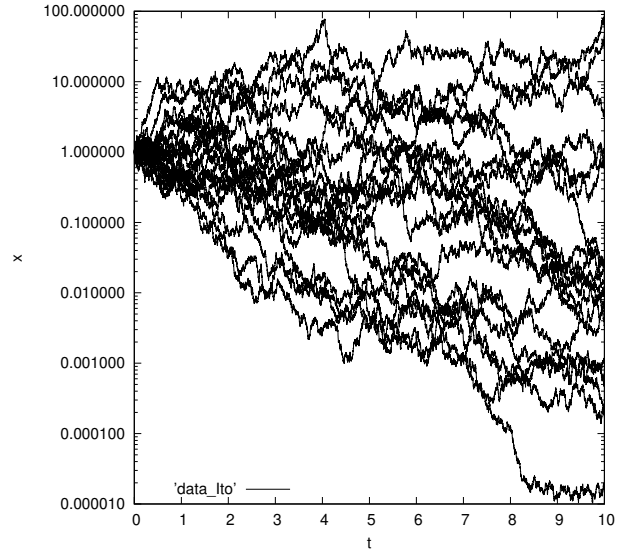
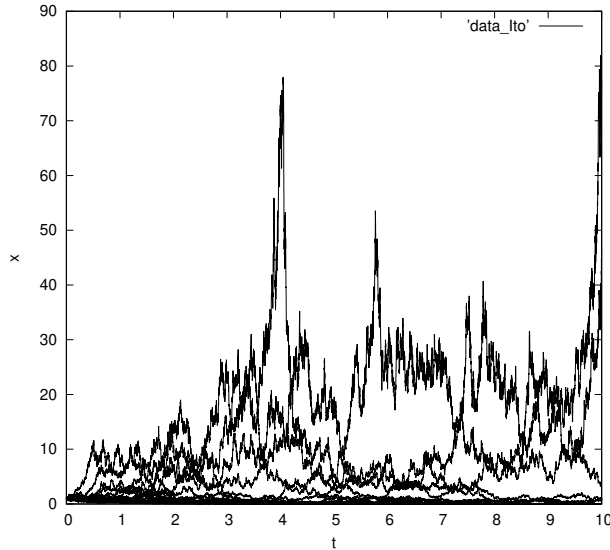
$$\text{It\^o: } x_{i+1} = x_i + \sqrt{2D\tau}x_i\zeta_{i+1/2}$$

$$\text{Stratonovich: } x_{i+1} = x_i + \sqrt{2D\tau}\frac{x_i + x_{i+1}}{2}\zeta_{i+1/2}$$

Both seem sane discretizations, but they will produce different dynamics. It is easy to see that in It\^o discretization we have  $Ex_{i+1} = Ex_i = x(0)$ . To the contrary, in Stratonovich discretization we have  $x_{i+1} = x_i(1 + \sqrt{2D\tau}\zeta/2)/(1 - \sqrt{2D\tau}\zeta/2) = x_i(1 + \sqrt{2D\tau}\zeta/2)(1 + \sqrt{2D\tau}\zeta/2 + 2D\tau\zeta^2/4 + \dots) = x_i(1 + \sqrt{2D\tau}\zeta + 2D\tau\zeta^2/4 + 2D\tau\zeta^2/4 + \dots) \rightarrow x_i(1 + D\tau + \sqrt{2D\tau}\zeta)$ . We have  $Ex_{i+1} = (1 + D\tau)Ex_i$ , and  $Ex(t) = e^{Dt}x(0)$ .

Here is a Python script solving  $dx/dt = x\xi$  using It\^o discretization ( $D = 1/2$ ):

```
from math import sqrt; from random import normalvariate
dt = 0.001
for n in range(0, 20):
    t, x = 0., 1.
    print(t, x)
    for i in range(0, 10000):
        t, x = t + dt, x * (1. + sqrt(dt) * normalvariate(0., 1.))
        print(t, x)
    print()
```



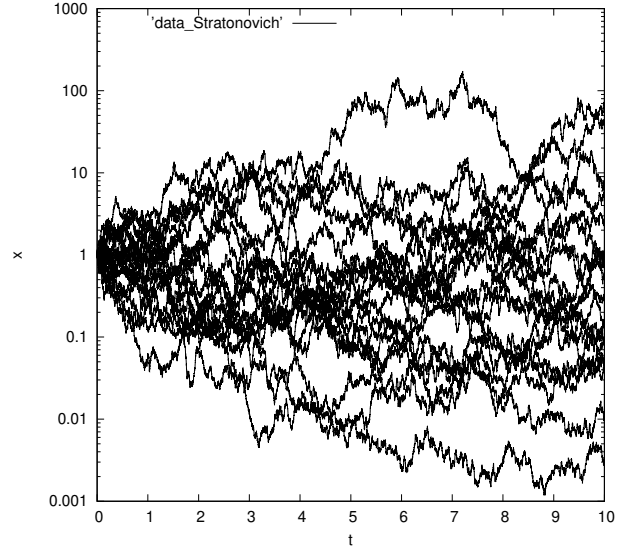
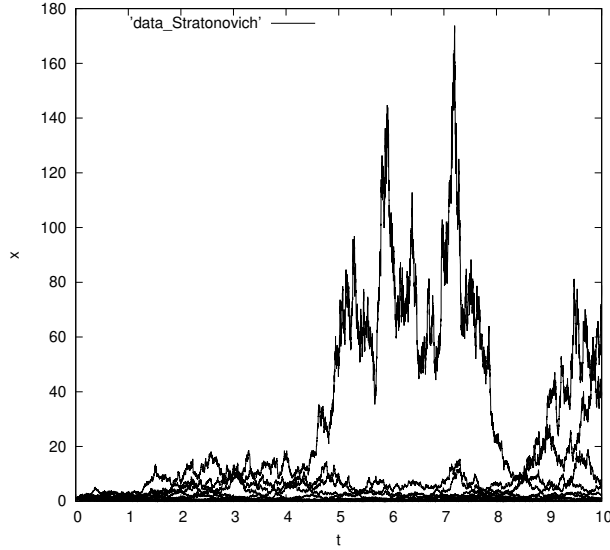
Here is a Python script solving  $dx/dt = x\xi$  using Stratonovich discretization ( $D = 1/2$ ):

```
from math import sqrt; from random import normalvariate
dt = 0.001
for n in range(0, 20):
    t, x = 0., 1.
    print(t, x)
    for i in range(0, 10000):
```

```

xi = sqrt(dt) * normalvariate(0., 1.) / 2.
t, x = t + dt, x * (1. + xi) / (1 - xi)
print(t, x)
print()

```



**Example 24.3:** Consider we start the diffusion process ( $D = 1/2$ ) with  $x(0) = 1$ . How the time  $T$  of hitting  $x = 0$  for the first time (*i.e.*,  $x(T) = 0$ ,  $x(t) > 0$  for all  $t < T$ , or  $T := \min_t x(t) \leq 0$ ) is distributed? This problem can be modeled by the diffusion equation  $\partial \rho(t, x) / \partial t = \frac{1}{2} \partial^2 \rho(t, x) / \partial x^2$  with [absorbing] boundary condition  $\rho(t, 0) = 0$  and initial condition  $\rho(0, x) = \delta(x - 1)$ . The solution looks like

$$\rho(t, x) = \frac{1}{\sqrt{2\pi t}} \left[ \exp\left(-\frac{(x-1)^2}{2t}\right) - \exp\left(-\frac{(x+1)^2}{2t}\right) \right]$$

We have  $F_T(t) = 1 - \int_0^\infty dx \rho(t, x)$ , and

$$f_T(t) = \frac{dF_T(t)}{dt} = - \int_0^1 dx \frac{\partial \rho(t, x)}{\partial t} = - \frac{1}{2} \int_0^1 dx \frac{\partial^2 \rho(t, x)}{\partial x^2} = \underbrace{\frac{1}{2} \frac{\partial \rho(t, x)}{\partial x} \Big|_{x=0}}_{\text{diffusive flux to the wall}} = \frac{\exp(-1/2t)}{\sqrt{2\pi t^3}}$$

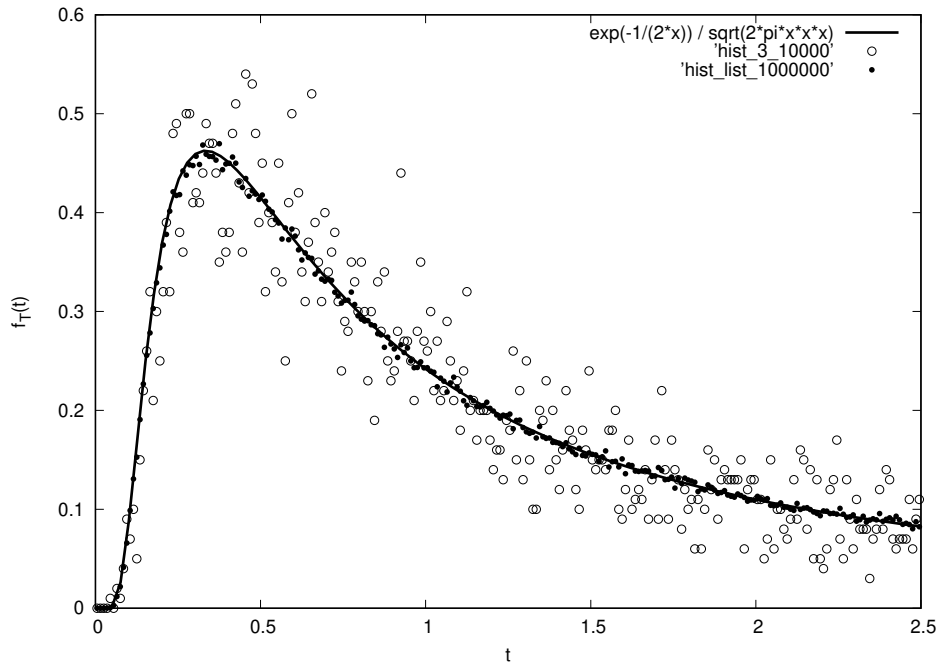
Here is a “naive” Python script which generates 10000 instances of the random walk, terminating each time the particle hits  $x = 0$  or when time  $t$  exceeds 3, whichever happens first:

```

from math import sqrt; from random import normalvariate
for i in range(0, 10000):
    x, t, dt = 1., 0., 0.0001
    while ((x > 0.) and (t < 3.)):
        x, t = x + sqrt(dt) * normalvariate(0., 1.), t + dt
    print(t)

```

If we would terminate only when the particle hits  $x = 0$ , then sometimes generating an instance of a random walk would take very long time. It is not surprising, as the expected value of the hitting time  $T$  is either  $+\infty$  or [formally] does not exists. The histogram from  $10^4$  [and also  $10^6$ ] trials looks like

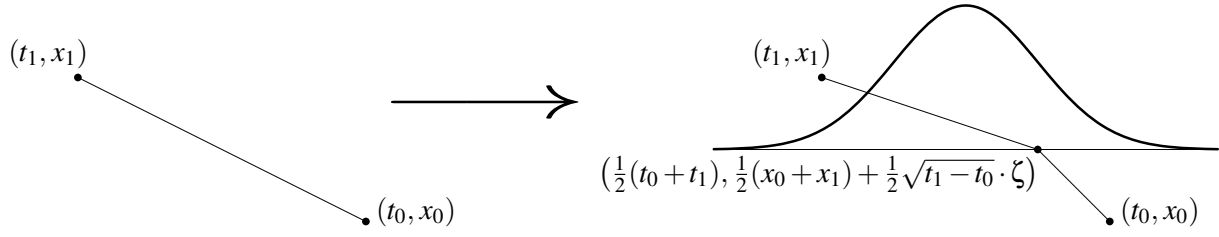


In order to work with many trials, the trials should be done in a more efficient way. In the “naive” script we spend too much time if we go far from the wall. There, to speed up the computation, we can safely do larger in time steps:

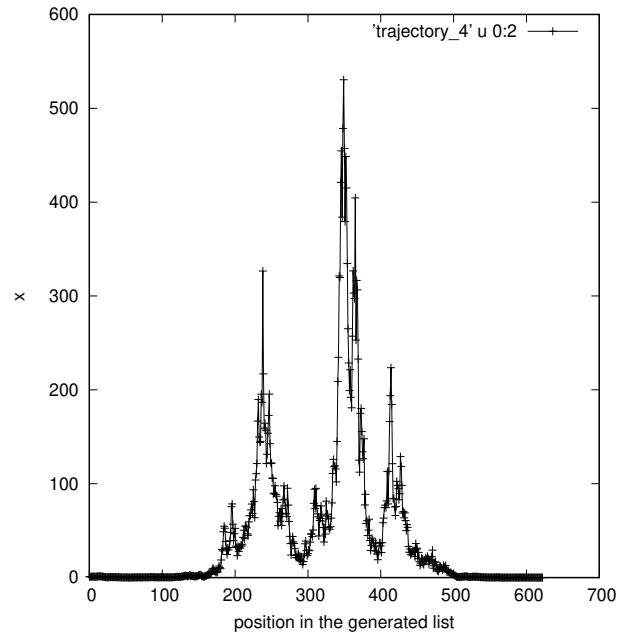
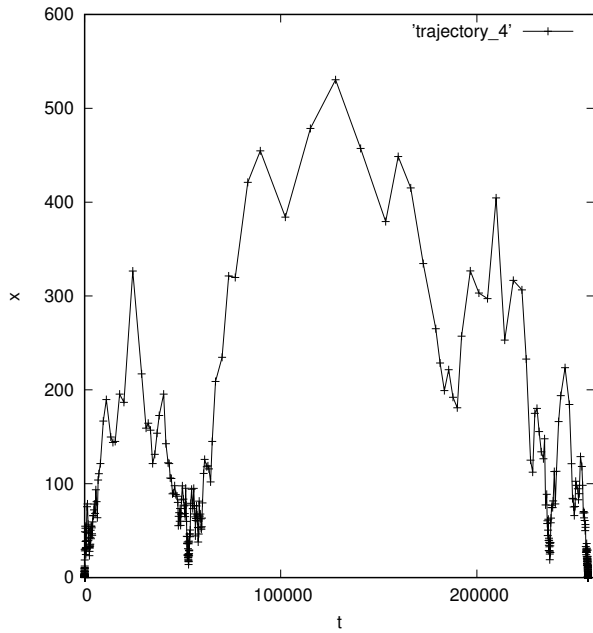
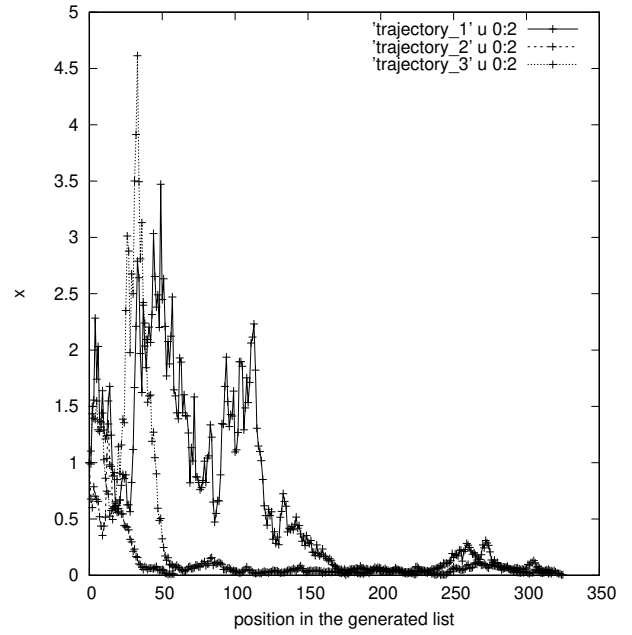
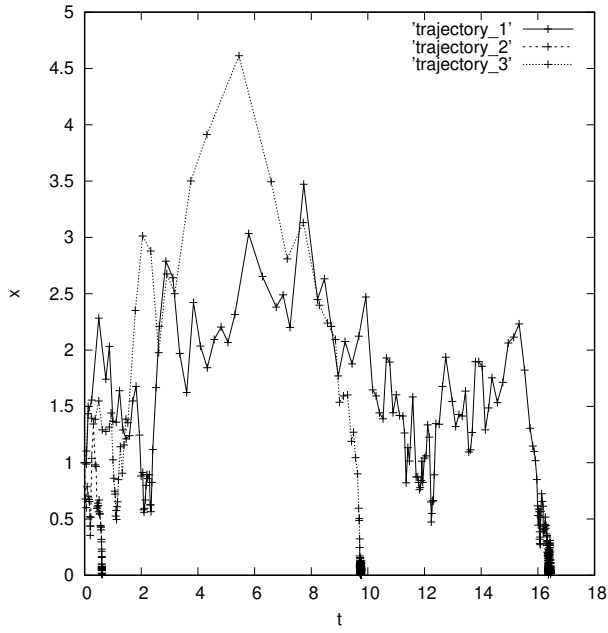
```
from math import sqrt; from random import normalvariate
t, x = [0.], [1.]
while (x[0] > 0.):
    dt = x[-1]**2; t.append(t[-1] + dt)
    x.append(x[-1] + normalvariate(0., sqrt(dt)))
    while (len(t) > 1):
        if ((t[1] - t[0] > 0.0001) and ((x[0] + x[1])**2 < 50. * (t[1] - t[0]))):
            dt = t[1] - t[0]; t.insert(1, 0.5 * (t[0] + t[1]))
            x.insert(1, 0.5 * (x[0] + x[1]) + normalvariate(0., sqrt(dt) / 2.))
            if (x[1] < 0.):
                del t[2:], x[2:]
        else:
            print(t[0], x[0])
            del t[0], x[0]
```

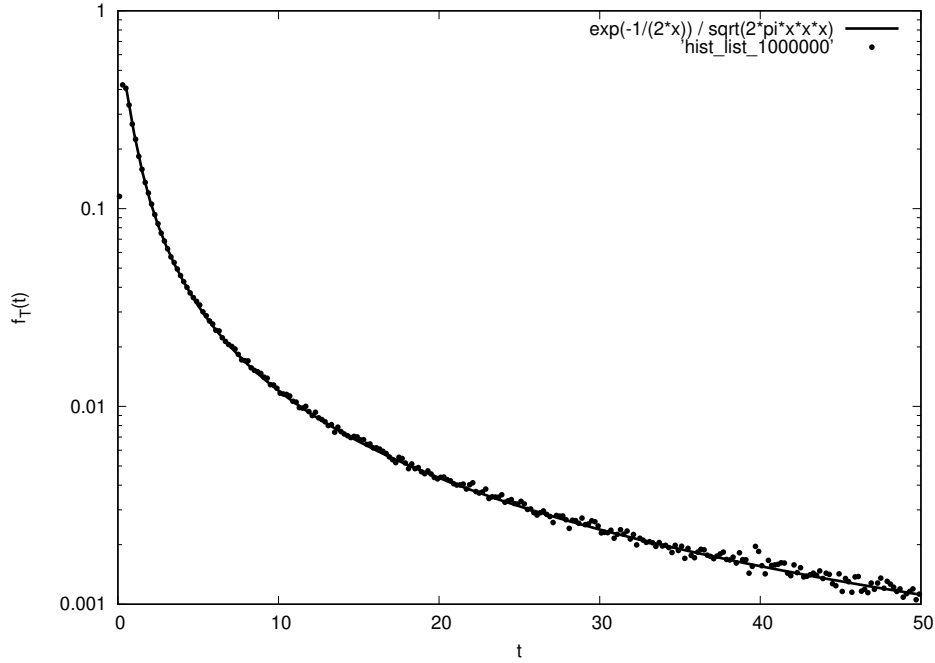
Here trajectory is built as needed. If we didn’t hit  $x = 0$  yet, we add a time step that is large enough to realistically expect a hitting event within the step. (Here it happens with the probability  $P(z > 1) \approx 0.16$ .) Next we review an already formed trajectory, trying to fill in the gaps and check whether hitting  $x = 0$  did happen there. If between the two points of the built trajectory the probability to hit  $x = 0$  is smaller than about  $10^{-10}$ , we don’t refine the trajectory there any further. (Same if the time difference between two points is less than 0.0001.)

