MATH 575A Numerical Analysis
Fall 2019 class notes

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

Part I
Rounding errors and numerical stability

Very often the study of numerical analysis starts with the topic of rounding errors. These class notes is not an exception.[1] Suggested reading: [TrBa97, Lecs. 12–15] and [Hig02, Chs. 1, 2].

1 Floating-point arithmetic

One can come up with 3 ways to represent information: I) analogue, II) digital, and III) algorithmical. Examples of information in analogue form would be: a length of a wooden stick, printed photo, audio tape recording,[2] a mass of an object. Examples of information in digital form are: post address, a text in a book, computer file, *etc.* Algorithmic way of presenting/storing information instead of giving information *per se*, just operates with a recipe how to produce the information. Information in digital form is easier to handle.[3] In computer the information is presented in bits that could be in 2 states: 0 and 1 (or low and high voltage). Having just 2 states makes circuit design easier.[4]

Let us say we want to devote N bits to store a [real] number. We have 2^N possible states, and thus no more than 2^N different numbers to play with. How would we spend these N bits? A possible [linear] way is to have numbers $\langle 1-2^{N-1}, 2-2^{N-1}, \ldots, -1, 0, 1, 2, \ldots, 2^{N-1}-1, 2^{N-1}\rangle/2^E$, where E is not far from $N/2$. This way we have both large (about 2^{N-E}) and small (about 2^{-E}) numbers. It is easy to proceed with addition in this system. Long multiplication (more elaborate than addition) can be used. The problem is that largest and smallest non-zero numbers in this system are not that very large or small.[5]

Logarithmic quantization, with numbers $\pm(1+\varepsilon)^E$, where $\varepsilon \ll 1$ and $E \lesssim 2^N$. This way we could get very large and very small numbers easily, even with small relative change between the

---

[1] Here is an essay that advocates the point of view that numerical analysis is far from being just the study of rounding errors: L. N. Threfethen, *The definition of numerical analysis*, 1992.

[2] Later there were devices which wrote and read audio or video information on a magnetic tape in digital form.

[3] Copying without distortion is possible. Error correction is much more effective.

[4] For any base or radix B one can introduce a so called "radix economy" $E(B) := B/\ln B$. It is roughly the number of B-digits needed to represent a [large] number N multiplied by B (number of different B-digits) and divided by $\ln N$. Ternary system is more radix economic, $E(3) < E(2)$, and there were some ternary computers built.

[5] In many image formats (JPEG, PNG, *etc.*) a typical color depth is $N = 8$ bits. Often in digital photos some areas are solid white due to clipping, while some others are too dark. In Compact Disc (CD) $N = 16$ is used. This seems to be unsatisfactory for some, which gave rise to DVD-Audio (up to $N = 24$) and Super Audio CD (SACD).

consequtive numbers. It is very easy to multiply numberes in it (but addition becomes even harder than long multiplication in linear quantization).

A hybrid approach, combining the two, is to spend some bits (*fraction* or *mantissa*) in linear way, while some others (*exponent*) in logarithmic. One bit is reserved to store the sign (positive or negative) of the number. This is so called floating point numerical system. Eventually it was standardized as IEEE 754 [IEEE85]. Fraction part allows easy addition and [long] multiplication, while dynamic range is huge due to the exponent part.

Consider the following hypothetical floating-point numerical system $\mathbf{FP}_N$: the number in it is either 0, or a floating-point number with the fraction made of N bits:[6]

$$\pm 2^E \cdot \big(1.b_1 b_2 \ldots b_N\big)_2 = \pm 2^E \left(1 + \sum_{n=1}^{N} b_n 2^{-n}\right)$$

The exponent E could be any integer number, so $\mathbf{FP}_N$ has arbitrarily small or large numbers (one can think of 0 having exponent $E = -\infty$). Not considering possibility of overflows/underflows, IEEE 754 single and double precision formats correspond to $\mathbf{FP}_{23}$ and $\mathbf{FP}_{52}$.

There is an obvious mapping $\mathrm{inj} : \mathbf{FP}_N \to \mathbf{R}$. Let us construct the "best numerical representation" mapping $\mathrm{num} : \mathbf{R} \to \mathbf{FP}_N$ by choosing $\mathrm{num}(x)$ to be the closest to x number from $\mathbf{FP}_N$ (in cases where two numbers from $\mathbf{FP}_N$ are at the same minimal possible distance, [to imitate "round half to even" rule] the number with larger E or (if the exponents are the same) with $b_N = 0$ is chosen). Obviously, $\mathrm{num}\big(\mathrm{inj}(x)\big) = x$ for any $x \in \mathbf{FP}_N$. The reverse is only approximately true:

Theorem 1: For any $x \in \mathbf{R}$ we have $\mathrm{inj}\big(\mathrm{num}(x)\big) = x(1+\varepsilon)$, where $|\varepsilon| \le \varepsilon_{\mathrm{machine}} \coloneqq 2^{-(N+1)}$. In other words, it is possible to represent real numbers in floating-point numerical system $\mathbf{FP}_N$ with small (*machine epsilon* or *unit roundoff* $\varepsilon_{\mathrm{machine}}$) relative error.

Proof: We have $\mathrm{num}(2x) = 2\,\mathrm{num}(x)$, so without loss of generality we can assume that $1 \le x < 2$. Within this interval $\mathbf{FP}_N$ contains numbers $1 + k2^{-N}$, where $0 \le k < 2^N$ and k is integer. Then $|\mathrm{num}(x) - x| \le 2^{-(N+1)}$, and $\varepsilon = (\mathrm{num}(x) - x)/x$.[7]

IEEE double precision arithmetic corresponds to $N = 52$, and $\varepsilon_{\mathrm{machine}} = 2^{-53} \approx 1.11 \cdot 10^{-16}$, *i.e.*, it deals with numbers having approximately 16 decimal digits.[8]

Analysis of numerical algorithms is impossible without concrete knowledge of how basic arithmetic operations [or built-in functions like $\exp(\cdot)$ or $\sin(\cdot)$] are implemented. In hypothetical implementation, that we will call *virtual*, the numerical arithmetic operations of addition, multiplication, and division are defined as

$$x \oplus y \coloneqq \mathrm{num}\big(\mathrm{inj}(x) + \mathrm{inj}(y)\big), \quad x \odot y \coloneqq \mathrm{num}\big(\mathrm{inj}(x) \cdot \mathrm{inj}(y)\big), \quad x \oslash y \coloneqq \mathrm{num}\big(\mathrm{inj}(x) / \mathrm{inj}(y)\big)$$

The operation of subtration $\ominus$ is defined similarly or, *e.g.*, as $x \ominus y \coloneqq x \oplus (-y)$, where $(-y)$ is number y with the sign bit being flipped. For built-in functions we would assume $f(x) = \mathrm{num}\big(f\big(\mathrm{inj}(x)\big)\big)$.

This way defined arithmetic operations satisfy the following fundamental property:

$$\text{for all } x, y \in \mathbf{FP}_N \text{ we have } x \circledast y = \big(\mathrm{inj}(x) * \mathrm{inj}(y)\big)(1+\varepsilon), \text{ where } |\varepsilon| \le \varepsilon_{\mathrm{machine}} \tag{1}$$

[6] One can come up with a floating-point numerical system with any *base* or *radix*. An early computer ENIAC was operating with [up to 20 digits] decimal numbers.

[7] For brevity, the mapping "inj" was skipped in several places.

[8] MATLAB® defines "`eps`" as $2\varepsilon_{\mathrm{machine}} \approx 2.22 \cdot 10^{-16}$.

```
[...]/teaching/2019-4/math_575a/notes/C$ cat print_bits.c
#include <stdio.h>
void print_bits(double x, char* s) { int i; unsigned char c;
  for (i = 0; i < 8; i++) for (c = 128; c != 0; c >>= 1)
    if (*((char *)(&x) + 7 - i) & c) printf("1"); else printf("0");
  printf("  %s\n", s); }
int main() { int i; double x, x52;
  print_bits(0., " 0");
  print_bits(1., " 1");
  print_bits(-1., "-1");
  print_bits(0.5, "1/2");
  print_bits(0.25, "1/4");
  print_bits(1. / 3., "1/3");
  print_bits(2., " 2");
  print_bits(4., " 4");
  print_bits(1.5, " 1 + 1/2");
  print_bits(1. + 1. / 1024., " 1 + 2^(-10)");
  x52 = 1.; for (i = 0; i < 52; i++) x52 /= 2.;
  print_bits(1. + x52, " 1 + 2^(-52)");
  print_bits(1. - x52, " 1 - 2^(-52)");
  print_bits(1. + (2. * x52), " 1 + 2^(-51)");
  print_bits(2. + (2. * x52), " 2 + 2^(-51)");
  x = 1.; for (i = 0; i < 1022; i++) x /= 2.; print_bits(x, " 2^(-1022)");
  print_bits(x * x52, " 2^(-1074)");
printf("|\\___  ____/\\_________________________  _______________________/\n");
printf("|    \\/                             \\/\n");
printf("sign  exponent (11 bits)               fraction (52 bits)\n");
  return 0; }
[...]/teaching/2019-4/math_575a/notes/C$ cc print_bits.c
[...]/teaching/2019-4/math_575a/notes/C$ ./a.out
0000000000000000000000000000000000000000000000000000000000000000   0
0011111111110000000000000000000000000000000000000000000000000000   1
1011111111110000000000000000000000000000000000000000000000000000  -1
0011111111100000000000000000000000000000000000000000000000000000  1/2
0011111111010000000000000000000000000000000000000000000000000000  1/4
0011111111010101010101010101010101010101010101010101010101010101  1/3
0100000000000000000000000000000000000000000000000000000000000000   2
0100000000010000000000000000000000000000000000000000000000000000   4
0011111111111000000000000000000000000000000000000000000000000000   1 + 1/2
0011111111110000000001000000000000000000000000000000000000000000   1 + 2^(-10)
0011111111110000000000000000000000000000000000000000000000000001   1 + 2^(-52)
0011111111101111111111111111111111111111111111111111111111111110   1 - 2^(-52)
0011111111110000000000000000000000000000000000000000000000000010   1 + 2^(-51)
0100000000000000000000000000000000000000000000000000000000000001   2 + 2^(-51)
0000000000010000000000000000000000000000000000000000000000000000   2^(-1022)
0000000000000000000000000000000000000000000000000000000000000001   2^(-1074)
|\___  ____/\_________________________  _______________________/
|    \/                                \/
sign  exponent (11 bits)               fraction (52 bits)
[...]/teaching/2019-4/math_575a/notes/C$
```

where $*$ is any of four $+$, $-$, $\cdot$, $/$ operations, with $\oplus$, $\ominus$, $\odot$, $\oslash$ being their floating point analogues. We will call $\varepsilon_{\text{machine}}$-*valid* any floating-point arithmetic implementation, such that the numerical arithmetic operations of addition, subtraction, multiplication, and division do satisfy (1).

Problems and exercises

1. Find the minimal positive integer n in $\mathbf{FP}_N$ such that $n+1$ is already not in $\mathbf{FP}_N$.
2. Prove that if $\text{num}(x) = 0$, then $x = 0$.
3. Prove that if $x \odot y = 0$, then at least one of x and y is equal to 0 ($\mathbf{FP}_N$ has no zero divisors).
4. Obviously, [in virtual implementation] addition $\oplus$ and multiplication $\odot$ operations in $\mathbf{FP}_N$ are commutative. Also 0 and 1 are indeed neutral elements for addition and multiplication, respectively [why?]. Prove of disprove by counterexample that addition $\oplus$ is (a) associative, (b) has opposites; multiplication $\odot$ is (c) associative, (d) has inverses; and (e) distibutive property holds.
5. Plot the polynomial $P(x) = (x-4)^6$ computed as

$$\Big(\Big((((x^6 - 24x^5) + 240x^4) - 1280x^3) + 3840x^2\Big) - 6144x\Big) + 4096$$

for $x = 3.98, 3.98002, 3.98004, \ldots, 4.02$. Explain the quantization of the values of $P(x)$. Also plot $P(x)$ computed as $(x-4)^6$.