If  $(t_1 - t_0 > 0.0001)$  and  $(x_0, x_1)$  are not too far from 0) then refine the trajectory:



$$f_{x(\frac{1}{2}(t_0+t_1))}(x) \propto \exp\left(-\frac{(x-x(t_0))^2}{2 \cdot \frac{1}{2}(t_1-t_0)}\right) \cdot \exp\left(-\frac{(x-x(t_1))^2}{2 \cdot \frac{1}{2}(t_1-t_0)}\right)$$





## 24.1 Importance sampling

Consider we have a random variable  $X$  which is distributed with density distribution function  $f_X(x)$ , and we want to find  $EA(X)$ , *i.e.*, we want to find what is the average of the quantity  $A$  which depends on  $X$  in some way. The simplest application of the Monte Carlo method would be to sample the distribution  $f_X$  many ( $N$ ) times, get values  $x_1, x_2, \dots, x_N$ , and then estimate  $EA(X) \approx \frac{1}{N} \sum_{i=1}^N A(x_i)$ .

Sometimes (that of course depends on the statistics of  $X$  and the nature of the function  $A(\cdot)$ ) the values of  $X$  that contribute to  $EA(X)$  are quite rare (and we can think of the value of  $EA(X)$  as small). In this case in order to accurately estimate  $EA(X)$  the number of samples  $N$  needs to be very large, so that all the relevant values of  $X$  were sampled enough many times.

Importance sampling is the technique of speeding up the accurate estimation of  $EA(X)$ . The idea is to sample not  $f_X(x)$ , but some other distribution  $g(x)$ , and write

$$EA(x) = \int dx A(x) f_X(x) = \int dx \left[ A(x) \frac{f_X(x)}{g(x)} \right] g(x)$$

One can interpret the expression on the right as calculation of the expected value of the quantity  $A(x)f_X(x)/g(x)$ , where  $x$  is distributed with density  $g(x)$ . If the values of  $x$  relevant for the expected value  $EA(x)$  are well covered by the distribution  $g(x)$ , then in our  $N$  samples we would comprehensively contain all the  $x$  needed. Each of that values of  $x$  would be weighted by an additional [small] factor  $f_X(x)/g(x)$ .

**Example 24.1.1:** Consider the exponential diststribution  $f(x) = e^{-x}$ ,  $x \geq 0$ . Let us estimate  $P(x > 10) = e^{-10} \approx 4.54 \cdot 10^{-5}$ . We have  $P(x > 10) = EA(x)$ , where  $A(x) = 1$  if  $x > 10$  and  $A(x) = 0$  if  $x \leq 10$ . The function  $A(x)$  cuts out the far tail of exponential distribution.

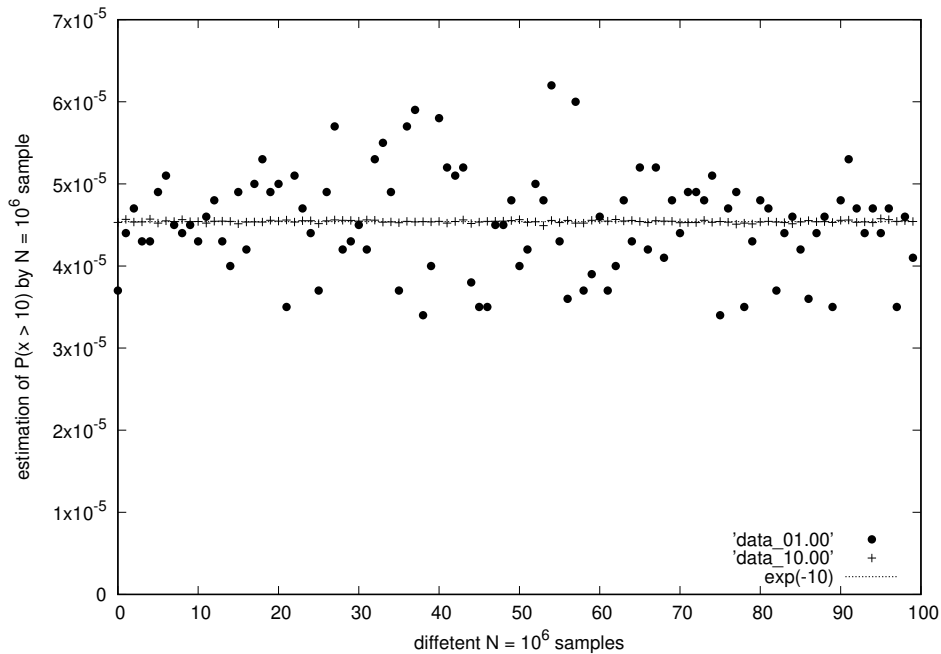
The simplest application of the Monte Carlo method with  $N = 10^6$  samples would produce only about 45 events with  $x > 10$  on average, with standard deviation being  $\approx \sqrt{45} \approx 7$  events. Thus the estimation of  $P(x > 10)$  would be something like  $(4.54 \pm 0.7) \cdot 10^{-5}$ .

Here is a Python script which estimates  $P(x > 10)$ , with the sample size  $N$  being one of the two inputs:

```
from sys import argv; from math import log, exp; from random import random
X, N, width, catch = 10., int(argv[2]), float(argv[1]), 0.
for i in range(0, N):
    x = -width * log(random())
    if (x > X):
        catch += width * exp((1. / width - 1.) * x)
print(catch / N)
```

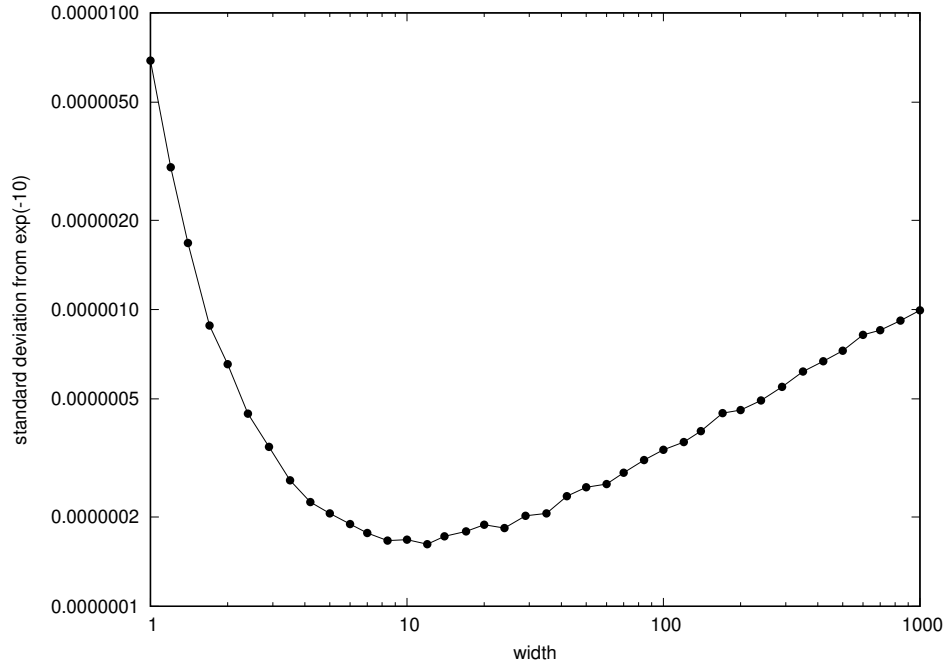
Here  $g(x) := \exp(-x/\text{width}) / \text{width}$ ,  $x \geq 0$ , *i.e.*, it is an exponential distribution of width `width`. When `width` is large, we often get not small, *i.e.*,  $> 10$  values of  $x$ , thus getting more events contributing into our estimation of  $P(x > 10)$ . Each realization of  $x > 10$  contributes not by  $1/N$ , but by a smaller amount to  $P(x > 10)$ , to compensate for the fact that we throw  $x > 10$  more frequently.

Here is a batch of 100 attempts to estimate  $P(x > 10)$  by  $N = 10^6$  sample size, with `width = 1` and `width = 10`:



The exact answer  $\exp(-10)$  is shown by a horizontal line, which is here all covered by points corresponding to `width = 10`. The accuracy of determining  $P(x > 10)$  from the same  $N = 10^6$  trials, but the ones adapted to the quantity of interest  $P(x > 10)$ , is improved a lot.

Here is how the standard deviation  $\sqrt{E((\text{estimation of } P(x > 10) \text{ from } N = 10^6 \text{ trials}) - \exp(-10))^2}$  depends on `width`:



It has the minimal value at around `width` = 10. There the share of samples that result in  $x > 10$  is of the order of 1 (namely with `width` = 10 the share is  $1/e \approx 0.37$ ).

Another variant (and one can think of many more) of deforming  $f_X(x) \rightarrow g(x)$  is shifting the distribution:  $g(x) = \exp(\text{shift} - x)$ ,  $x \geq \text{shift}$ . Increasing `shift` we produce  $x > 10$  events more often, and our estimation of  $P(x > 10)$  is going to be more accurate. At `shift` = 10 the computation of  $P(x > 10)$  is going to be *exact*, as the chosen  $g(x)$  exactly coincides with conditional density  $f_X(x|x > 10)$ . With `shift` = 10 all the samples will fall into  $x > 10$  region and in the importance sampling setting will have the same weight:  $A(x)f_X(x)/g(x) = 1 \cdot \exp(-x)/\exp(\text{shift} - x) = e^{-10}$ .<sup>11</sup> When `shift` > 10 our estimation of  $P(x > 10)$  will be underestimating no matter how large is our sample size — the distribution  $g(x)$  does *not* cover some of the values of  $x$  that *do* contribute to  $P(x > 10)$ , namely  $10 < x < \text{shift}$  ones.

**Example 24.1.2:** Consider a stochastic differential equation  $dx/dt = -x + \epsilon x^2 + \xi$ , where  $\xi(t)$  is white noise with correlation function  $\langle \xi(t_1)\xi(t_2) \rangle = \delta(t_1 - t_2)$ . The parameter  $\epsilon$  is small. The white noise  $\xi$  causes  $x(t)$  to diffuse. When  $x$  is not large,  $x < 1/\epsilon$ , there is a tendency to move towards the origin  $x = 0$ . When  $x > 1/\epsilon$ , then non-random part of  $dx/dt$  is positive, and with high chances  $x(t)$  will move to the right without return (because of  $\epsilon x^2$  term in  $dx/dt$  the trajectory  $x(t)$  will reach  $x = +\infty$  in finite time). During any time interval there is a non-zero probability that the particle in its diffusing process overcomes the tendency to move towards the origin and escapes to large  $x > 1/\epsilon$ . What is the probability per unit time to escape?

This problem is a typical one for which the instanton method works. In order to escape the noise should push the particle to the right more eagerly than usual. The shape of the noise  $\xi(t)$  should be

<sup>11</sup> For more complicated situations it may be not that easy to choose  $g(x)$  in such a way that importance sampling produces the exact answer with no fluctuations. We would like to have  $g(x)$  being normalized  $A(x)f_X(x)$  (which is possible only if  $A(x) \geq 0$  for all  $x$ ), and we'll have to compute the normalization factor which is  $EA(x)$  itself.

somewhat optimized for making the particle to escape, any substantial deviation from the optimal shape would necessitate the increase in noise amplitude, thus greatly reducing the probability of such noise  $\xi(t)$  to appear.

(a) One can write down the noise optimality equations, and then solve them. We want to maximize the density distribution function of  $\xi$ , but the noise  $\xi(t)$  should be such that  $x(t)$  reaches  $1/\epsilon$ . This can be written as the following variational problem: We need to find extrema of

$$\mathcal{L}\{x(t), \xi(t), p(t)\} = \frac{1}{2} \int dt \xi^2(t) + \int dt p(t) \left( \frac{dx(t)}{dt} + x(t) - \epsilon x^2(t) - \xi(t) \right)$$

The noise density  $f\{\xi(t)\} \propto \exp(-\frac{1}{2} \int dt \xi^2(t))$ , so maximizing it means minimizing the integral inside the exponent. Here  $p(t)$  is a Lagrange multiplier for the equality constraint meaning that the equation for  $x$  is satisfied at the moment  $t$ . The boundary conditions are  $x(-\infty) = 0$  (we start at typical  $x$  not far from the origin) and  $x(+\infty) = 1/\epsilon$  (we reach the point after which  $\xi$  being switched off the trajectory moves to the right).

Making variation with respect to  $p$ , we just reproduce the equation of motion for  $x$ . Making variation w.r.t.  $\xi$ , we get  $\xi(t) = p(t)$ . Variation w.r.t.  $x$  produces the equation for  $p(t)$ :  $dp/dt = p(1 - 2\epsilon x)$ . The equations for  $x$  and  $p$  produce a Hamiltonian system:  $\mathcal{H}(x, p) := \frac{1}{2}p^2 - p(x - \epsilon x^2)$ , and  $\frac{dx}{dt} = \frac{\partial \mathcal{H}}{\partial p}$ ,  $\frac{dp}{dt} = -\frac{\partial \mathcal{H}}{\partial x}$ . On the starting and ending points,  $(x, p) = (0, 0)$  and  $(1/\epsilon, 0)$ , we have  $\mathcal{H} = 0$ , while  $\mathcal{H}$  as the Hamiltonian does not depend on time. Thus  $\mathcal{H} \equiv 0$ , and  $p = 2(x - \epsilon x^2)$ . This gives  $\frac{dx}{dt} = x - \epsilon x^2$  with the solution  $x(t) = \frac{1}{2\epsilon} (1 + \tanh(t/2))$  (and actually also its shifts in time). We get  $\xi(t) = p(t) = 2(x(t) - \epsilon x^2(t)) = 1/2\epsilon \cosh^2(t/2)$ . The probability per unit time to escape (with some accuracy) can be estimated as

$$P \sim f\{\xi(t)\} \sim \exp\left(-\frac{1}{2} \int dt \xi^2(t)\right) \approx \exp\left(-\frac{1}{8\epsilon^2} \int \frac{dt}{\cosh^4(t/2)}\right) = \exp\left(-\frac{1}{3\epsilon^2}\right)$$

(b) The density distribution function  $\rho(t, x)$  satisfies Fokker–Planck equation

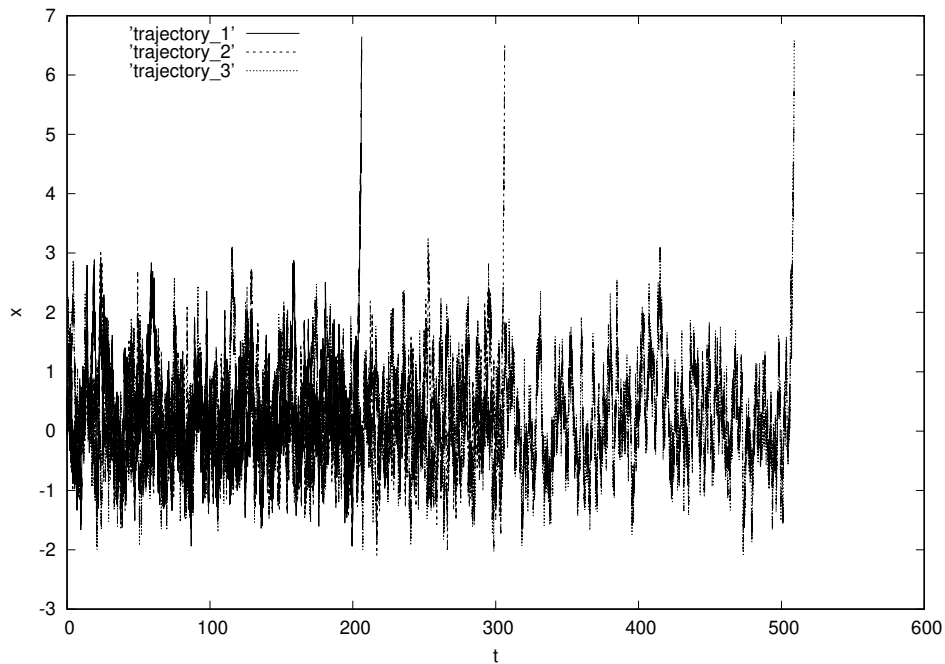
$$\frac{\partial \rho(t, x)}{\partial t} = \frac{\partial}{\partial x} \left( \frac{1}{2} \frac{\partial}{\partial x} + x - \epsilon x^2 \right) \rho(t, x)$$

If  $\epsilon = 0$  (i.e.,  $dx/dt = -x + \xi$ ), then  $\rho(t, x) = \exp(-x^2)/\sqrt{\pi}$  is a stationary solution. For any  $\epsilon > 0$  there is no normalizable stationary solution, but one can write down a solution with constant flux:  $(\frac{1}{2} \frac{\partial}{\partial x} + x - \epsilon x^2) \rho(x) = -J$ , and  $\rho(x) = 2J \int_x^\infty dx' \exp(-x'^2 + 2\epsilon x'^3/3 + x'^2 - 2\epsilon x'^3/3)$ . (The integral over  $x'$  converges because of  $-2\epsilon x'^3/3$  term inside the exponent.) We have  $\rho(0) = 2J \int_0^\infty dx' \exp(x'^2 - 2\epsilon x'^3/3) \approx 2J\sqrt{\pi} \exp(1/3\epsilon^2)$ . (The last integration was calculated by the saddle-point method, the integrand is cumulated near  $x' \approx 1/\epsilon$ .) The flux  $J$  is small, so  $\rho(x)$  tries to be similar to  $\exp(-x^2/2)/\sqrt{\pi}$  near the origin, with  $\rho(0) = 1/\sqrt{\pi}$ , which gives  $J \approx \exp(-1/3\epsilon^2)/2\pi$ .

(c) We can directly simulate the SDE  $dx/dt = x - \epsilon x^2 + \xi$ :

```
from sys import argv; import numpy as np; from random import normalvariate
t, x, dt, eps = 0., 0., 0.01, float(argv[1])
sigma = np.sqrt(dt)
while (x < 2. / eps):
    print(t, x)
    t, x = t + dt, x + dt * (-x + eps * x**2) + normalvariate(0., sigma)
```

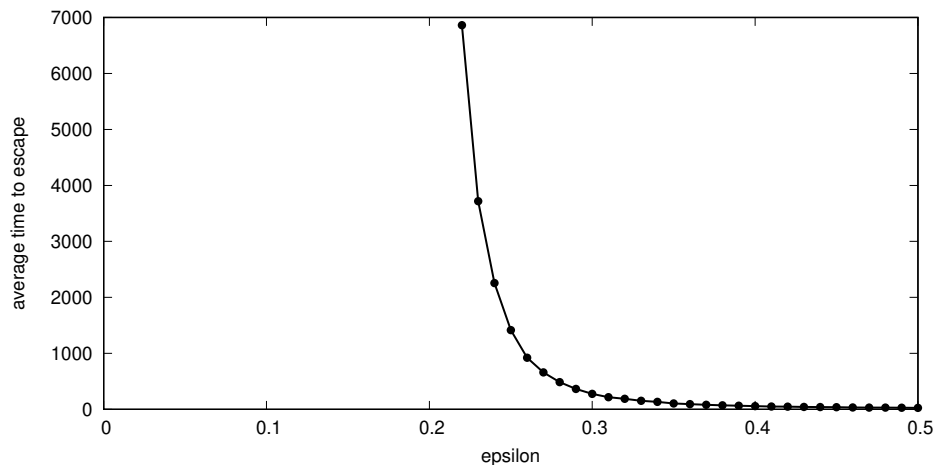
and get how  $x(t)$  could depends  $t$ . Here are 3 realizations for  $\varepsilon = 0.3$ :

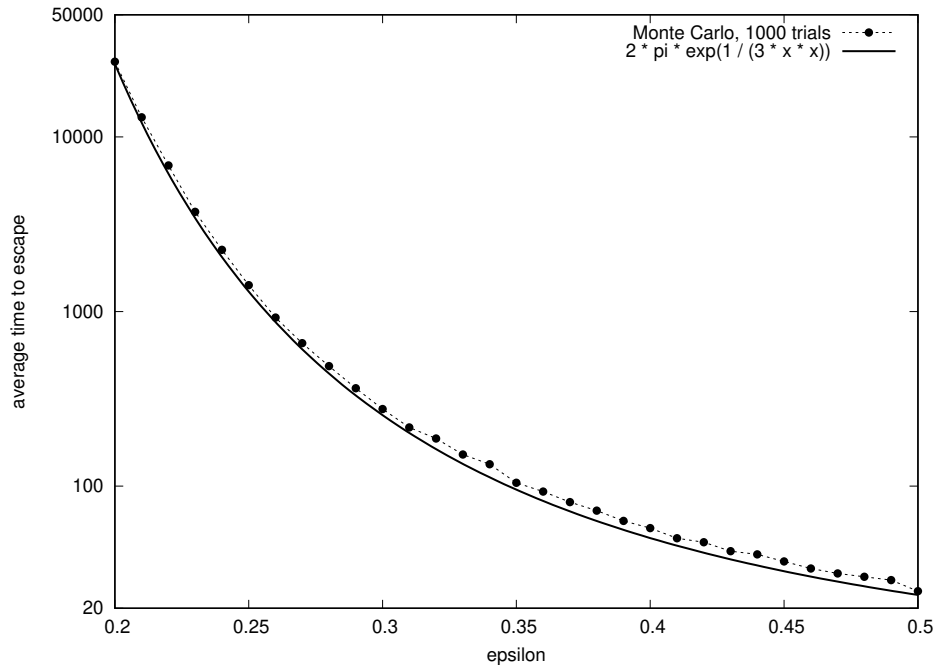


Here is a Python script that computes average time to escape by naive Monte Carlo, from 1000 trials (for loop over m), parameter  $\varepsilon$  is supplied from the command line:

```
from sys import argv; import numpy as np; from random import normalvariate
dt, eps, AVG = 0.01, float(argv[1]), 0.
sigma = np.sqrt(dt)
for m in range(0, 1000):
    t, x = 0., 0.;
    while (x < 2. / eps):
        t, x = t + dt, x + dt * (-x + eps * x**2) + normalvariate(0., sigma)
    print(t)
    AVG += t
print('#', AVG / 1000.)
```

Here is the graph of average time to escape as a function of  $\varepsilon$ :



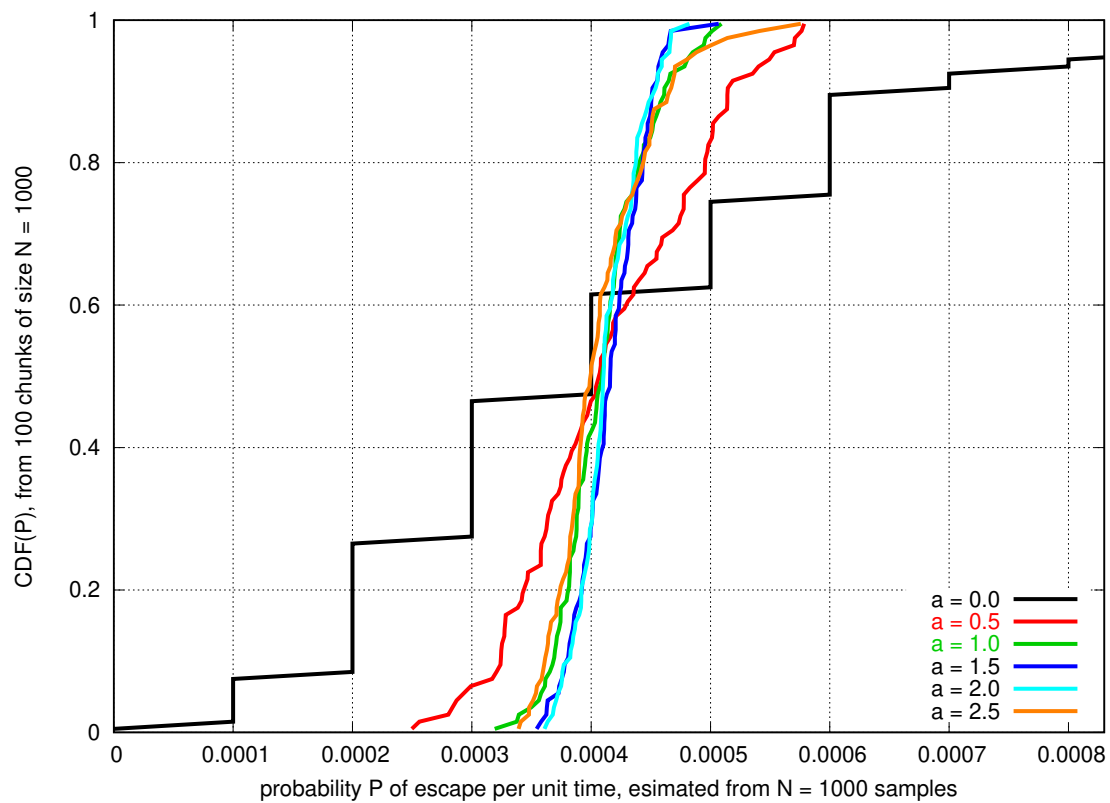
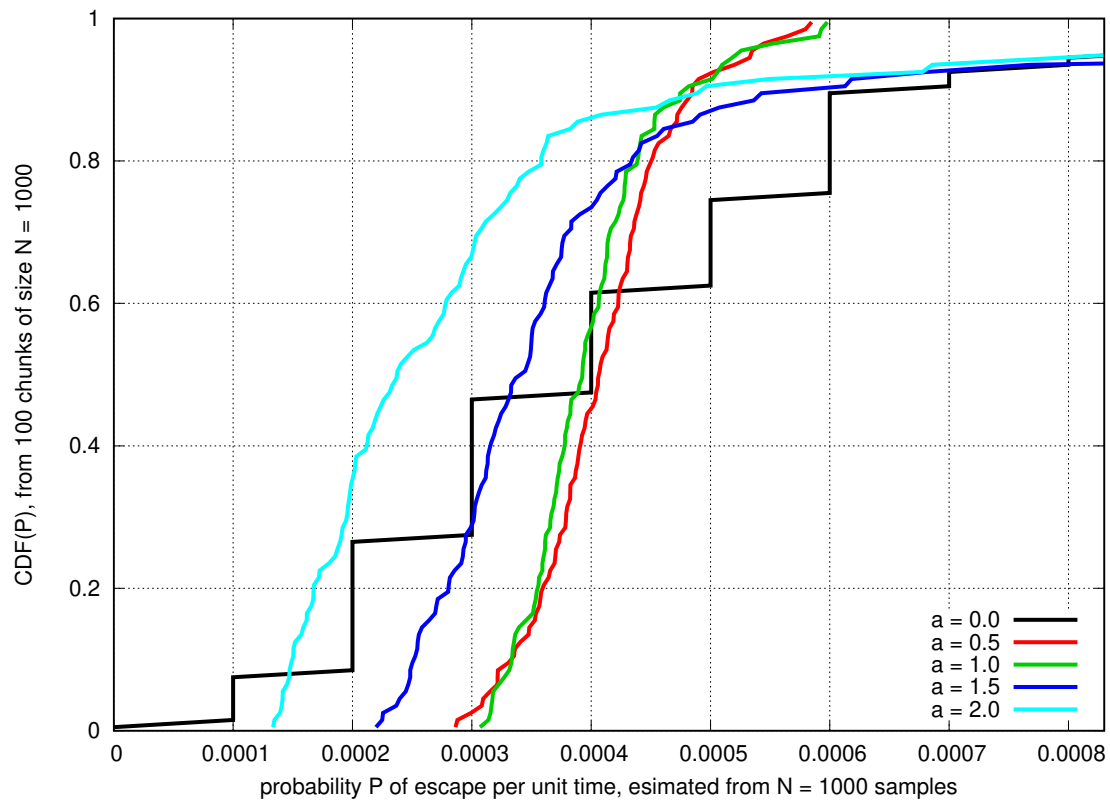


Let us apply the importance sampling technique in 2 [similar] ways:




```
[...]/teaching/2020-1/math_575b/notes/Monte_Carlo/importance_sampling/ex_24.1.2$
cat ex_24.1.2_IS1.py
from sys import argv; import numpy as np; from random import normalvariate
dt, eps, a, T = 0.01, float(argv[1]), float(argv[2]), 10.
sigma = np.sqrt(dt)
for m in range(0, 100000):
    t, x, compensation = 0., 0., 1.
    while ((t < T) and (x < 2. / eps)):
        v, dt_xi = -x + eps * x**2, normalvariate(0., sigma)
        V = v + a
        t, x = t + dt, x + dt * V + dt_xi
        compensation *= np.exp((v - V) * (dt_xi - 0.5 * dt * (v - V)))
    if (x > 1.5 / eps):
        print(compensation / T)
    else:
        print(0.)
[...]/teaching/2020-1/math_575b/notes/Monte_Carlo/importance_sampling/ex_24.1.2$
diff --suppress-common-lines -tyW 156 ex_24.1.2_IS1.py ex_24.1.2_IS2.py
V = v + a
V = (1. - a) * v if (v < 0.) else v
[...]/teaching/2020-1/math_575b/notes/Monte_Carlo/importance_sampling/ex_24.1.2$
```

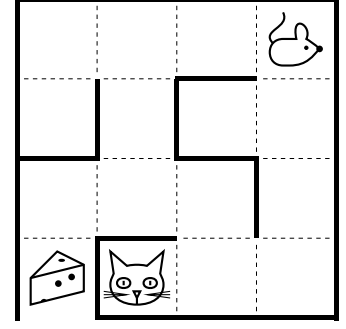
Here we substitute the update rule  $x_{i+1} = x_i + v\tau + \sqrt{\tau}\zeta$   $\rightarrow$   $x_{i+1} = x_i + V\tau + \sqrt{\tau}\zeta$ , and the compensation factor is  $\text{compensation} = \exp((x_{i+1} - x_i - v\tau)^2 / 2\tau) / \exp((x_{i+1} - x_i - V\tau)^2 / 2\tau)$ .

The two graphs below correspond to these 2 ways. The second way tries to mimic the instanton from (a), and would correspond to  $a = 2$ , it is then the velocity  $dx/dt$  is changed from  $-x + \epsilon x^2$   $\rightarrow$   $-x + \epsilon x^2 + p = x - \epsilon x^2$ .



## Problems and exercises

1. A mouse, starting from , runs through the maze on the right. At each step it moves to a neighboring cell that is not separated by a wall (chosen with equal probability, independently of the past). The mouse continues moving in this way until it eats the cheese at  (after that it escapes to the outside), or it is eaten by the cat at . Find (by means of your choice, *e.g.*, analytically, or studying eigenvectors of the corresponding Markov chain transition matrix, or by Monte Carlo method, *etc.*) the probability that the mouse escapes.



2. Consider a diffusing particle (with diffusion coefficient  $D = \frac{1}{2}$ ) inside the triangle  $x \geq 0$ ,  $y \geq 0$ , and  $x + y \leq 1$ . The particle starts at  $x(0) = y(0) = \frac{1}{3}$ . Find the distribution of the hitting the walls of the triangle time.

3. Consider a random variable  $X \sim N(0, 1)$ . Use importance sampling to estimate  $EX^{20}$ .

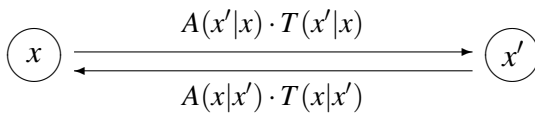
4. Consider a Markov chain with transition probabilities  $T(x, x+1) = 1/3$ ,  $0 \leq x < N$ ;  $T(0, 0) = T(x, x-1) = 2/3$ ,  $0 < x < N$ ; and  $T(N, N) = 1$ . The chain starts from  $X_0 = 0$ . As  $T(x, x+1) = \frac{1}{3} < \frac{2}{3} = T(x, x-1)$ , there is a substantial bias to the left, and typically  $X_n$  is kept not too far from 0. There is [not too large] probability  $\gamma$  per step/unit time to escape/reach the absorbing state  $N$ . Find it for  $N = 20$  by (a) direct simulation of the Markov chain, also (b) speed up the computation of  $\gamma$  by some kind of importance sampling.

## 25 Monte Carlo Markov chains (MCMC)

A way to sample an arbitrary distribution using Markov chains was proposed in 1953 by Metropolis *et al.*<sup>12</sup> and generalized in 1970 by Hastings.<sup>13</sup> This is what is known as Metropolis–Hastings algorithm. The idea of the method is the following:

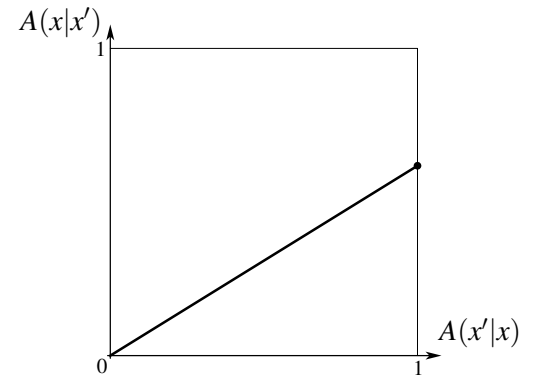
We want to sample a distribution with density  $P(x)$ . We would like to form a Markov chain whose stationary distribution is  $P(x)$  by construction, and then simulate it. A simple way to ensure that  $P(x)$  is indeed the stationary distribution of the Markov chain is making it so through detailed balance:

probability flux  $\rightarrow$  is  $A(x'|x) T(x'|x) P(x)$



probability flux  $\leftarrow$  is  $A(x|x') T(x|x') P(x')$

detailed balance:  $A(x'|x) T(x'|x) P(x) = A(x|x') T(x|x') P(x')$



<sup>12</sup> N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, E. Teller, *Equation of state calculations by fast computing machines*, J. Chem. Phys. **21** (6) 1087 (1953).