2 Stable and unstable numerical algorithms

Definition 2.1: A numerical *algorithm* is a mapping $F : \mathcal{X} \subseteq \mathbf{R}^m \to \mathbf{FP}_N^n$ from m-dimensional input data to n-dimensional output, with the way how the mapping is computed [including how the input is processed and how the output is interpreted] being clearly described. It is an attempt to numerically simulate an ideal mapping $F_{\text{exact}} : \mathcal{X} \to \mathbf{R}^n$ (which is the underlying [mathematical] *problem*, as in [TrBa97, Lec. 12]).

Example 2.1: Consider, *e.g.*, the problem of finding the point on a circle $x^2 + y^2 = 1$ that is the closest to a given point (x_0, y_0). A possible algorithm for solving it would be a mapping $(x_0, y_0) \in \mathbf{R}^2 \backslash (0,0) \mapsto \big(x_0/\sqrt{x_0^2 + y_0^2},\, y_0/\sqrt{x_0^2 + y_0^2}\big)$, here $m = n = 2$. Another algorithm would be a mapping $(x_0, y_0) \mapsto \varphi \coloneqq \texttt{atan2}(y_0, x_0)$, with $n = 1$, and the output is interpreted as the point $(\cos\varphi, \sin\varphi)$.[9]

Let us say, we want to calculate $F(\boldsymbol{x})$, with $\boldsymbol{x} \in \mathbf{R}^m$ being the input. Obviously, the components of $\boldsymbol{x}$ are not necessarily exactly representable in $\mathbf{FP}_N$. Inevitably one can expect a relative error about $\varepsilon_{\text{machine}}$ just from entering $\boldsymbol{x}$ into a computer. Instead of calculating $F(\boldsymbol{x})$, we are computing $F(\boldsymbol{x} + \Delta\boldsymbol{x})$, where $\Delta\boldsymbol{x}$, which is called *backward error*, could be just a round off error in $\boldsymbol{x}$.[10]

Definition 2.2: An algorithm F is called *backward stable*, if for any input $\boldsymbol{x}$ we have $F(\boldsymbol{x}) = F_{\text{exact}}(\boldsymbol{x} + \boldsymbol{\eta})$ with $\|\boldsymbol{\eta}\| \sim \varepsilon_{\text{machine}}\|\boldsymbol{x}\|$.[11] In other words, the output $F(\boldsymbol{x})$ is the *exact* answer to the problem with input relatively very close to $\boldsymbol{x}$.[12]

An algorithm being backward stable is the best one can hope for, as backward error is unavoidable. In Example 2.1 the `atan2` version is backward stable, while $m = n = 2$ version is not, as the

[9] `atan2` in C — arc tangent function of two variables.

[10] Sometimes the input is known only up to some uncertainty. In many cases, *e.g.*, weather prediction, people compute $F(\boldsymbol{x} + \Delta\boldsymbol{x})$, with several $\Delta\boldsymbol{x}$ of the order of the uncertainty, to estimate the resulting uncertainty in $F(\boldsymbol{x})$.

[11] Here $\|\cdot\|$ is a magnitude measured somehow. See definition of norm in MATH 527 (theory) course, and also Sec 4.1.

[12] It is assumed that all floating point arithmetical operations inside the algorithm F are done in [some] $\varepsilon_{\text{machine}}$-valid implementation. One can introduce a notion of *virtually backward stable* algorithm, if virtual implementation is used.

resulted point may be not exactly on the unit circle (whenever the exact answer lies in a lower than n dimensional submanifold, it is almost hopeless for the numerical answer to be in it and for the algorithm to be backward stable).

Definition 2.3: An algorithm F is called *stable*, if for any [reasonable of feasible] input $\boldsymbol{x}$ we have $F(\boldsymbol{x}) = F_{\text{exact}}(\boldsymbol{x}+\boldsymbol{\eta})+\boldsymbol{\varepsilon}$ with $\|\boldsymbol{\eta}\| \sim \varepsilon_{\text{machine}}\|\boldsymbol{x}\|$ and $\|\boldsymbol{\varepsilon}\| \sim \varepsilon_{\text{machine}}\|F(\boldsymbol{x})\|$. In other words, the output $F(\boldsymbol{x})$ is relatively close answer to the exact one for the problem with input very close to $\boldsymbol{x}$.

Any backward stable algorithm is stable, but not *vice versa*. Both algorithms in Example 2.1 are stable. Both definitions of backward stable and stable algorithms are about asymptotic behavior of errors in the limit $\varepsilon_{\text{machine}} \to 0$. Namely, for sufficiently small $\varepsilon_{\text{machine}}$, the relative errors are bounded by some constant multiplied by $\varepsilon_{\text{machine}}$.[13]

Example 2.2: computing $\exp(x)$ using Taylor series

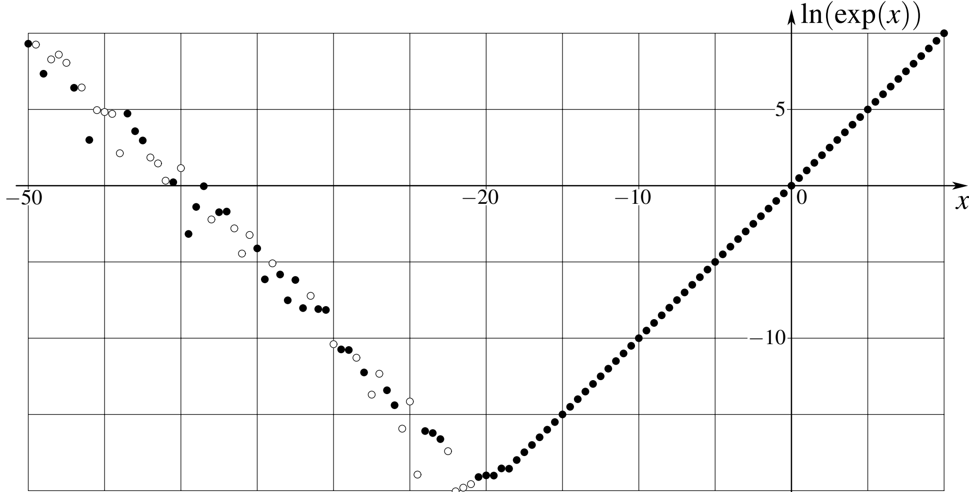
A typical scenario in which numerical accuracy is lost is when a small number is obtained as a sum/difference of almost opposite/same numbers.