<sup>13</sup> W. K. Hastings, *Monte Carlo sampling methods using Markov chains and their applications*, Biometrika **57** (1) 97–109 (1970).

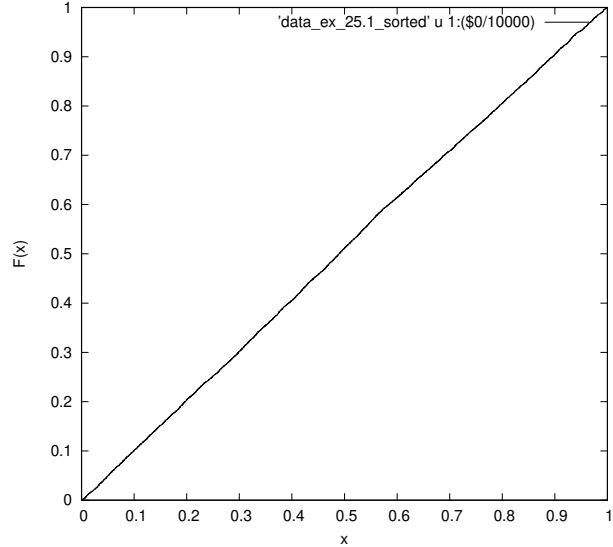
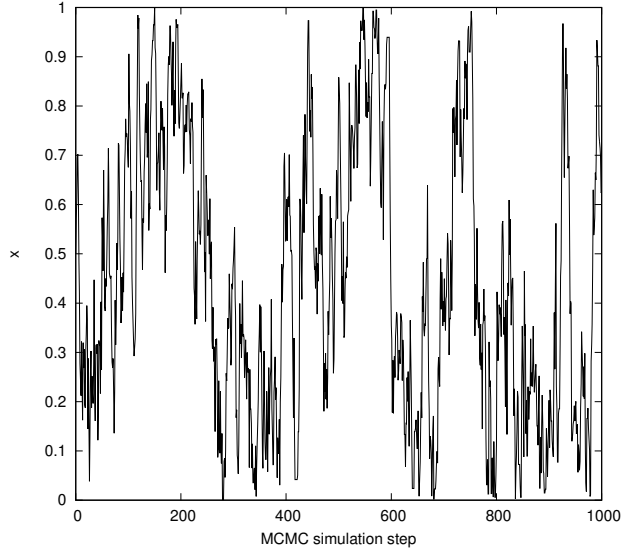
Here  $T(x'|x)$  is the distribution of the attempted next state  $x'$  of the Markov chain, given that the current state is  $x$ . The quantity  $A(x'|x)$  is the probability that when attempting the transition  $x \rightarrow x'$  we would accept it (otherwise we just stay at state  $x$ ). Obviously,  $0 \leq A \leq 1$ , and in order to speed the dynamics inside the Markov chain we would like to have the acceptance probabilities as large as possible. That is why the pair  $(A(x'|x), A(x|x'))$  should lie on the boundary of the square  $[0, 1]^2$ . We set

$$A(x'|x) = \min\left(1, \frac{P(x') T(x|x')}{P(x) T(x'|x)}\right), \quad \text{notice that if } A(x'|x) < 1, \text{ then } A(x|x') = 1$$

Note that the acceptance probabilities  $A$  depend only on the ratio of values of  $P$  (and  $T$ ). One needs to know just the shape of  $P(x)$ , but not its normalization. A version of the algorithm from 1953 did assume that  $T(x'|x) = T(x|x')$ , in this case  $A(x'|x) = \min(1, P(x')/P(x))$ .

**Example 25.1:** It is not a practically reasonable thing to do, but let us construct a MCMC for sampling the uniform on  $[0, 1]$  distribution. We have  $P(x) = 1$  for  $0 \leq x < 1$ , otherwise  $P(x) = 0$ . We choose  $T(x'|x) = \exp(-(x' - x)^2 / 2\sigma^2) / \sqrt{2\pi}\sigma$ , i.e.,  $x' = x + \sigma \cdot \zeta$ . In our MCMC simulation we always are going to have  $0 \leq x < 1$ , so  $P(x) = 1$ . The acceptance factor is  $A(x'|x) = 1$  if  $0 \leq x' < 1$ , and  $A(x'|x) = 0$  otherwise. Here is a Python code with  $\sigma = 0.1$ :

```
from random import normalvariate
x, sigma = 0.5, 0.1
for i in range(0, 10000):
    print(x)
    xp = x + normalvariate(0., sigma)
    if ((xp >= 0.) and (xp < 1.)):
        x = xp
```



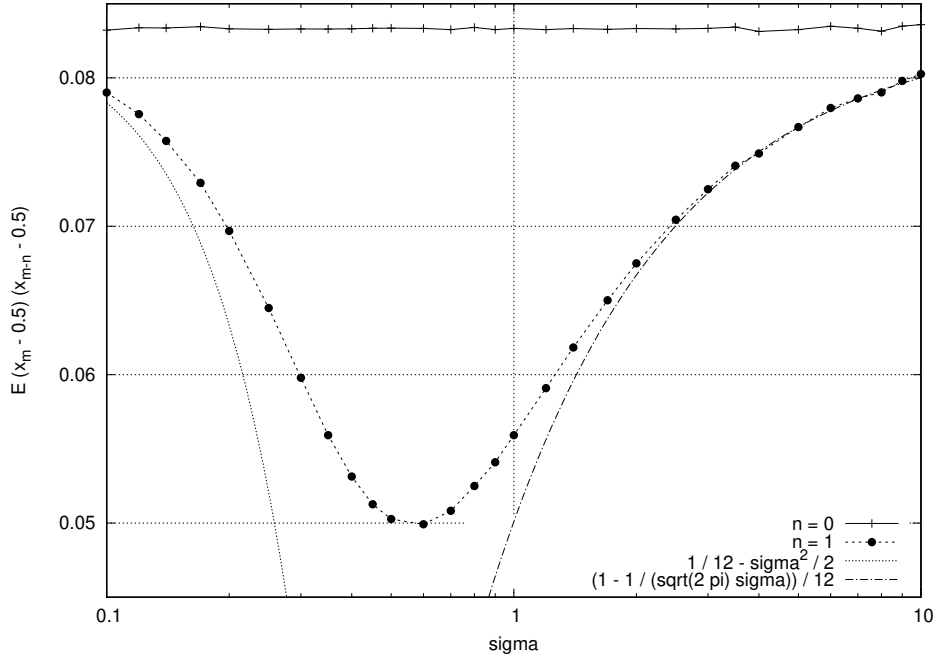
A discrete analog of that would be a Markov chain with transition matrix

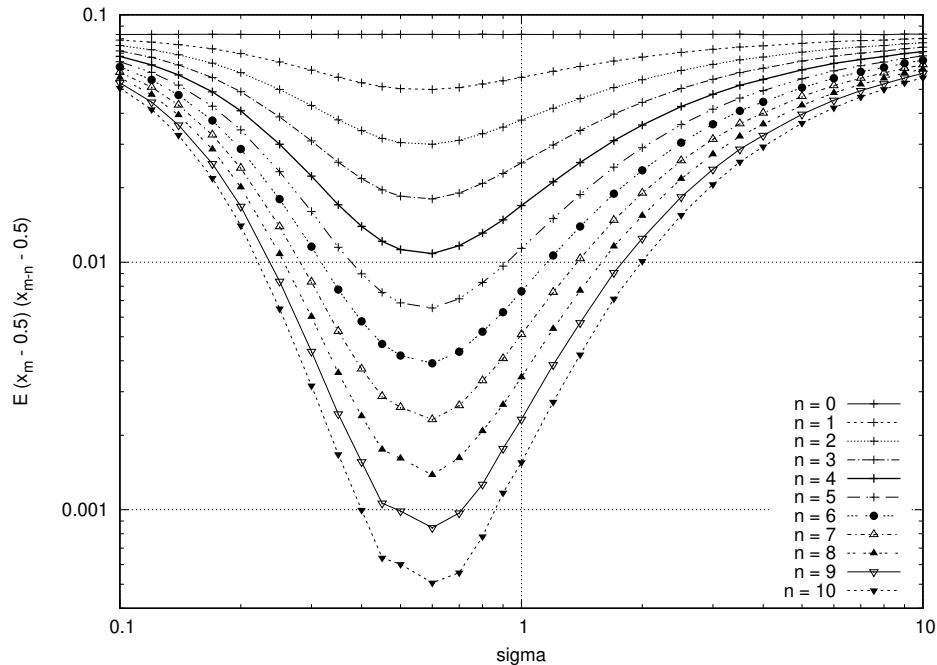
$$\hat{T} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

With probability  $\frac{1}{2}$  we move either to the left or to the right. If we try to move to the left from the most left state, we stay still. Same for the right end. The stationary distribution of this MC is  $[\frac{1}{6} \ \frac{1}{6} \ \frac{1}{6} \ \frac{1}{6} \ \frac{1}{6} \ \frac{1}{6}]$ .

What value of  $\sigma$  is the best? We would like the equilibration of the MCMC to be the fastest, *i.e.*, if  $x_m$  is the Markov chain state at the  $m^{\text{th}}$  time step, we would like  $x_{m-n}$  and  $x_m$  to be as independent as possible for even not so large  $n$ . One of the possible measures is the auto-correlation function  $K(n) = E(x_m - \frac{1}{2})(x_{m-n} - \frac{1}{2})$ . We have  $K(0) = 1/12$  as the dispersion of uniform on  $[0, 1]$  distribution. For  $K(1)$  we have ( $\xi = \sigma\xi$ ):

$$\begin{aligned}
E(x' - \tfrac{1}{2})(x - \tfrac{1}{2}) &= \int_0^1 dx \left[ \int_{-\infty}^{-x} d\xi (x - \tfrac{1}{2})^2 + \int_{-x}^{1-x} d\xi (x + \xi - \tfrac{1}{2})(x - \tfrac{1}{2}) + \int_{1-x}^{\infty} d\xi (x - \tfrac{1}{2})^2 \right] \frac{e^{-\xi^2/2\sigma^2}}{\sqrt{2\pi}\sigma} \\
&= \int_0^1 dx (x - \tfrac{1}{2})^2 + \int_0^1 dx (x - \tfrac{1}{2}) \int_{-x}^{1-x} d\xi \xi \frac{e^{-\xi^2/2\sigma^2}}{\sqrt{2\pi}\sigma} = \frac{1}{12} + \int_{-1}^1 d\xi \xi \frac{e^{-\xi^2/2\sigma^2}}{\sqrt{2\pi}\sigma} \int_{\max(0, -\xi)}^{\min(1, 1-\xi)} dx (x - \tfrac{1}{2}) \\
&= \frac{1}{12} + \int_{-1}^1 d\xi \xi \frac{e^{-\xi^2/2\sigma^2}}{\sqrt{2\pi}\sigma} \left[ \frac{\frac{1}{4} - (-\xi - \frac{1}{2})^2}{2} \chi_{\xi < 0} + \frac{(\frac{1}{2} - \xi)^2 - \frac{1}{4}}{2} \chi_{\xi > 0} \right] \\
&= \frac{1}{12} + \int_{-1}^1 d\xi \xi \frac{e^{-\xi^2/2\sigma^2}}{\sqrt{2\pi}\sigma} \left[ \frac{|\xi|}{2} - \frac{\xi}{2} \right] = \frac{1}{12} - \int_0^1 d\xi \frac{e^{-\xi^2/2\sigma^2}}{\sqrt{2\pi}\sigma} \xi^2 (1 - \xi) = K(1)
\end{aligned}$$





**Example 25.2:** Consider a distribution function  $F(x) = \sqrt{x}/(1 + \sqrt{x})$ ,  $x \geq 0$ , with density function  $P(x) = 1/2\sqrt{x}(1 + \sqrt{x})^2$ . Let us sample this distribution by Metropolis–Hastings algorithm in 2 different ways:

First, let us choose  $T(x'|x) = \exp(-(x' - x)^2/2\sigma^2)/\sqrt{2\pi}\sigma$ , with  $\sigma = 0.1$ :

```
from math import sqrt; from random import random, normalvariate
def P(x):
    if (x <= 0.):
        return 0.
    else:
        return 1. / (2. * sqrt(x) * (1 + sqrt(x))**2)

x, sigma = 1., 0.1
for i in range(0, 1000000):
    print(x)
    xp = x + normalvariate(0., sigma)
    if (random() < (P(xp) / P(x))):
        x = xp
```

Second, let us choose  $T(x'|x) = \exp(-(x' - x)^2/2x^2)/\sqrt{2\pi}x$ . Here  $T(x'|x) \neq T(x|x')$  here, and the generalization from 1970 is relevant:

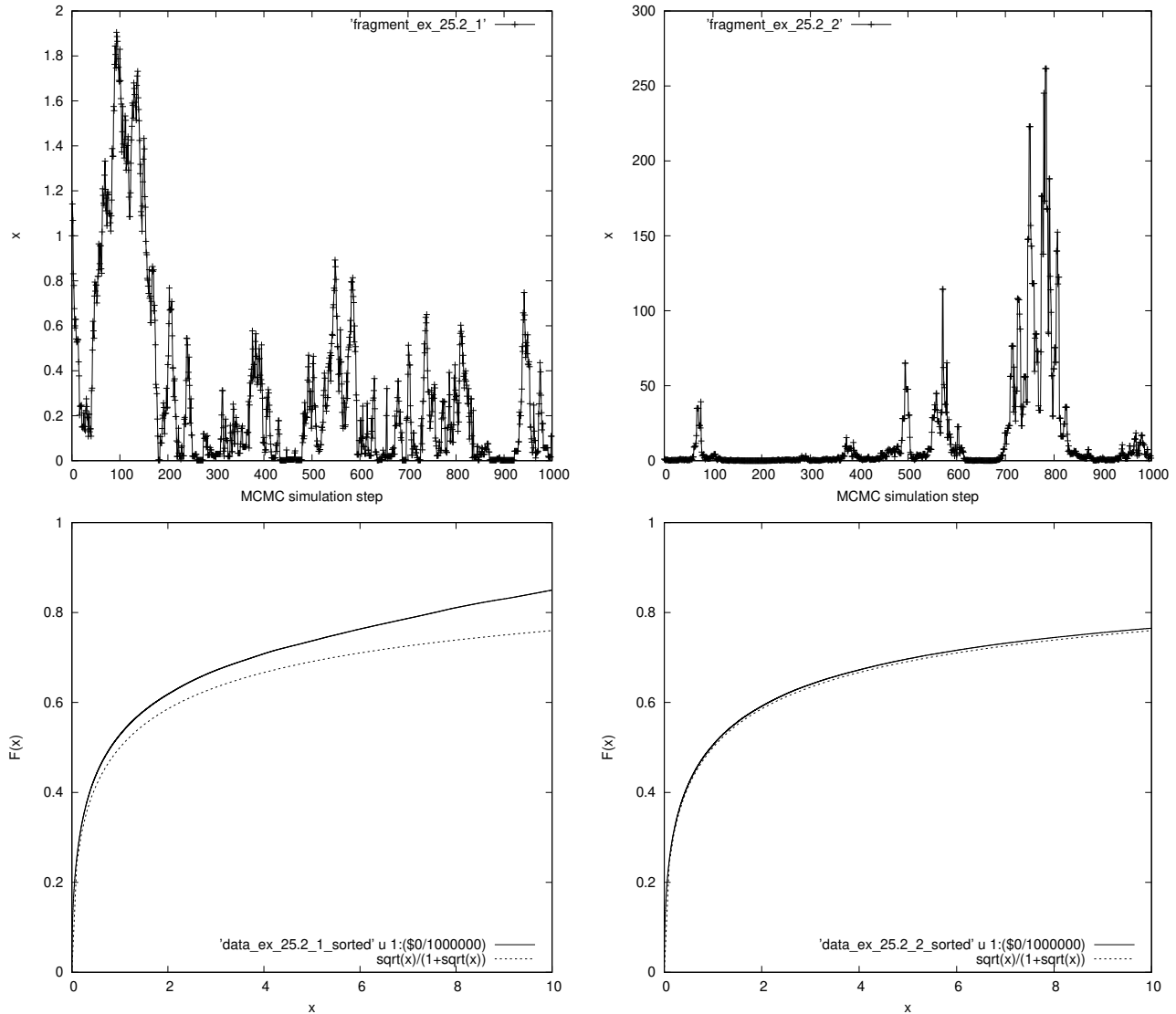
```
from math import sqrt, exp; from random import random, normalvariate
def P(x):
    if (x <= 0.):
        return 0.
    else:
        return 1. / (2. * sqrt(x) * (1 + sqrt(x))**2)

x = 1.
for i in range(0, 1000000):
```

```

print(x)
xp = x + normalvariate(0., x)
if (xp > 0.):
    TT = (x / xp) * exp(((xp - x)**2 / 2.) * (x**(-2) - xp**(-2)))
    if (random() < (P(xp) / P(x)) * TT):
        x = xp

```



The largest observed value of  $x$  in the MCMC simulation with  $10^6$  steps in the first/second way was 33.36.../about  $5 \cdot 10^7$ .

## 25.1 Ising model

See, e.g., [W. Janke, Monte Carlo methods in classical statistical physics](#).

Imagine we have a joint distribution of many random variables  $P(x_1, x_2, \dots, x_N)$ . *Gibbs sampling* is a MCMC algorithm that starts from some initial  $\mathbf{x}$ , and then at each step we 1) randomly (or in some pre-defined order) choose a variable  $x_k$  from  $x_1, x_2, \dots, x_N$ ; and 2) set  $x_k$  according to the

conditional distribution  $P(x_k | x_1, x_2, \dots, x_{k-1}, x_{k+1}, \dots, x_N)$ . Such sampling could be convenient when the conditional distribution of one (or not so many) variable(s) can be easily computed (while the whole  $P(\mathbf{x})$  is not).

Gibbs sampling is a partial case of Metropolis–Hastings algorithm, where the transitional probabilities  $T(x'|x)$  are non-zero only if just one variable has different value in the pair of states  $x$  and  $x'$ . (One can think that a move is rejected in the Gibbs sampling MCMC, if the value of a variable  $x_k$  didn't change after being chosen according to the conditional distribution.)

Consider a  $D$ -dimensional integer lattice  $\mathbf{Z}^D$ , with each site  $\mathbf{n} = (n_1, n_2, \dots, n_D)$  containing a binary variable  $\sigma_{\mathbf{n}} = \pm 1$ . The values of all the binary variables, a vector  $\boldsymbol{\sigma}$ , form a “state”, and we introduce the following distribution over the states:

$$P(\boldsymbol{\sigma}) := \frac{\exp(-E(\boldsymbol{\sigma})/T)}{Z(T)}, \quad E(\boldsymbol{\sigma}) := - \sum_{\mathbf{n} \in \mathbf{Z}^D} \sum_{i=1}^D \sigma_{\mathbf{n}} \sigma_{\mathbf{n}+\mathbf{e}_i}$$

For this expressions to have sense, one should consider them on a finite part of the  $\mathbf{Z}^D$  lattice, let us say of size  $L$ , with appropriate boundary conditions. (It is possible to properly define a thermodynamics limit  $L \rightarrow \infty$ .) The quantity  $Z(T)$ , or so called partition function, is hard to calculate. It is connected to [Helmholtz] free energy as  $F = -T \ln Z$ , where  $T = 1/\beta$  is temperature, or  $Z = \sum_i e^{-\beta E_i} = e^{-\beta F}$ .

Here is the program in C that samples  $P(\boldsymbol{\sigma})$  for 2D Ising model using Gibbs sampling (which for Ising model is also called Glauber algorithm/dynamics<sup>14</sup> or heat bath algorithm):

$$S = \sigma_{\mathbf{n}+\mathbf{e}_1} + \sigma_{\mathbf{n}-\mathbf{e}_1} + \sigma_{\mathbf{n}+\mathbf{e}_2} + \sigma_{\mathbf{n}-\mathbf{e}_2}, \quad P(\sigma_{\mathbf{n}}) = \frac{\exp(\sigma_{\mathbf{n}} S/T)}{\exp(S/T) + \exp(-S/T)}$$

```
[...]/teaching/2020-1/math_575b/notes/Monte_Carlo/Ising_model$ ls
Ising.c
[...]/teaching/2020-1/math_575b/notes/Monte_Carlo/Ising_model$ cat Ising.c
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
#define L 256
#define FOR_ALL_SITES for (i = 0; i < L; i++) for (j = 0; j < L; j++)
int main(void) { gsl_rng * RNG; FILE *out; char name[32];
    int i, j, t, m, S[L][L], n[L], p[L]; double u, T = 3., E;
    RNG = gsl_rng_alloc(gsl_rng_ranlux389); p[0] = L - 1; n[L - 1] = 0;
    for (i = 0; i < L - 1; i++) { n[i] = i + 1; p[i + 1] = i; }

    FOR_ALL_SITES S[i][j] = gsl_rng_get(RNG) % 2;

    for (t = 0; t < 500; t++) { sprintf(name, "%03d.pgm", t);
        out = fopen(name, "w"); fprintf(out, "P5 %d %d 255\n", L, L);
        FOR_ALL_SITES fputc(255 * S[i][j], out); fclose(out);

        for (m = 0; m < 81920; m++) {
            i = gsl_rng_get(RNG) % L; j = gsl_rng_get(RNG) % L;
            E = 2. * (double) (S[n[i]][j] + S[p[i]][j] + S[i][n[j]] + S[i][p[j]] - 2);
            S[i][j] = (gsl_rng_uniform(RNG) < 1. / (1 + exp(-2. * E / T))); } }
```

<sup>14</sup> R. J. Glauber, *Time-dependent statistics of the Ising model*, J. Math. Phys. **4** (2) 294–307 (1963).

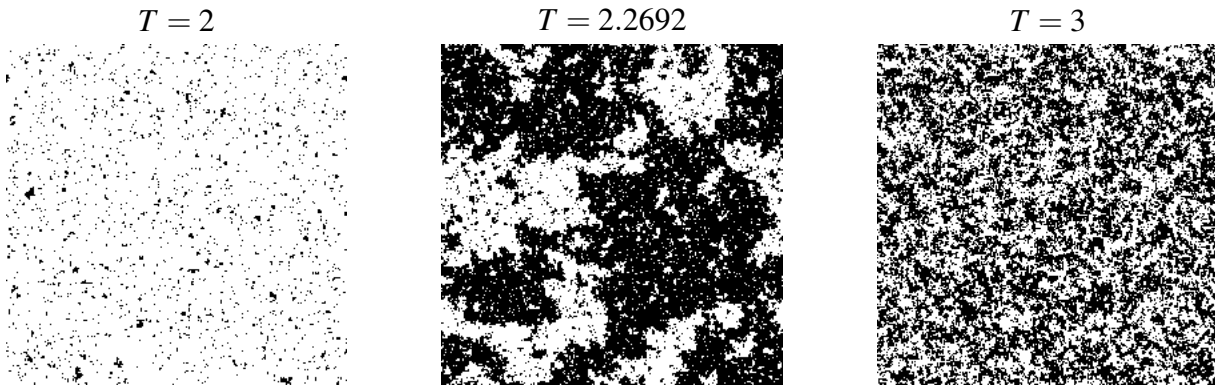
```

    gsl_rng_free(RNG); return 0; }
[...]/teaching/2020-1/math_575b/notes/Monte_Carlo/Ising_model$ cc Ising.c -lm -l
gsl ; ./a.out ; ffmpeg -r 25 -f image2 -s 256x256 -i %03d.pgm -vcodec libx264 -c
rf 30 -pix_fmt yuv420p -loglevel error Ising.mp4 ; rm *.pgm ; ls
a.out Ising.c Ising.mp4
[...]/teaching/2020-1/math_575b/notes/Monte_Carlo/Ising_model$

```

It could be argued that the dynamics arising in this Markov chain is not unreasonable dynamics of the corresponding would be a system of “spins” (the dynamics in general tries to reduce energy, attempts to be in thermal equilibrium with temperature  $T$ ).

The typical state in the MCMC simulation of the 2D Ising model looks like (left/middle/right is corresponding to temperature below/at/above the phase transition)



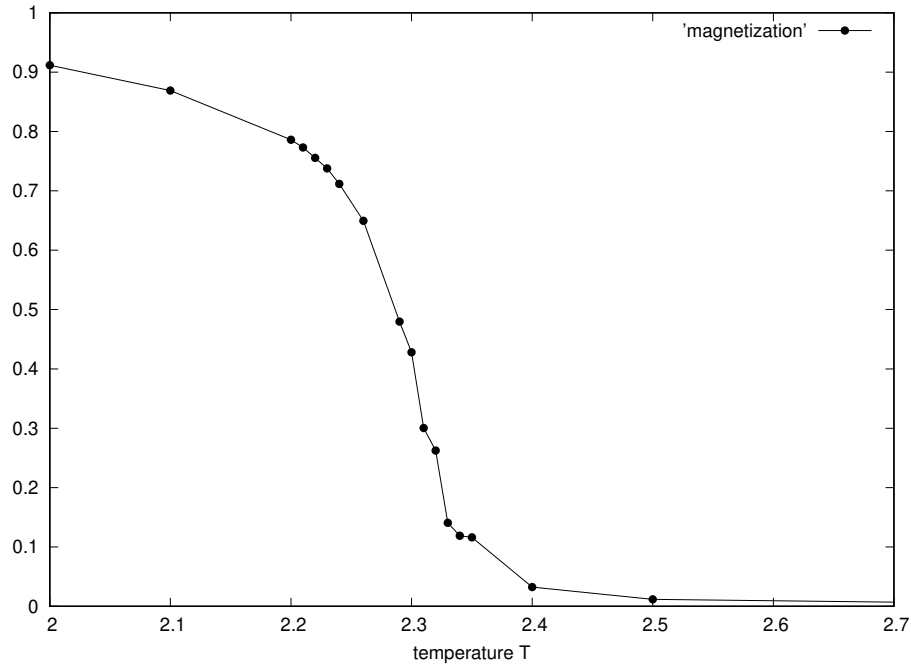
Of course, we would be interested in how fast the mixing within the state space is happening. We do not have any chance to visit a notable share of states, as the number of states is the  $2^{\text{number of spin sites}}$ . We visit a large number of states and hope that it is representative enough for our statistical purposes. Imagine we look after some quantity  $A(\sigma)$  and check how it does depend on time. We can introduce a so called autocorrelation function

$$\text{normalized } K_A(\tau) := \frac{\langle A_t A_{t+\tau} \rangle - \langle A_t \rangle^2}{\langle A_t^2 \rangle - \langle A_t \rangle^2}, \quad K_A(0) = 1$$

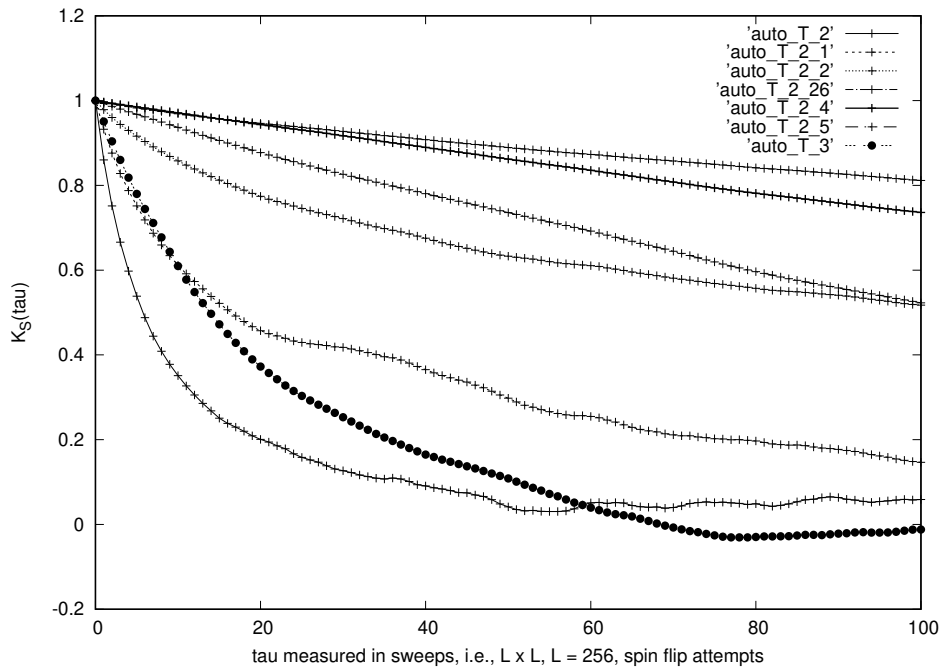
How fast  $K_A(\tau)$  decays with  $\tau$  is our estimation about how uncorrelated/independent are our samples of  $P(\sigma)$ .

Let us choose the quantity  $S = \sum_n \sigma_n$  — the sum of all the spins. Its average value divided by the number of spins is called magnetization  $M$ . (As  $K_A(\tau)$  is normalized, we have  $K_S(\tau) \equiv K_M(\tau)$ .)

Here is how “average” (I’m starting the MCMC with all spins up) magnetization depends on temperature  $T$ :

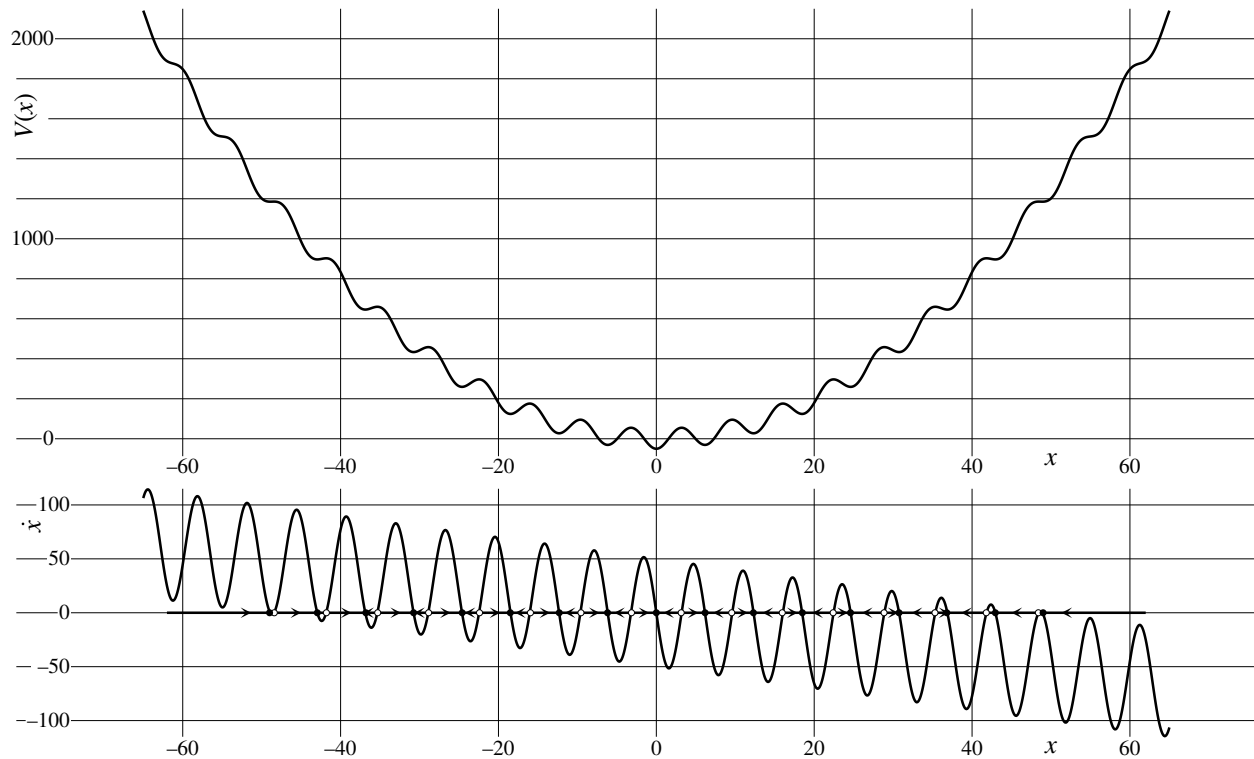


Here is how  $K_S(\tau)$  falls down with  $\tau$ :



## 25.2 Stochastic optimization

**Example 25.2.1:** Consider  $V(x) = x^2/2 - 50\cos x$ . To minimize it we can try the gradient descent method,  $dx/dt = -dV(x)/dx = -x - 50\sin x$ .

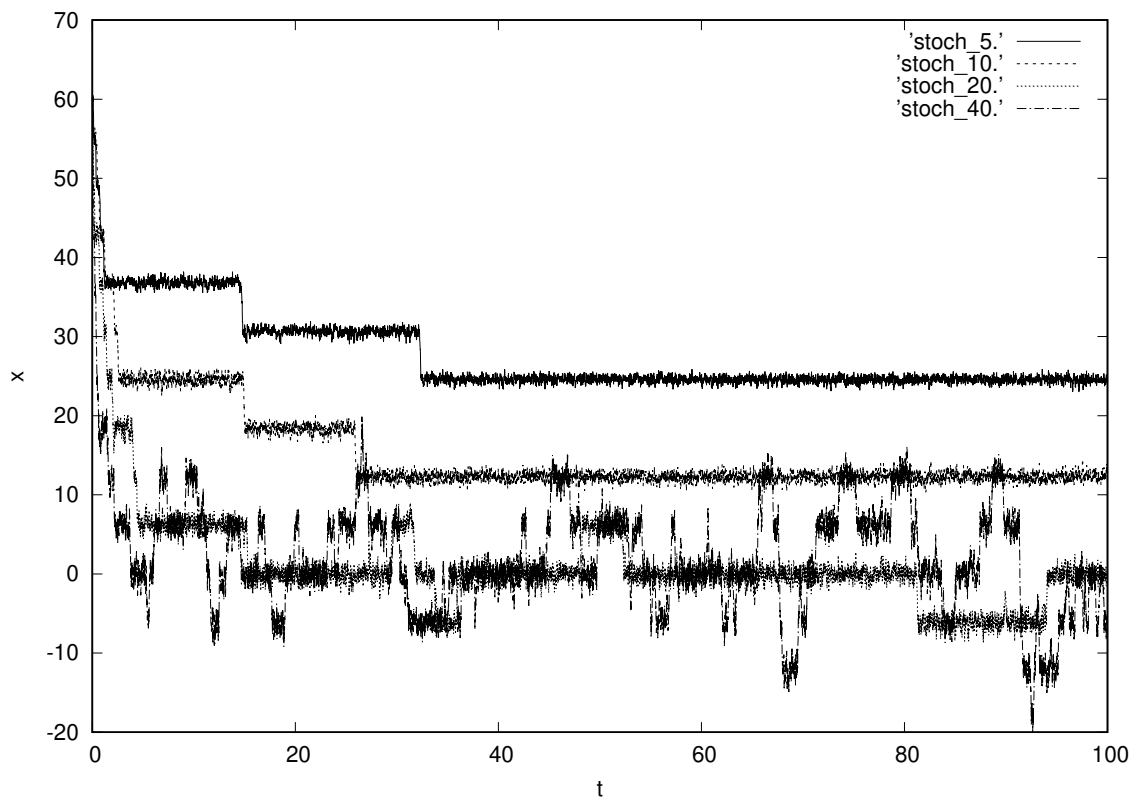


Here is a Python script that updates  $x$  according to the rule  $x(t + dt) := x(t) - dt \cdot V'(x) + \sqrt{2Ddt}\zeta$ , where  $\zeta \sim N(0, 1)$ . The case  $D = 0$  would correspond to standard gradient descent (its ODE formulation with forward Euler method). When  $D > 0$ , but  $\gamma = 1$ , we use additive noise in the updates of  $x$  in order to not be stuck in shallow minima of  $V(x)$ . The larger  $D$  is, the deeper are the minima we can realistically climb out of. In order to eventually converge/freeze, the diffusion coefficient  $D$  is gradually decreased (like in [simulated annealing](#)),  $D(t + dt) := \gamma \cdot D(t)$ .