Algorithm $\exp(x)\wedge$: Input: $x \in \mathbf{R}$. Output: $\exp(x)$, computed by summing $\sum_{n=0}^{\infty} x^n/n!$ from left to right, stopping when adding a term does not change the partial sum.

```
[...]/teaching/2019-4/math_575a/notes/Python$ cat exp.py
from math import exp
for x in [-50., -40., -30., -20., -15., -10., -2., -1., 0., 1., 10., 30.]:
    taylor, sum, n, xnf = 0., 1., 1, x
    while taylor != sum:
        taylor, sum, n, xnf = sum, sum + xnf, n + 1, x * xnf / (n + 1.)
    print('exp({0: 3.0f}.) = {1: .15e} = {2:.15e}'.format(x, taylor, exp(x)))
[...]/teaching/2019-4/math_575a/notes/Python$ python exp.py
exp(-50.) =  1.107293338289197e+04 = 1.928749847963918e-22
exp(-40.) = -3.165731894063124e+00 = 4.248354255291589e-18
exp(-30.) = -3.066812356356220e-05 = 9.357622968840175e-14
exp(-20.) =  5.621884472130418e-09 = 2.061153622438558e-09
exp(-15.) =  3.059094197302007e-07 = 3.059023205018258e-07
exp(-10.) =  4.539992962303128e-05 = 4.539992976248485e-05
exp( -2.) =  1.353352832366128e-01 = 1.353352832366127e-01
exp( -1.) =  3.678794411714424e-01 = 3.678794411714423e-01
exp(  0.) =  1.000000000000000e+00 = 1.000000000000000e+00
exp(  1.) =  2.718281828459046e+00 = 2.718281828459045e+00
exp( 10.) =  2.202646579480671e+04 = 2.202646579480672e+04
exp( 30.) =  1.068647458152447e+13 = 1.068647458152446e+13
[...]/teaching/2019-4/math_575a/notes/Python$
```

The series $\sum_{n=0}^{\infty} x^n/n!$ converges for any x. If the operations with numbers are done exactly, then the algorithm (although never finishing, as the whole series needs to be went through) would output an exact answer. When x is large and negative, some terms in the series are large, although the answer is small. The small answer $\exp(-|x|)$ is produced as an almost cancellation of as large as $\exp(|x|)$ numbers. Whenever $|x|$ is so large, that $\exp(-|x|)$ is of the order of $2^{-N}\exp(|x|)$, one should not trust any digits of an outputted answer. For IEEE double precision ($N = 52$) this happens for $x \sim -\ln(2^{52})/2 \approx -18$. The algorithm is heavily unstable when applied to large negative x.

[13] For the algorithm to be practically useful, it doesn't necessarily have to be stable. Let us call an algorithm ζ-*semistable* with $0 < \zeta \leq 1$, if $\lim_{\varepsilon_{\text{machine}} \to 0^+} \ln(\text{relative errors})/\ln(\varepsilon_{\text{machine}}) = \zeta$. The Algorithm $\zeta(3)\wedge$ from Example 2.3 is $(2/3)$-semistable, while Algorithm $\pi\cap$ from Example 2.4 and possible algorithm from Example 3.2 are $(1/2)$-semistable. The Algorithm $\exp(1)\wedge$ from Example 2.2 that computes $\mathrm{e} \approx 2.72$ is 1-semistable, but [strictly speaking] is not stable.



Even when applied for $x > 0$ only (*i.e.*, when there are no sign changes in the term of the series), the algorithm is not [strictly speaking] backward stable. The larger is N, the larger is the number of terms that one needs to sum (although here it slowly grows with N, as $n!$ grows very fast). When the partial sum is already not much smaller than the answer, each addition potentially introduces a relative error of the order of $\varepsilon_{\text{machine}}$.

In practice, for large positive x, one would expect [why?] the algorithm to produce $\exp(x) \cdot \big(1 + O(x^{1/2}\varepsilon_{\text{machine}})\big)$, or maybe even $\exp(x) \cdot \big(1 + O(x^{1/4}\varepsilon_{\text{machine}})\big)$.

Example 2.3: computing $\zeta(3)$ by summing the series

Whenever one has a bunch of numbers to sum, it is more safe to add small numbers together first, and add large numbers later.

Let us consider two algorithms for computing

$$\text{Apéry's}^{14} \text{ constant } \zeta(3) := \sum_{n=1}^{\infty} \frac{1}{n^3} = 1.202056903159594285...$$

Algorithm $\zeta(3)\wedge$: Input: none. Output: $\zeta(3)$, computed by summing $\sum_{n=1}^{\infty} 1/n^3$ from left to right, stopping when adding a term does not change the partial sum.

Algorithm $\zeta(3)\vee$: Input: none. Output: $\zeta(3)$, computed by summing $\sum_{n=1}^{\infty} 1/n^3$ from right to left, starting at some sufficiently large n.

```
[...]/teaching/2019-4/math_575a/notes/Python$ cat zeta3.py
zeta3, sum, n = 1., 0., 1
while zeta3 != sum:
    zeta3, sum, n = sum, sum + 1. / float(n**3), n + 1
print(' forward summation: zeta(3) = {0:.15f}'.format(zeta3))
zeta3, n = 0., 25000000
while n > 0:
    zeta3, n = zeta3 + 1. / float(n**3), n - 1
print('backward summation: zeta(3) = {0:.15f}'.format(zeta3))
[...]/teaching/2019-4/math_575a/notes/Python$ python3 zeta3.py
 forward summation: zeta(3) = 1.202056903150321
backward summation: zeta(3) = 1.202056903159594
[...]/teaching/2019-4/math_575a/notes/Python$
```

[14] R. Apéry, *Irrationalité de* $\zeta(2)$ *et* $\zeta(3)$, Astérisque **61**, 11–13 (1979). [Number $\zeta(3)$ is proved to be irrational.]

As it is not possible to represent the number $\zeta(3)$ exactly in $\mathbf{FP}_N$ [for any finite N], *any* numerical algorithm for computing $\zeta(3)$ *can not* be backward stable. The answer in Algorithm $\zeta(3)\wedge$ ignores the tail of the series starting at about $n_* \sim 1/\varepsilon_{\text{machine}}^{1/3}$, and the value of the ignored tail is about $1/n_*^2 \sim \varepsilon_{\text{machine}}^{2/3}$ — about $2/3$ of the significant digits should be correct.

Example 2.4: computing π by maximizing $\sin(\cdot)$

Another common source of accuracy loss is determining a quantity from a function which is not very sensitive to it, *e.g.*, finding position of an extremum of a function.

Algorithm $\pi\cap$: Input: none. Output: π, determined as 2 multiplied by the position of the first maximum of $\sin(x)$. The latter computed as the largest x such that $\sin(x)$ still grows locally:

```
[...]/teaching/2019-4/math_575a/notes/C$ cat find_pi.c
#include <stdio.h>
#include <math.h>
int main() { double pi2, step;
  for (pi2 = 0., step = 1.; pi2 + step != pi2; )
    if ((sin(pi2 + 2. * step) > sin(pi2 + step))
     && (sin(pi2 + step) > sin(pi2)))
      pi2 = pi2 + step; else step *= 0.5;
  printf("computed pi = %22.16e\n", 2. * pi2);
  printf("         pi = %22.16e\n", M_PI);
  return 0; }
[...]/teaching/2019-4/math_575a/notes/C$ cc find_pi.c -lm
[...]/teaching/2019-4/math_575a/notes/C$ ./a.out
computed pi = 3.1415926218032837e+00
         pi = 3.1415926535897931e+00
[...]/teaching/2019-4/math_575a/notes/C$
```

Here π is found from the maximization of $\sin(x)$ near $x = \pi/2$. We have $\sin(x) \approx 1 - (x - \pi/2)^2/2$ there. Whenever $(x - \pi/2)^2 \sim 2^{-N}$, the difference of $\sin(x)$ from 1 is too small for $\mathbf{FP}_N$ to handle. At the distance about $2^{-N/2}$ from $\pi/2$ the numerical function $\sin(x)$ stops to change, which causes only about half of the digits in computed value of π to be correct.

```
[...]/teaching/2019-4/math_575a/notes/C$ cat sin_near_1.c
#include <stdio.h>
#include <math.h>
double x53;
double f(double x) { return (x53 * (1. - sin(x))); }
int main() { int i;
  for (x53 = 1., i = 0; i < 53; i++) x53 *= 2.;
  printf("f(x) = 2^(53) * (1 - sin(x))\n");
  printf("f(pi / 2 - 2.e-8)   = %6.4f\n", f(M_PI / 2. - 2.e-8));
  printf("f(pi / 2 - 1.8e-8)  = %6.4f\n", f(M_PI / 2. - 1.8e-8));
  printf("f(pi / 2 - 1.1e-8)  = %6.4f\n", f(M_PI / 2. - 1.1e-8));
  printf("f(pi / 2 - 1.05e-8) = %6.4f\n", f(M_PI / 2. - 1.05e-8));
  printf("f(pi / 2 - 1.e-8)   = %6.4f\n", f(M_PI / 2. - 1.e-8));
  printf("f(pi / 2)           = %6.4f\n", f(M_PI / 2.));
  printf("f(pi / 2 + 1.e-8)   = %6.4f\n", f(M_PI / 2. + 1.e-8));
  return 0; }
[...]/teaching/2019-4/math_575a/notes/C$ cc sin_near_1.c -lm
[...]/teaching/2019-4/math_575a/notes/C$ ./a.out
f(x) = 2^(53) * (1 - sin(x))
f(pi / 2 - 2.e-8)   = 2.0000
```

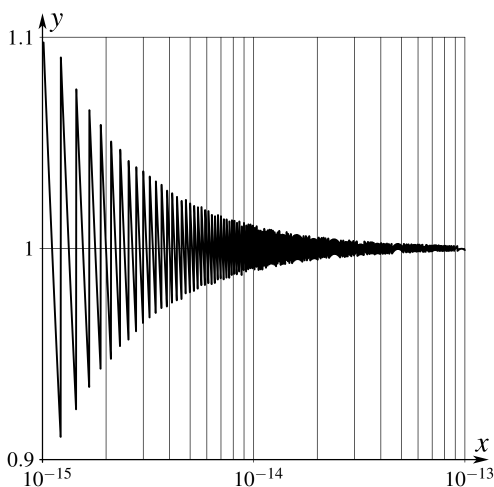
```
f(pi / 2 - 1.8e-8)  = 1.0000
f(pi / 2 - 1.1e-8)  = 1.0000
f(pi / 2 - 1.05e-8) = 0.0000
f(pi / 2 - 1.e-8)   = 0.0000
f(pi / 2)           = 0.0000
f(pi / 2 + 1.e-8)   = 0.0000
[...]/teaching/2019-4/math_575a/notes/C$
```

IEEE double precision, similar to $\mathbf{FP}_{52}$, contains numbers $1-k\cdot 2^{-53}$ and $1+k\cdot 2^{-52}$ with [not too large] non-negative k necessarily being integer. For up to $|x-\pi/2|<10^{-8}$ this integer number, in order to represent the value of $\sin(x)$ better, is equal to 0.