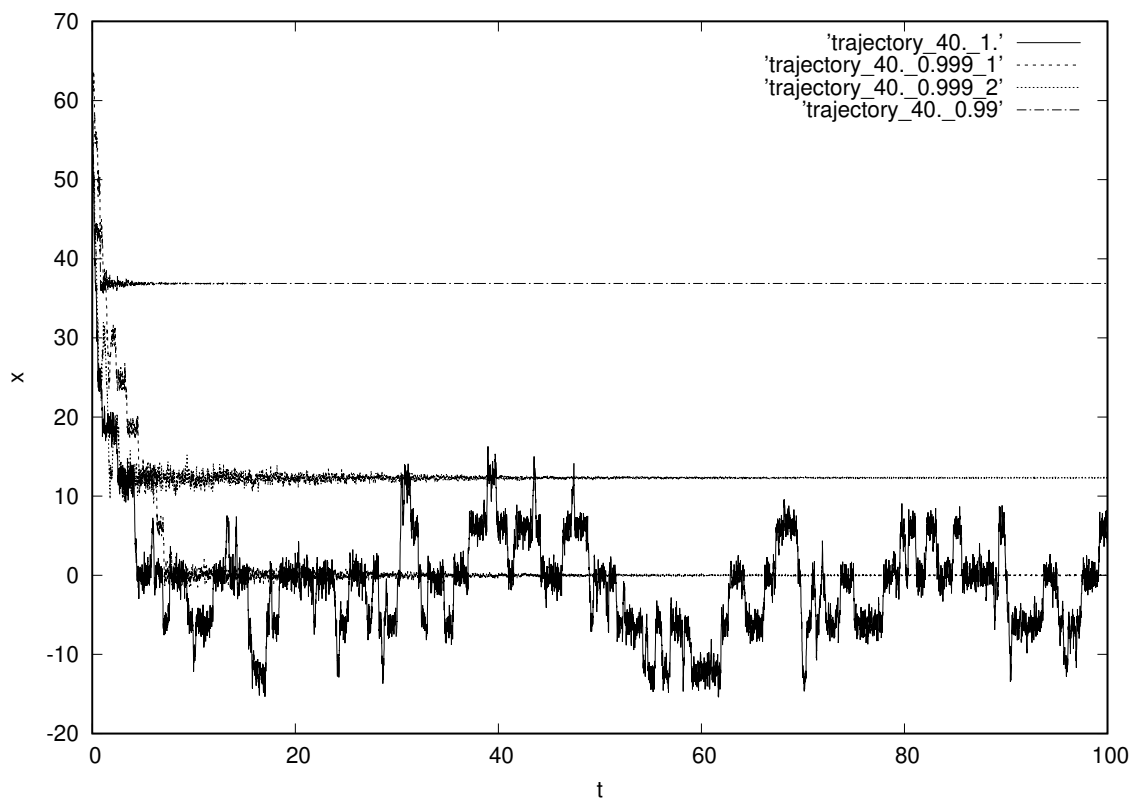
```
from sys import argv
from math import cos, sin, sqrt
from random import normalvariate
def f(x):
    return 0.5 * x**2 - 50 * cos(x)
def df(x):
    return x + 50 * sin(x)

x, t, dt, D, gamma = 60., 0., 0.01, float(argv[1]), float(argv[2])
best_x, best_f = x, f(x)
print(t, x, f(x), df(x), best_x, best_f)
while (t < 100.):
    x, t, D = x - dt * df(x) + normalvariate(0., sqrt(2.*D*dt)), t + dt, gamma * D
    if (f(x) < best_f):
        best_x, best_f = x, f(x)
    print(t, x, f(x), df(x), best_x, best_f)
```

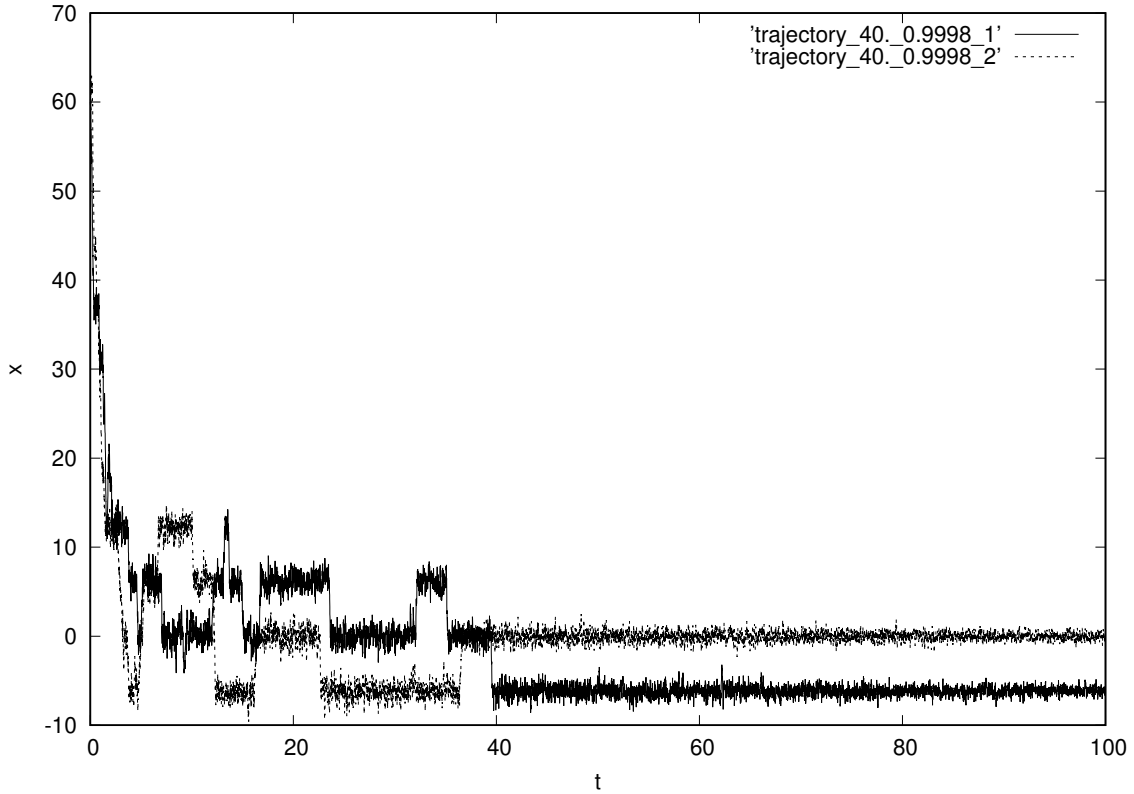
$\gamma = 1$ :



various  $\gamma$ 's:



$\gamma = 0.9998$ :



## Problems and exercises

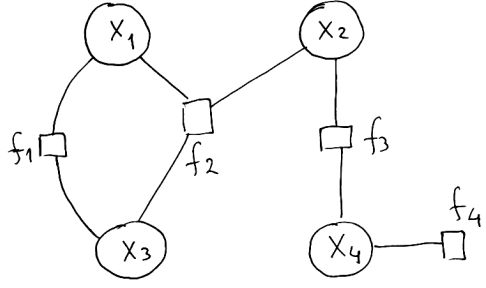
1. Using Metropolis algorithm, simulate a Markov chain with  $P(x) = \exp(-x)$ ,  $x \geq 0$  density and transition function  $T(x'|x) = \exp(-(x' - x)^2/2\sigma^2)/\sqrt{2\pi}\sigma$  (with appropriate acceptance probability  $A(x'|x)$ ). Find  $\sigma$  that minimizes  $K_{xx}(1) = E(x_m - 1)(x_{m-1} - 1)$ , where  $x_m$  is the state of the Markov chain at time  $m$ .

2. Using either Metropolis algorithm or Glauber dynamics, simulate a MCMC for 2D Ising model on a  $L \times L$  torus of binary variables/spins. Compute and plot as a function of temperature  $T$  (e.g., for  $2 \leq T \leq 2.6$ ) the average value of  $m(T) = |\sum_n \sigma_n|/L^2$  for  $L = 32, 64$ , and  $128$ . Is the transition from non-zero to [almost] zero value of  $m(T)$  becomes sharper for larger  $L$ ?

## 26 Message passing/belief propagation

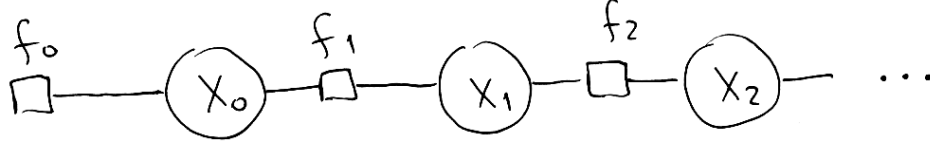
A factor graph  $(X, F, E)$  is a bipartite graph, whose vertices are separated into two groups: variable vertices  $X = (x_1, x_2, \dots, x_N)$  and factor vertices  $F = (f_1, f_2, \dots, f_m)$ . For each factor vertex  $1 \leq \alpha \leq M$  let us consider a subset of variable vertices  $X_\alpha \subseteq X$  that are connected to  $f_\alpha$  by an edge. Similarly, for each  $1 \leq i \leq N$  let  $F_i$  be the subset of factor vertices that are connected to  $i$ .<sup>15</sup> Let us associate with a factor graph  $(X, F, E)$  a factorized density distribution function  $g(X) = (1/Z) \prod_{\alpha=1}^M f_\alpha(X_\alpha)$ , where  $Z$  is the normalization factor ensuring  $\sum_X g(X) = 1$ .

<sup>15</sup> A similar structure would be a [hypergraph](#), where [hyper]edges could connect an arbitrary subset of vertices.



$$g(x_1, x_2, x_3, x_4) \propto f_1(x_1, x_3) f_2(x_1, x_2, x_3) f_3(x_2, x_4) f_4(x_4)$$

The state evolution in Markov chain could be expressed by a factor graph:



Here  $f_t(x_{t-1}, x_t)$  is the transition function with  $\int dx_t f_t(x_{t-1}, x_t) = 1$  (or a constant not depending on  $x_{t-1}$ ). Integrating over  $x_{t+1}x_{t+2}x_{t+3}\dots$  we would get the marginal distribution  $g(x_0, x_1, \dots, x_t) \propto f_0(x_0)f_1(x_0, x_1)f_2(x_1, x_2)\dots f_t(x_{t-1}, x_t)$ .

Imagine we would like to find marginal distributions  $g(x_i) = \sum_{X \setminus x_i} g(X)$ . To compute them directly could be prohibitively expensive, as there could be too much terms in the sum. If, e.g., the variables  $x_i$  are binary, the the sum has  $2^{N-1}$  terms.

Belief propagation<sup>16</sup> is an approximate algorithm of computing marginals distributions  $g_i(x_i) = \sum_{X \setminus x_i} g(X)$ . On our factor graph we pass messages from the variable vertices to the factor ones and back, with the messages being calculated locally at each vertex according to  $(\mu_{i \rightarrow \alpha}^{(-1)} := 1)$

$$\mu_{\alpha \rightarrow i}^{(t)}(x_i) := \sum_{x_{X_\alpha \setminus i}} f_\alpha(X_\alpha) \prod_{j \in X_\alpha \setminus i} \mu_{j \rightarrow \alpha}^{(t-1)}(x_j), \quad \mu_{i \rightarrow \alpha}^{(t)}(x_i) := \prod_{\beta \in F_i \setminus \alpha} \mu_{\beta \rightarrow i}^{(t)}(x_i)$$

At each iteration the marginal distribution  $g_i(x_i)$  is approximated by  $g_i(x_i) \propto \prod_{\alpha \in F_i} \mu_{\alpha \rightarrow i}^{(t)}(x_i)$ . If we put this expression to the definition of what the messages are, then we get

$$g_i(x_i) \propto \mu_{i \rightarrow \alpha}(x_i) \mu_{\alpha \rightarrow i}(x_i) = \sum_{x_{X_\alpha \setminus i}} f_\alpha(X_\alpha) \underbrace{\prod_{j \in X_\alpha} \mu_{j \rightarrow \alpha}(x_j)}_{\propto g_{X_\alpha}(x_{X_\alpha})}$$

$$g(X) \propto \prod_{\alpha} f_{\alpha}(x_{X_{\alpha}}) = \prod_{\alpha} \frac{g_{X_{\alpha}}(x_{X_{\alpha}})}{\prod_{i \in X_{\alpha}} \mu_{i \rightarrow \alpha}(x_i)} = \frac{\prod_{\alpha} g_{X_{\alpha}}(x_{X_{\alpha}})}{\prod_{i \in \alpha} \frac{g_i(x_i)}{\mu_{\alpha \rightarrow i}(x_i)}} = \frac{\prod_{\alpha} g_{X_{\alpha}}(x_{X_{\alpha}})}{\prod_i (g_i(x_i))^{|F_i|-1}}$$

## 26.1 Error correcting codes

A simple example of an error correcting code would be a [spelling alphabet](#), where instead of one letter we say [through a noisy phone connection] the whole word that starts from it.

<sup>16</sup>R. Gallager (1963), J. Pearl (1982), D. J. C. MacKay, R. M. Neal (1996)

Another example is a repetition code, each bit is sent, *e.g.*, 3 times, while on the other end the decoding is done by majority rule. To transmit “0”, we send “000” through the communication channel, and decode “000”, “001”, “010”, “100” outputs as “0” (and similarly for “1”). If the probability of a bit to be flipped by the channel is  $p \ll 1$ , then the error probability after the decoding is  $3p^2(1-p) + p^3 = 3p^2 - 2p^3 \approx 3p^2 \ll p$ . The probability of error  $p \rightarrow 3p^2$  can be greatly reduced.

Our whole language is redundant in order for communication to be reliable, and the following once popular joke is one of many demonstrations of that:

Arocdnicg to rsceearch at Cmabrigde Uinervtisy, it deosn't mtttaer in waht oredr the ltteers in a wrod are, the olny iprmoatnt tihng is taht the frist and lsat ltteer are in the rghit pcale. The rset can be a toatl mses and you can sitll raed it wouthit pobelrm. Tihs is buseace the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe.

We can read partially incomplete text, correct typos, *etc.*

In case of a linear binary error correcting code, an encoded (thus longer due to the added redundancy) message can be viewed as a block of  $N$  bits taking values 0 and 1. These are the variable vertices in the corresponding factor graph (often also called Tanner graph of the code). The encoded message satisfied  $M_{PC}$  parity checks, which are some of the factor vertices. Each parity check factor vertex is connected to the bits that are participating in this parity check. Another  $N$  factor vertices (in case of message distortion by the communication channel to be independent from bit to bit) are attached in one-to-one fashion to the bits — they correspond to the statistics of the channel output.

The factor nodes corresponding to the parity checks have the function

$$f_{\alpha}(x_1, x_2, \dots, x_k) = \begin{cases} 1, & x_1 + x_2 + \dots + x_k \equiv 0 \pmod{2} \\ 0, & x_1 + x_2 + \dots + x_k \equiv 1 \pmod{2} \end{cases} \quad \begin{array}{l} \text{parity is satisfied} \\ \text{parity is not satisfied} \end{array}$$

This makes the summing over  $X$  [with  $g(X)$  in mind] going over only such configurations of  $x_1, x_2, \dots, x_N$  that do satisfy all the parity checks (so called codewords).