Problems and exercises

1. Consider the following algorithm: Input: two vectors $\boldsymbol{x},\boldsymbol{y}\in\mathbf{R}^n$. Output: geometric angle between the two vectors, computed as $\arccos\big((\boldsymbol{x}\cdot\boldsymbol{y})/\|\boldsymbol{x}\|\,\|\boldsymbol{y}\|\big)$. The norm $\|\boldsymbol{x}\|$ is calculated as $\sqrt{\boldsymbol{x}\cdot\boldsymbol{x}}$. It this algorithm backward stable, stable but not backward stable, or unstable?

2. The graph on the right is the result of numerical calculation of $(\exp(x)-1)/x$ (being computed as it is written). On the other hand, if the function would be computed as $(\exp(x)-1)/\ln(\exp(x))$, then the result would be very close to 1 (as it should be). Explain the difference between the two cases.[15] [16]



3 Condition number

Consider a numerical algorithm $F:\mathcal{X}\subseteq\mathbf{R}^m\to\mathbf{FP}^n_N$. As there are inevitable rounding off errors in processing the input (and in internal calculations), it is important to realize how sensitive F is to small perturbations of the input. A useful quantity to measure that is [relative] *condition number*, which is defined as

$$\kappa(F,\boldsymbol{x}) := \underbrace{\frac{\|F(\boldsymbol{x}+\Delta\boldsymbol{x})-F_{\text{exact}}(\boldsymbol{x})\|}{\|F_{\text{exact}}(\boldsymbol{x})\|}}_{\text{relative change of output due to perturbation }\Delta\boldsymbol{x}\text{ and numerical errors}} \Big/ \overbrace{\frac{\|\Delta\boldsymbol{x}\|}{\|\boldsymbol{x}\|}}^{\text{of the order of }\varepsilon_{\text{machine}}}, \qquad \kappa(F) := \max_{\boldsymbol{x}\in\mathcal{X}}\kappa(F,\boldsymbol{x})$$

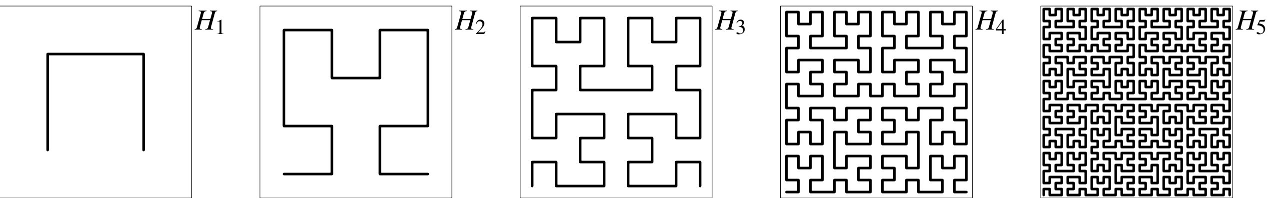
This definition is loose, as how the magnitude $\|\cdot\|$ of relative changes is measured is not specified. If all computations inside the algorithm are assumed to be exact, the condition number is the property of the mapping F_{exact} ifself (or of underlying the [mathematical] *problem*, as in [TrBa97, Lec. 12]). In the limit $\varepsilon_{\text{machine}}\to 0$ and in the case of that mapping being differentiable, the condition number is determined by Jacobian of F_{exact} mapping: $\kappa(F_{\text{exact}},\boldsymbol{x}) = \|J(F_{\text{exact}})\big|_{\text{at }\boldsymbol{x}}\|\,\|\boldsymbol{x}\|/\|F_{\text{exact}}(\boldsymbol{x})\|$.

Example 3.1: Consider a function that is a^{th} power a number: $\cdot^a : x\mapsto x^a$, with $m=n=1$ (here $x>0$ and a is fixed). In the limit $\varepsilon_{\text{machine}}\to 0$, we have $\kappa(\cdot^a) = \big((x^a)'/x^a\big)/(1/x) = a$.

[Pathological] **Example 3.2**: Consider the mapping $H:[0,1]\to[0,1]^2$ which is the Hilbert curve.

[15] You can assume that both $\exp(\cdot)$ and $\ln(\cdot)$ are implemented as "numerical f"$(x) = \operatorname{num}\big(f(\operatorname{inj}(x))\big)$.

[16] Look up `expm1` in Python — e^x-1 to full precision, even for small x.



We have $H = \lim_{n\to\infty} H_n$. One has $\kappa(H) \sim \varepsilon_{\text{machine}}^{-1/2}$. Even if implemented in the best possible way, the algorithm would produce only half of the answer's signigicant digits right.

Example 3.3: Consider the algorithm $\ominus : (x,y) \in \mathbf{R}^2 \mapsto \big(\text{num}(x) \ominus \text{num}(y)\big) \in \mathbf{FP}_N$, with $m = 2$ and $n = 1$, which calculates the difference between the numbers x and y. We have

$$\kappa(\ominus, x, y) = \frac{\varepsilon_{\text{machine}}|x| + \varepsilon_{\text{machine}}|y|}{|x-y|} \cdot \frac{1}{\varepsilon_{\text{machine}}} = \frac{|x|+|y|}{|x-y|}$$

The condition number is large when x and y are relatively close, *i.e.*, when the result $x - y$ is much smaller than x or y. Addition/subtraction of large numbers resulting in small number could greatly deteriorate the relative accuracy. (See, *e.g.*, Example 2.2.)