In the case of  $x_i$ ,  $1 \leq i \leq N$ , being binary variables, taking, *e.g.*, the values 0 and 1, the distribution of one such variable can be fully described by the so called logarithmic likelihood  $m_i = (1/2) \ln(g_i(0)/g_i(1))$ . The message passing becomes (here  $\eta := (1/2) \ln(\mu(0)/\mu(1))$ )<sup>17</sup>

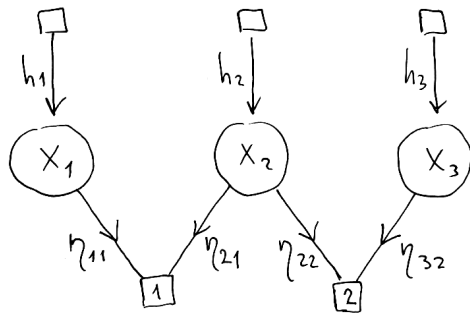
$$\eta_{\alpha \rightarrow i}^{(t)} = \frac{1}{2} \ln \left( \frac{\sum_{X_{\alpha}, \text{parity}, x_i=0} \prod_{j \in X_{\alpha} \setminus i} \mu_{j \rightarrow \alpha}^{(t-1)}(x_j)}{\sum_{X_{\alpha}, \text{parity}, x_i=1} \prod_{j \in X_{\alpha} \setminus i} \mu_{j \rightarrow \alpha}^{(t-1)}(x_j)} \right), \quad \eta_{i \rightarrow \alpha}^{(t)}(x_i) := h_i + \sum_{\beta \in F_i \setminus \alpha} \eta_{\beta \rightarrow i}^{(t)}, \quad m_i^{(t)} := h_i + \sum_{\alpha \in F_i} \eta_{\alpha \rightarrow i}^{(t)}$$

$$\eta_{\alpha \rightarrow i}^{(t)} = \text{arctanh}((\Sigma_0 - \Sigma_1)/(\Sigma_0 + \Sigma_1)) = \text{arctanh} \left( \prod_{j \in X_{\alpha} \setminus i} \tanh \eta_{j \rightarrow \alpha}^{(t)} \right)$$

$$\frac{\mu(0)v(0) + \mu(1)v(1) - \mu(0)v(1) - \mu(1)v(0)}{\mu(0)v(0) + \mu(1)v(1) + \mu(0)v(1) + \mu(1)v(0)} = \frac{\mu(0) - \mu(1)}{\mu(0) + \mu(1)} \cdot \frac{v(0) - v(1)}{v(0) + v(1)}$$

**Example 26.1:** Consider a repetition code

<sup>17</sup>  $\text{arctanh}(x) = (1/2) \ln((1+x)/(1-x))$



The numbers  $h_1$ ,  $h_2$ , and  $h_3$  are logarithmic likelihoods from the output of the communication channel. Positive/negative value of  $h$  means that 0/1 is more probable. The magnitude of  $h$  is corresponding to how much more probable.

The parity checks  $f_1$  and  $f_2$  make the only allowed configurations of bits being “000” and “111”.

```
[...]/teaching/2020-1/math_575b/notes/message_passing$ cat MP.m
function [eta] = MP(h, eta)
    eta11 = h(1);
    eta21 = h(2) + eta(4);    % eta(4) = eta32
    eta22 = h(2) + eta(1);    % eta(1) = eta11
    eta32 = h(3);
    m1 = h(1) + eta21;
    m2 = h(2) + eta11 + eta32;
    m3 = h(3) + eta22;
    [m1 m2 m3]
    eta = [eta11, eta21, eta22, eta32];
[...]/teaching/2020-1/math_575b/notes/message_passing$ octave-cli
GNU Octave, version 4.4.1
[... copyright notice and links ...]
octave:1> format compact
octave:2> MP([1, 3, -2], [0, 0, 0, 0])
ans =
    4    2    1

ans =
    1    3    3   -2

octave:3> MP([1, 3, -2], [1, 3, 3, -2])
ans =
    2    2    2

ans =
    1    1    4   -2

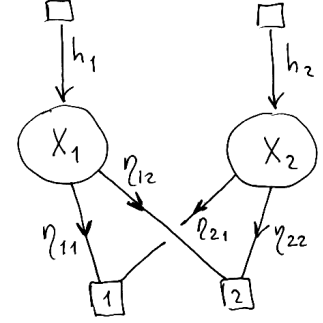
octave:4> MP([1, 3, -2], [1, 1, 4, -2])
ans =
    2    2    2

ans =
    1    1    4   -2
```

Here there are no  $\text{arctanh}(\cdot)$  functions because the product of  $\tanh(\cdot)$ 's inside it always contains just one factor, and “arctanh” and “tanh” eat/kill/cancel each other.

**Example 26.2:** Consider a code with 2 bits and 2 parity checks each connected to both of the bits. The allowed configurations are “00” and “11”, the parity checks are redundant, *i.e.*, they double each other. The message passing equations are  $\eta_{11}^{(t)} = h_1 + \eta_{22}^{(t-1)}$ ,  $\eta_{12} \leftarrow h_1 + \eta_{21}$ ,  $\eta_{21} \leftarrow h_2 + \eta_{12}$ , and  $\eta_{22} \leftarrow h_2 + \eta_{11}$ , with the decoding output being  $m_1 = h_1 + \eta_{21} + \eta_{22}$ ,  $m_2 = h_2 + \eta_{11} + \eta_{12}$ .

From these equations we have, for example,  $\eta_{11} + \eta_{22} \leftarrow h_1 + h_2 + \eta_{11} + \eta_{22}$ , which means that  $\eta_{11}^{(t)} + \eta_{22}^{(t)} = t \cdot (h_1 + h_2)$ .



## Part VII

# Machine learning

## 27 Regression

Consider we have a bunch of data of type  $(\mathbf{x}_i, y_i)$ , and we want to come up with a function  $f(\mathbf{x})$  such that  $f(\mathbf{x}_i) \approx y_i$ .

We are not necessarily targeting for  $f(\mathbf{x}_i) = y_i$  exactly, as the values  $y_i$  may contain some noise of measurement error, so we assume that  $y_i$  is a “true” function  $f_{\text{true}}(\mathbf{x})$  distorted in a certain way. In order to have an idea how well a given approximation  $f(\mathbf{x})$  works, we need to assume a certain statistics of the distortion  $P(y | f_{\text{true}}(\mathbf{x}))$ . A popular assumption is that  $y_i = f_{\text{true}}(\mathbf{x}) + \xi_i$ , where  $\xi_i \sim N(0, \sigma^2)$  is an additive noise[, and  $\xi_i$  for different  $i$  are independent].

Let us say, we decide to choose our function approximating the data from a family of functions  $f(\mathbf{x}, \boldsymbol{\theta})$ , where  $\boldsymbol{\theta}$  is the vector of parameters. The task of choosing a good function is now the task of choosing a suitable vector of parameters  $\boldsymbol{\theta}$ .

One approach to find the values of the parameters  $\boldsymbol{\theta}$  is called *maximum likelihood*. We set

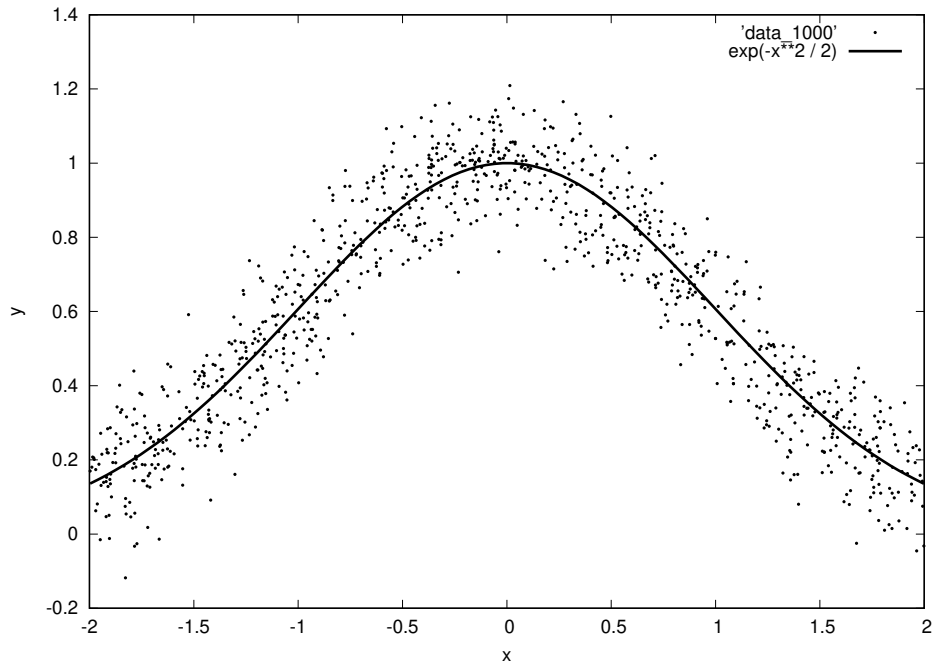
$$\boldsymbol{\theta}_{\text{ML}} := \arg \max_{\boldsymbol{\theta}} \underbrace{\prod_{i=1}^N P(y_i | f(\mathbf{x}_i, \boldsymbol{\theta}))}_{\text{likelihood}}, \quad \boldsymbol{\theta}_{\text{ML}} = \arg \min_{\boldsymbol{\theta}} \underbrace{\sum_{i=1}^N (y_i - f(\mathbf{x}_i, \boldsymbol{\theta}))^2}_{\text{loss function}}$$

where  $(\mathbf{x}_i, y_i)$ ,  $i = 1, 2, \dots, N$  is the data from which we estimate  $\boldsymbol{\theta}$ .

The case  $f(\mathbf{x}, \boldsymbol{\theta}) = \sum_{m=1}^M \theta_m f_m(\mathbf{x})$  is called linear regression. In case of additive Gaussian noise the maximum likelihood is a least squares problem:  $\boldsymbol{\theta}_{\text{ML}} = \arg \min_{\boldsymbol{\theta}} \sum_{i=1}^N (y_i - \sum_{m=1}^M \theta_m f_m(\mathbf{x}_i))^2$ , *i.e.*,  $\boldsymbol{\theta}_{\text{ML}}$  is found from the minimization of a quadratic function of  $\boldsymbol{\theta}$ .

**Example 27.1:** Consider data being generated by a Python script

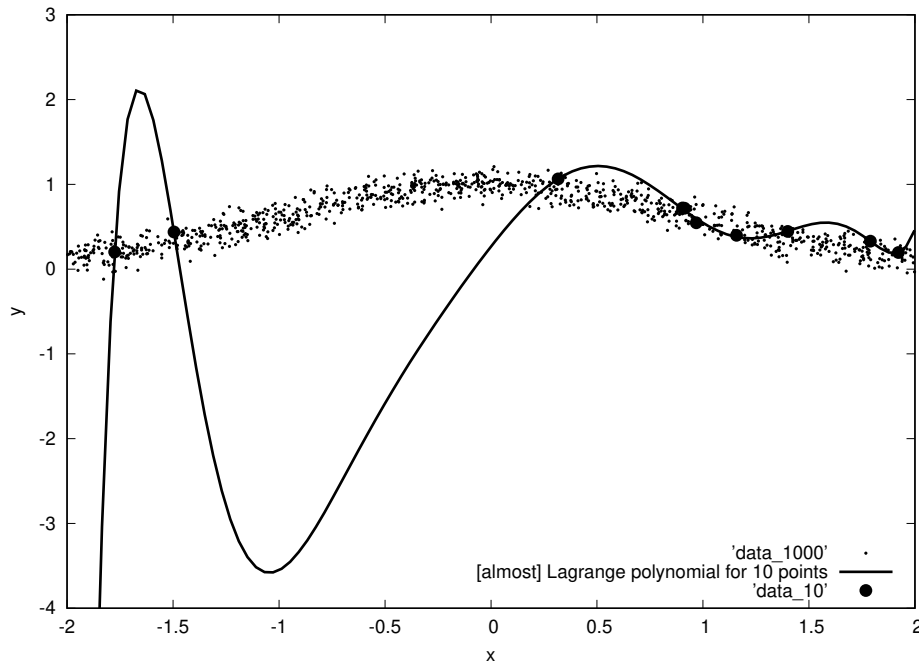
```
from math import exp; from random import random, normalvariate
for i in range(0, 1000):
    x = 4. * random() - 2.
    y = exp(-0.5 * x**2) + normalvariate(0., 0.1)
    print(x, y)
```



Imagine we take the first 10 points of the data and then try to fit them by the 9<sup>th</sup> degree polynomial  $f(x, \theta) = \sum_{m=0}^9 \theta_m x^m$ :

```
from sys import argv; import numpy as np; from scipy.optimize import minimize
global xy
def loss(theta):
    loss, N, M = 0., xy.shape[0], theta.shape[0]
    for i in range(0, N):
        y = 0.
        for m in range(0, M):
            y += theta[m] * xy[i, 0]**m
        loss += (xy[i, 1] - y)**2
    return loss

xy = np.loadtxt(argv[1])
res = minimize(loss, np.zeros(10), method='BFGS', jac = None)
print(res.x)
```



We then have a situation that is called *overfitting*. The suggested function  $f(x, \theta)$  is going well through the points that were used in estimation of  $\theta$  (*training data*), but does not *generalize* well to new data.

Here is an example of an application of another model to the data,  $f(x, \theta) = \theta_0 \exp(-(x - \theta_1)^2 / 2\theta_2^2)$ . This family of functions contains  $f_{\text{true}}(x) = \exp(-x^2/2)$  for  $\theta_0 = 1$ ,  $\theta_1 = 0$ , and  $\theta_2 = 1$ . Here [non-linear] regression gives an answer close to  $f_{\text{true}}$ :

```
[...]/teaching/2020-1/math_575b/notes/regression$ cat gauss.py
from sys import argv; import numpy as np; from scipy.optimize import minimize
global xy
def loss(theta):
    if (theta[2] <= 0.):
        return 1.e+100
    loss, N = 0., xy.shape[0]
    for i in range(0, N):
        y = theta[0] * np.exp(-(xy[i, 0] - theta[1])**2 / (2. * theta[2]**2))
        loss += (xy[i, 1] - y)**2
    return loss

xy = np.loadtxt(argv[1])
res = minimize(loss, np.array([2., 2., 2.]), method='BFGS', jac = None)
print(res.x)
[...]/teaching/2020-1/math_575b/notes/regression$ python3 gauss.py data_10
[ 1.08778132 -0.06268182  1.01929772]
[...]/teaching/2020-1/math_575b/notes/regression$ python3 gauss.py data_1000
[ 1.00205441 -0.01667168  0.98588332]
[...]/teaching/2020-1/math_575b/notes/regression$
```

To detect the overfitting, we can divide the data we have to two parts: training data and *validation set*. We use the training data to estimate the parameters  $\theta$ . Then we check how large is the loss function being calculated on the validation set. If the loss function [per data point] on the training

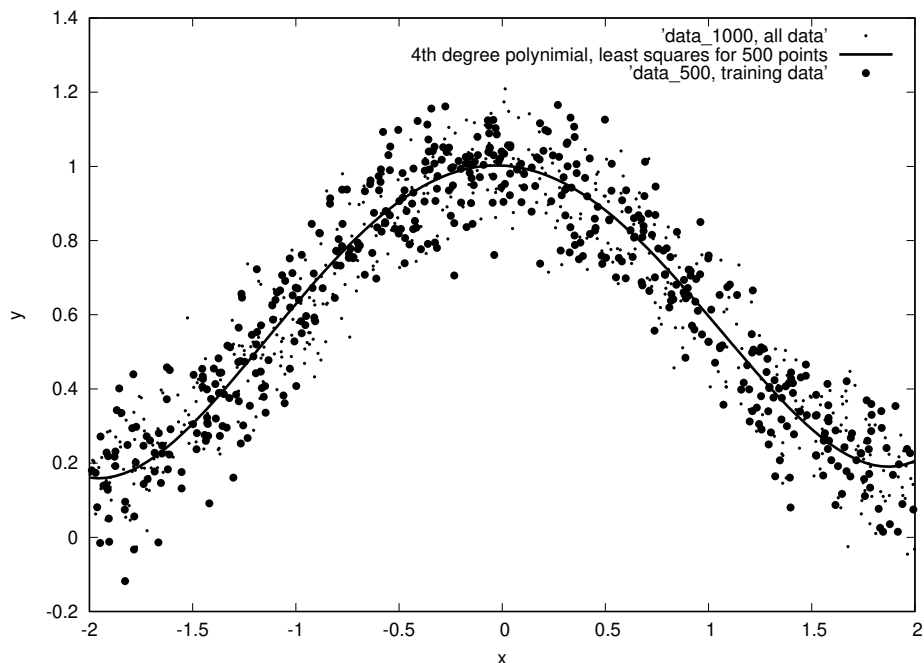
data is much less then on the validation set, then we have overfitting. If they are comparable, then  $f(\mathbf{x}, \boldsymbol{\theta})$  generalizes to new data.

```
[...]/teaching/2020-1/math_575b/notes/regression$ cat poly_4.py
from sys import argv; import numpy as np; from scipy.optimize import minimize
global xy
def loss(theta):
    loss, N, M = 0., xy.shape[0], theta.shape[0]
    for i in range(0, N):
        y = 0.
        for m in range(0, M):
            y += theta[m] * xy[i, 0]**m
        loss += (xy[i, 1] - y)**2
    return loss

xy = np.loadtxt('data_500')
res = minimize(loss, np.zeros(5), method='BFGS', jac = None)
print(res.x)
print(' loss function per data point on training data:', loss(res.x) / 500.)

xy = np.loadtxt('data_500_end')
print('loss function per data point on validation set:', loss(res.x) / 500.)

xy = np.loadtxt('data_1000')
res = minimize(loss, np.zeros(5), method='BFGS', jac = None)
print(res.x)
[...]/teaching/2020-1/math_575b/notes/regression$ python3 poly_4.py
[ 1.00226136 -0.02551919 -0.45139884  0.00918124  0.06167193]
 loss function per data point on training data: 0.010732697473740654
loss function per data point on validation set: 0.010292081361308608
[ 0.9908246  -0.01466693 -0.44185825  0.00426412  0.06000264]
[...]/teaching/2020-1/math_575b/notes/regression$
```



## Problems and exercises

1. Consider data generated by the following Python script:

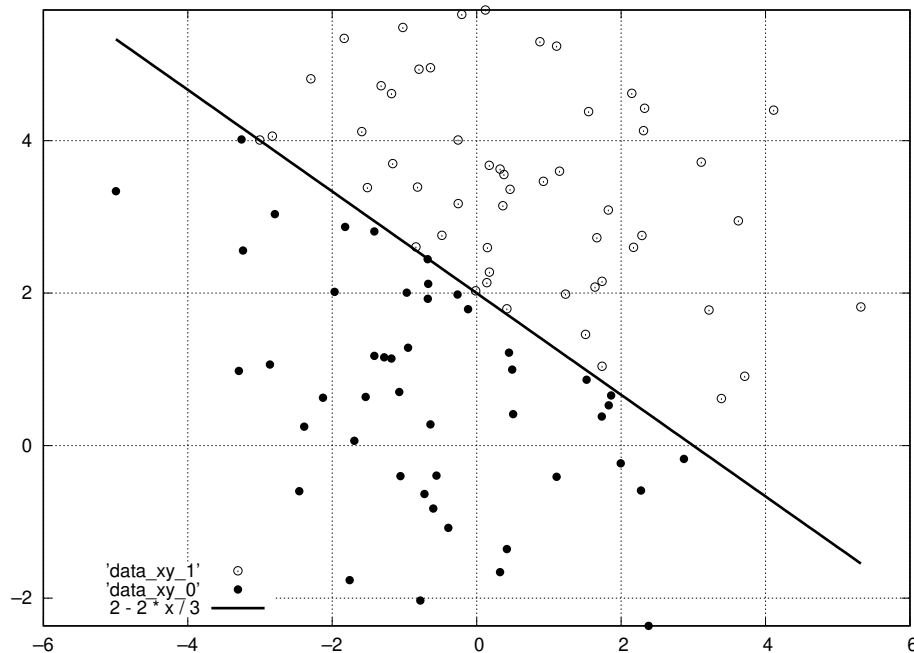
```
from math import sin, pi; from random import seed, random, normalvariate
seed(0)
for i in range(0, 1000):
    x = 2. * pi * random() - pi
    print(x, sin(x) + normalvariate(0., 0.1))
```

We would like to fit the data by the  $M^{\text{th}}$  degree polynomial  $f(x, \boldsymbol{\theta}) = \sum_{m=0}^M \theta_m x^m$ . Let the loss function per data point be  $\frac{1}{\text{number of data points}} \sum_{i \text{ in data}} (y_i - f(x_i, \boldsymbol{\theta}))^2$ . Proceed with [linear] regression, using as the training data first  $N > M$  points of the whole data [of 1000 points], and use the rest as a validation set. Find an appropriate value of  $M$  (the loss function hardly decreases if you increase  $M$ ) and the size of training data that is sufficient to learn optimal  $\boldsymbol{\theta}$  (the loss function per data point on training and validation sets are comparable).