Example 3.4: In "Lucky Numbers" part of "Surely, You are joking, Mr. Feynman" an "impossible" task to calculate $\tan(10^{100})$ [in one minute with 10% accuracy, no computer,] is posed. On a computer, if you use standard single or double accuracy floating point numerical system, you have only 8 or 16 signigicant [decimal] digits, while in order to find out where 10^{100} is inside the period of $\tan(\cdot)$, you need to know π with no less than 100 digits.

```
[...]/teaching/2019-4/math_575a/notes/C$ cat size_of.c
#include <stdio.h>
int main() { printf("sizeof(   float   ) = %2lu\n", sizeof(float));
  printf("sizeof(   double  ) = %2lu\n", sizeof(double));
  printf("sizeof(long double) = %2lu\n", sizeof(long double)); return 0; }
[...]/teaching/2019-4/math_575a/notes/C$ cc size_of.c
[...]/teaching/2019-4/math_575a/notes/C$ ./a.out
sizeof(   float   ) =  4
sizeof(   double  ) =  8
sizeof(long double) = 16
[...]/teaching/2019-4/math_575a/notes/C$
```

The C type `long double` is non-standard, and its size could be anything starting at 8 bytes. In 90s personal computers its size was typically 8 bytes. Later it was sometimes 12 bytes. FORTRAN has `real*16` and `complex*32` types, but [if you have concerns about the speed] you may check whether they are natively supported. Even with 16 bytes you get about $30 < 100$ digits. There is a lot of software to work with arbitrary- or multiple-precision, allowing very large integers and floats having plenty of significant digits.[17] Here is $\tan(10^{100})$ being calculated by Wolfram|Alpha and GP/PARI:

```
[...]/teaching/2019-4/math_575a/notes/gp-pari$ gp
[... technical stuff ...]
                  GP/PARI CALCULATOR Version 2.11.1 (released)
[... copyright notice and links ...]
parisize = 8000000, primelimit = 500000, nbthreads = 4
? tan(10^100)
```

[17] GNU MP, UBASIC, mpmath, and many, many more.

```
%1 = 0.40123196199081435418575434365329495832
? tan(10.^100.)
  ***   at top-level: tan(10.^100.)
  ***                 ^-------------
  *** tan: precision too low in mpcosm1.
  ***   Break loop: type 'break' to go back to GP prompt
break> break

? \p 120
   realprecision = 134 significant digits (120 digits displayed)
? tan(10^100)
%2 = 0.401231961990814354185754343653294958323870261129244068319441538116871809822
119121146726730974932083113492712621181822475
? tan(10.^100.)
%3 = 0.401231961990814354185754343653294315479678760973444893804899408956723411864
441328951791123274926250333504377945214070810
? \p 220
   realprecision = 231 significant digits (220 digits displayed)
? tan(10^100)
%4 = 0.401231961990814354185754343653294958323870261129244068319441538116871809822
119121146726730974932083113492712621181822474683781490917255223862435549174655457
227844401117202350955319498957782439757435921759611849862972786 3
? tan(10.^100.)
%5 = 0.401231961990814354185754343653294958323870261129244068319441538116871809822
119121146726730974932083113492712621181822474683781492016409266871958335124547226
60345007515059914210285253295208919782382450050131666368417531 25
? quit
Goodbye!
[...]/teaching/2019-4/math_575a/notes/gp-pari$
```

The expressions `tan(10^100)` and `tan(10.^100.)` are parsed differently by GP/PARI, and from the output for the latter one can deduce that about 100 last digits are wrong, consistently with $\kappa(\tan(x)) = 2x/\sin(2x) \sim |x|$ and $x = 10^{100}$.

Problems and exercises

1. Write a program that calculates the Hilbert curve $H(x)$, $0 \leq x \leq 1$ (Example 3.2). Compute $H(1/3)$ and $H(1/\sqrt{2})$ using single and double precision. How many digits are correct?

2. Consider an algorithm $f_h' : \mathbf{R} \to \mathbf{FP}_N$ that estimates the derivative of the function f (which we can compute at any point in stable way) at x as the finite difference $\big(f(x \oplus h) \ominus f(x)\big) \oslash h$. Assuming that $f(x)$ near the point of interest doesn't have any dramatic features (*e.g.*, f, f', f'', ... are of the order of f, f/L, f/L^2, ..., where L is characteristic scale of x) find how $\kappa(f_h')$ depends on h and $\varepsilon_{\text{machine}}$. Which h would you choose? Estimate $f'(1)$ for $f(x) = 1/(1+x^2)$ using $h = 10^{-n}$, $n = 0, 1, 2, ..., 17$.

Part II
Numerical Linear Algebra

4 Matrices, singular value decomposition (SVD)

4.1 Matrices, vectors, orthogonality, norms

An $m \times n$ *matrix* is a rectangular table of numbers/matrix elements, with m rows and n columns. The matrix could originate from writing down in a neat way the coefficients of the system of m linear equations with n variables; or correspond to a linear transformation $\mathbf{C}^n \to \mathbf{C}^m$.

An $m \times 1$ matrix with just one column is an m-dimensional *vector*. An $m \times n$ matrix could be viewed as an [ordered] collection of n such vectors/columns. A 1×1 matrix $[a]$ could be identified with its only matrix element a as a number.

A matrix element of the matrix $\hat{A}$ ($\hat{B}$, $\hat{\Gamma}$, *etc.*) at i^{th} row and j^{th} column will be denoted as $(\hat{A})_{ij} = a_{ij}$ (b_{ij}, γ_{ij}, ...). We will call a [not necessarily square] $m \times n$ matrix $\hat{A}$ *diagonal*, if $(\hat{A})_{ij} = 0$ whenever $i \neq j$. A diagonal matrix is fully determined by its diagonal matrix elements, *e.g.*, $\hat{A} = \operatorname{diag}(a_{11}, a_{22}, ..., a_{ll})$, where $l = \min(m,n)$. We will denote $m \times n$ *zero matrix* (all matrix elements are zero) as $\hat{O}_{m,n}$, while $\hat{I}_n = \operatorname{diag}(1,1,...,1)$ will stand for $n \times n$ *identity matrix*, with $(\hat{I}_n)_{ij} = \delta_{ij}$.