## 28 Classification

Let there be several groups of objects. Each group has a certain label  $l \in L$ , where  $L$  is the set of all possible labels. Each object can be described, *e.g.*, by its numerical features  $\mathbf{x} \in \mathbf{R}^D$ . We would like to be able to find the object's label from its numerical representation — a *classification* problem. This can be viewed as a problem of approximation of a function  $f: \mathbf{R}^D \rightarrow L$ .

**Example 28:** Consider the points on  $(x, y)$ -plane. The points differ in whether they are above (label “1”) or below (label “0”) the line  $x/3 + y/2 = 1$  or  $y = 2 - 2x/3$ :



Here is a Python script that generated this data:

```
from random import normalvariate
def class_true(x, y):
```

```

    return 1. if (x / 3. + y / 2. > 1.) else 0.
for n in range(0, 100):
    x, y = normalvariate(0, 2.), 2. + normalvariate(0, 2.)
    print(x, y, class_true(x, y))

```

On the training data we have the values  $(x, y) \in \mathbf{R}^2$  and also the correct labels.<sup>18</sup> This is called *supervised learning* (a supervisor provided the labels for us to learn how they are assigned).

```

[...]/teaching/2020-1/math_575b/notes/linear_classifier$ cat classifier.py
import numpy as np; from scipy.optimize import minimize
global xy, N

def class_true(x, y):
    return 1. if (x / 3. + y / 2. > 1.) else 0.

def loss(theta):
    loss = 0.
    for i in range(0, xy.shape[0]):
        f_true = class_true(xy[i, 0], xy[i, 1])
        f = theta[0] + theta[1] * xy[i, 0] + theta[2] * xy[i, 1]
        f = 1. if (f > 0.) else 0.
        loss += (f - f_true)**2
    return loss

xy = np.loadtxt('data_xy')
res = minimize(loss, np.zeros(3), method='BFGS', jac = None)
print(res.x, loss(res.x))
[...]/teaching/2020-1/math_575b/notes/linear_classifier$ python3 classifier.py
[1.03849898e-05 3.46166326e-05 3.63474643e-05] 27.0
[...]/teaching/2020-1/math_575b/notes/linear_classifier$ diff --suppress-common-
lines -tyW 156 classifier.py classifier_10.py
res = minimize(loss, np.zeros(3), method='BFGS', jac = None)
res = minimize(loss, np.array([0., 1., 0.]), method='BFGS', jac = None)
[...]/teaching/2020-1/math_575b/notes/linear_classifier$ python3 classifier_10.py
[0. 1. 0.] 33.0
[...]/teaching/2020-1/math_575b/notes/linear_classifier$

```

Here we try the following change  $\theta_0 + \theta_1 x + \theta_2 y \longrightarrow \theta_0 + \cos(\theta_1)x + \sin(\theta_1)y$ :

```

[...]/teaching/2020-1/math_575b/notes/linear_classifier$ diff --suppress-common-
lines -tyW 156 classifier.py class_cos_sin.py
    f = theta[0] + theta[1] * xy[i, 0] + theta[2] * xy[i, 1]
    f = theta[0] + np.cos(theta[1]) * xy[i, 0] + np.sin(theta[1]) * xy[i, 1]
res = minimize(loss, np.zeros(3), method='BFGS', jac = None)
res = minimize(loss, np.zeros(2), method='BFGS', jac = None)
[...]/teaching/2020-1/math_575b/notes/linear_classifier$ python3 class_cos_sin.py
[0. 0.] 33.0
[...]/teaching/2020-1/math_575b/notes/linear_classifier$ diff --suppress-common-
lines -tyW 156 class_cos_sin.py class_cos_sin_12.py
res = minimize(loss, np.zeros(2), method='BFGS', jac = None)
res = minimize(loss, np.array([1., 2.]), method='BFGS', jac = None)
[...]/teaching/2020-1/math_575b/notes/linear_classifier$ python3 class_cos_sin_12.py
[1. 2.] 40.0
[...]/teaching/2020-1/math_575b/notes/linear_classifier$

```

---

<sup>18</sup> In scripts below just  $x$  and  $y$  values are read, but there is `class_true` function which returns the correct label.

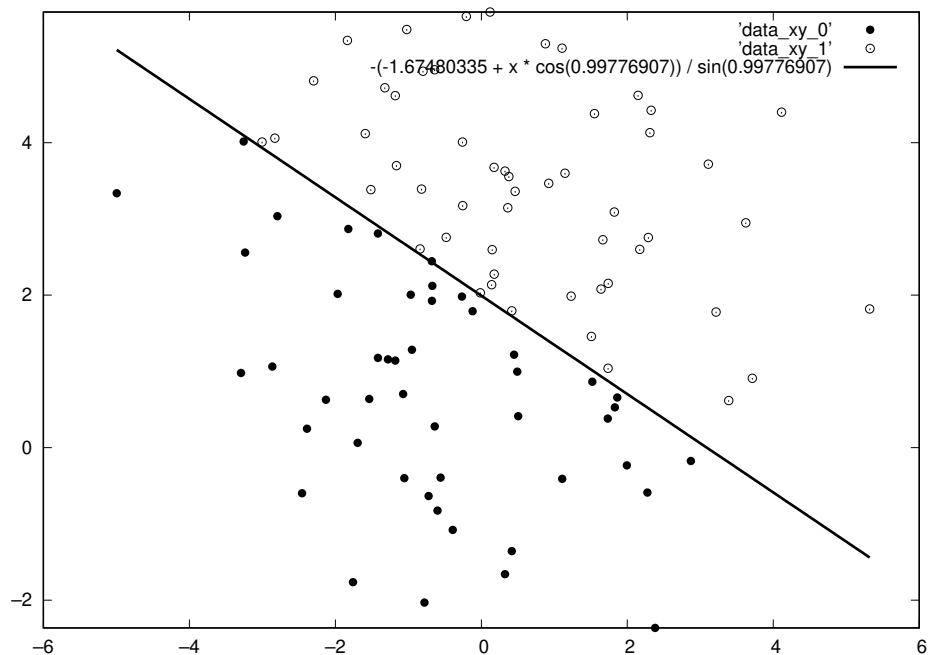
Here we make the function, that approximates  $f_{\text{true}}(x,y) \rightarrow \{0,1\}$ , smooth:  $f(x,y) := 1/(1 + \exp(-10\xi))$ , where  $\xi := \theta_0 + \cos(\theta_1)x + \sin(\theta_1)y$ :

```
[...]/teaching/2020-1/math_575b/notes/linear_classifier$ cat smooth.py
import numpy as np; from scipy.optimize import minimize
global xy, N

def class_true(x, y):
    return 1. if (x / 3. + y / 2. > 1.) else 0.

def loss(theta):
    loss = 0.
    for i in range(0, xy.shape[0]):
        f_true = class_true(xy[i, 0], xy[i, 1])
        f = theta[0] + np.cos(theta[1]) * xy[i, 0] + np.sin(theta[1]) * xy[i, 1]
        f = 1. / (1. + np.exp(-10. * f))
        loss += (f - f_true)**2
    return loss

xy = np.loadtxt('data_xy')
res = minimize(loss, np.zeros(2), method='BFGS', jac = None)
print(res.x, loss(res.x))
[...]/teaching/2020-1/math_575b/notes/linear_classifier$ dif
f --suppress-common-lines -tyW 156 classifier.py smooth.py
    f = theta[0] + theta[1] * xy[i, 0] + theta[2] * xy[i, 1]
    f = theta[0] + np.cos(theta[1]) * xy[i, 0] + np.sin(theta[1]) * xy[i, 1]
    f = 1. if (f > 0.) else 0.
    f = 1. / (1. + np.exp(-10. * f))
res = minimize(loss, np.zeros(3), method='BFGS', jac = None)
res = minimize(loss, np.zeros(2), method='BFGS', jac = None)
[...]/teaching/2020-1/math_575b/notes/linear_classifier$ python3 smooth.py
[-1.67480335  0.99776907] 1.2944486110163806
[...]/teaching/2020-1/math_575b/notes/linear_classifier$
```



## 29 Clustering

Consider there is data/objects  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N \in \mathbf{R}^D$  (or of whatever nature). We want to aggregate/lump objects into not so many groups, with objects in each group being close in some sense to each other. This is a task called *clustering*.

Clustering could be set up as a supervised learning, where clustering labels for some of the objects are provided by the supervisor/teacher/human. Sometimes the supervisor provides the desired final number of clusters. Often the clustering algorithm is to produce the clusters (with their number) on its own — *unsupervised learning*.

The number of groups in the end could be fixed or left to be determined from the data. The division into groups could be strict or in distributional sense (for each object there is a probability/likelihood to belong to each cluster). It could be allowed to claim that some objects do not belong to any [dense of well defined] cluster.

There are numerous strategies to cluster data:

- hierarchical clustering
  - agglomerative: Initially clusters are data points. Two “closest” points/clusters are found and then merged, with clusters closeness measure redefined/updated.
  - divisive: The whole data is divided/cut into two [large] groups, which are further processed.
- setting clustering as an optimization problem, then solving it:

$$\text{\textit{k-means}} : \quad \arg \min_{C_1, C_2, \dots, C_k} \sum_{i=1}^k \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2, \quad \boldsymbol{\mu}_i := \frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x}$$

Such discrete optimization problem is typically hard, so approximation are used:

```
import numpy as np
global xy, mu, label

def assign_labels():
    global mu, label
    for j in range(0, xy.shape[0]):
        best, best_i = 1.e+10, -1
        for i in range(0, k):
            distance = np.linalg.norm(xy[j] - mu[i])
            if (distance < best):
                best, best_i = distance, i
        label[j] = best_i

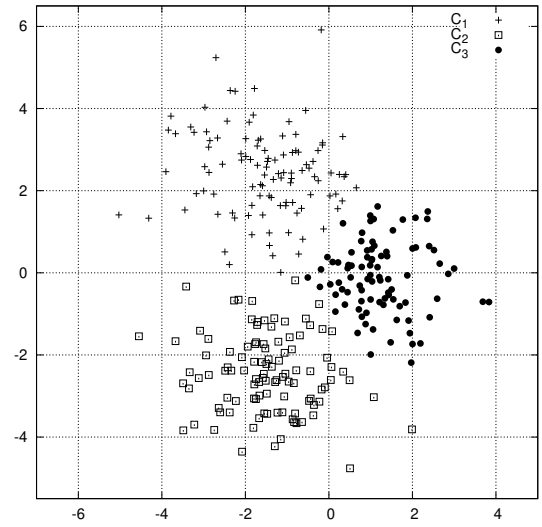
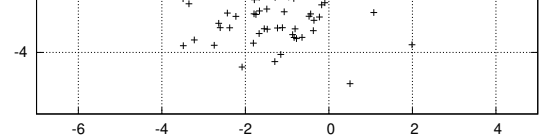
def compute_mu():
    global mu
    for i in range(0, k):
        mu[i], n = np.zeros(2), 0
        for j in range(0, xy.shape[0]):
            if (label[j] == i):
```

```

        mu[i], n = mu[i] + xy[j], n + 1
    mu[i] /= n
    return mu

xy = np.loadtxt('data_300')
k, label = 3, [0] * xy.shape[0]
mu = np.random.normal(0., 1., (k, 2))
assign_labels()
while (1 > 0):
    compute_mu()
    old_label = label
    assign_labels()
    if (old_label == label):
        break
for j in range(0, xy.shape[0]):
    print(xy[j, 0], xy[j, 1], label[j])

```



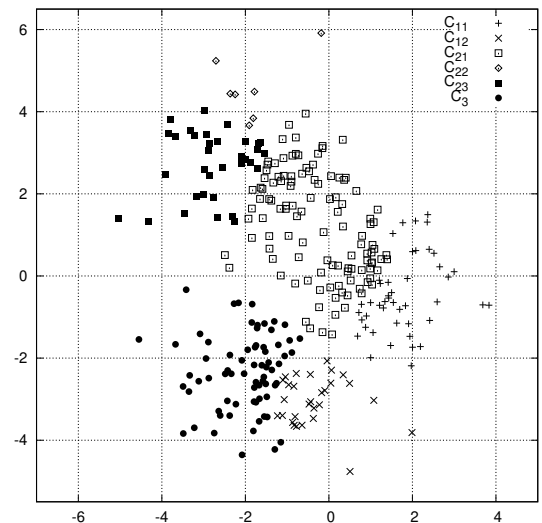
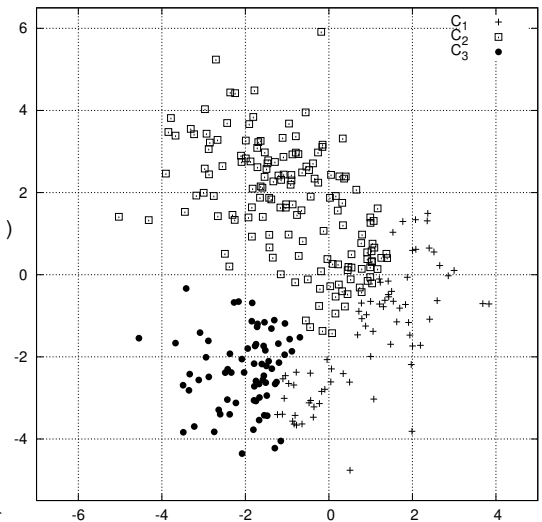
Here is a [not at all optimized] Python script that implements a [hierarchical] agglomerative method of clustering called *complete-linkage* clustering:

```
import numpy as np
global xy, N, label, dist, lab_d

xy = np.loadtxt('data_300')
N = xy.shape[0]
dist, label = np.zeros((N, N)), np.arange(N)
for i in range(0, N - 1):
    for j in range(i + 1, N): # i < j
        dist[i, j] = np.linalg.norm(xy[i] - xy[j])

while (np.unique(label).shape[0] > 3):
    lab_d = np.zeros((N, N))
    for i in range(0, N - 1):
        for j in range(i + 1, N): # i < j
            if (label[i] != label[j]):
                smaller, greater = label[i], label[j]
                if (smaller > greater):
                    smaller, greater = greater, smaller
                if (lab_d[smaller, greater] < dist[i][j]):
                    lab_d[smaller, greater] = dist[i, j]
    best = 1.e+10
    for i in range(0, N - 1):
        for j in range(i + 1, N): # i < j
            if (label[i] != label[j]):
                smaller, greater = label[i], label[j]
                if (smaller > greater):
                    smaller, greater = greater, smaller
                if (lab_d[smaller, greater] < best):
                    best_s, best_g, best = smaller, greater, lab_d[smaller, greater]
    label[np.where(label == best_g)] = best_s
    print(label)

for i in range(0, N):
    print(xy[i, 0], xy[i, 1], label[i])
```



## Problems and exercises

1. Consider data generated by the following Python script:

```

from random import seed, random
seed(0)
n = 0
while (n < 1000):
    x, y = 2. * random() - 1., 2. * random() - 1.
    r2 = x**2 + y**2
    if ((r2 < 1.) and ((r2 < 0.36) or (r2 > 0.64))):
        print(x, y)
        n += 1

```

We want to divide the data points into 2 clusters. Decide which method, [k-means](#), [single-linkage](#), or [complete-linkage](#) clustering is more suitable for task. Proceed with the clustering and plot the 2 resulting clusters.

## References

- [BoVa09] S. Boyd, L. Vandenberghe, *Convex optimization*, 7<sup>th</sup> ed. (Cambridge U. Press, 2009).
- [Cal20] Jeff Calder, *[The Calculus of Variations](#)*.
- [TrBa97] L. N. Trefethen, D. Bau III, *Numerical linear algebra* (SIAM, 1997).