A *dot product* or *scalar product* (making $\mathbf{C}^m$ an inner product space) of two vectors $\boldsymbol{x} = [x_i]$ and $\boldsymbol{y} = [y_i]$ is defined as $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \boldsymbol{x} \cdot \boldsymbol{y} \coloneqq \sum_{i=1}^m x_i^* y_i$, where $\cdot^*$ stands for complex conjugation. The latter is needed so that $\boldsymbol{x} \cdot \boldsymbol{x} = \sum_i |x_i|^2$ is always a non-negative real. A number $\|\boldsymbol{x}\| \coloneqq \sqrt{\boldsymbol{x} \cdot \boldsymbol{x}}$ is called a [L^2-norm or] *length* of vector $\boldsymbol{x}$. *Unit vectors* are vectors of length 1. We say two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are *othogonal*, if $\boldsymbol{x} \cdot \boldsymbol{y} = 0$. Zero vector $\mathbf{0}$ is orthogonal to any vector.

For $\hat{A} : \mathbf{C}^n \to \mathbf{C}^m$, the *adjoint* $\hat{A}^\dagger$ is introduced through $\langle \hat{A}^\dagger \boldsymbol{x}, \boldsymbol{y} \rangle = \langle \boldsymbol{x}, \hat{A}\boldsymbol{y} \rangle = \sum_{i=1}^m x_i^* \sum_{j=1}^n a_{ij} y_j = \sum_{j=1}^n \sum_{i=1}^m a_{ij} x_i^* y_j = \sum_{j=1}^n \left(\sum_{i=1}^m a_{ij}^* x_i \right)^* y_j$, which gives $(\hat{A}^\dagger)_{ji} = a_{ij}^*$, or $n \times m$ matrix $\hat{A}^\dagger = (\hat{A}^{\mathrm{T}})^*$ is the complex conjugate of the transpose of $\hat{A}$, or *Hermitian conjugate* of $\hat{A}$. A [necessarily square] matrix $\hat{A}$ is called *Hermitian*, if $\hat{A}^\dagger = \hat{A}$; it is the complex analogue of real symmetric matrix. The dot product $\boldsymbol{x} \cdot \boldsymbol{y}$ is equal to $\boldsymbol{x}^\dagger \boldsymbol{y}$ as a 1×1 matrix, that is the product of $1 \times m$ matrix $\boldsymbol{x}^\dagger$ and $m \times 1$ matrix $\boldsymbol{y}$.

We will call an $m \times n$ matrix $\hat{A}$ *unitary* or *orthogonal*, if $\hat{A}^\dagger \hat{A} = \hat{I}_n$. This is not a standard terminology.[18] [19] Necessarily, we have $m \geq n$, as $\operatorname{rank} \hat{A} \leq \min(m,n)$. The statement $\hat{A}^\dagger \hat{A} = \hat{I}_n$ simply means that all columns of $\hat{A}$ are unit vectors (the diagonal of $\hat{I}_n$) that are pair-wise orthogonal (off-diagonal content of $\hat{I}_n$).

In some cases other than "L^2" norms are more convenient. Commonly used vector norms are

$$\begin{aligned}
L^p\text{-norm}: &\quad \|\boldsymbol{x}\|_p \coloneqq \left(|x_1|^p + |x_2|^p + ... + |x_m|^p\right)^{1/p}, \qquad 1 \leq p < +\infty \\
L^\infty\text{-norm}: &\quad \|\boldsymbol{x}\|_\infty \coloneqq \max\left(|x_1|, |x_2|, ...|x_m|\right) = \lim_{p \to +\infty} \|\boldsymbol{x}\|_p \\
L^0\text{-"not a norm"}: &\quad \|\boldsymbol{x}\|_0 \coloneqq \left(\text{number of non-zero components of } \boldsymbol{x}\right) \\
\text{weighted norm}: &\quad \|\boldsymbol{x}\|_{\hat{W}} \coloneqq \|\hat{W}\boldsymbol{x}\|, \qquad \operatorname{rank} \hat{W} = m, \text{ arbitrary norm on the right}
\end{aligned}$$

[18] In standard definition, a matrix $\hat{U}$ is *unitary* if it is square, invertible, and $\hat{U}^{-1} = \hat{U}^\dagger$.

[19] People sometimes consider complex matrices $\hat{Q}$ such that $\hat{Q}^{-1} = \hat{Q}^{\mathrm{T}}$, and call them *orthogonal*. All such matrices form a Lie group [why?]. Real orthogonal matrices are unitary.

For matrices / linear transformations / operators most useful are *induced* or *operator* norms:

$$\hat{A} : \mathcal{X} \to \mathcal{Y}, \qquad \|\hat{A}\| \coloneqq \sup_{\boldsymbol{x}\in\mathcal{X}, \boldsymbol{x}\neq\boldsymbol{0}} \frac{\|\hat{A}\boldsymbol{x}\|_{\text{in } \mathcal{Y}}}{\|\boldsymbol{x}\|_{\text{in } \mathcal{X}}} = \sup_{\boldsymbol{x}\in\mathcal{X}, \|\boldsymbol{x}\|_{\text{in } \mathcal{X}}=1} \|\hat{A}\boldsymbol{x}\|_{\text{in } \mathcal{Y}}$$

When both $\mathcal{X}$ and $\mathcal{Y}$ are finite dimensional, the operator norm is finite (the transformation is continuos and the “sphere” $\|\boldsymbol{x}\|_{\text{in } \mathcal{X}} = 1$ is compact).

Plenty of other norms in the vector space $\mathbf{C}^{mn}$ of $m \times n$ matrices could be constructed, and some of them are in use. Let us mention *Frobenius* or *Hilbert–Schmidt* norm:

$$\hat{A} = [a_{ij}], \quad \boldsymbol{a}_j = [a_{ij}], \qquad \|\hat{A}\|_{\text{F}}^2 \coloneqq \sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 = \sum_{j=1}^{n} \|\boldsymbol{a}_j\|_2^2 = \operatorname{tr}\big(\hat{A}^\dagger \hat{A}\big) = \operatorname{tr}\big(\hat{A}\hat{A}^\dagger\big) = \|\hat{A}^\dagger\|_{\text{F}}^2$$

Let $\hat{Q}$ be an $m \times n$ unitary matrix. Then for any $\boldsymbol{x} \in \mathbf{C}^n$ we have $\|\hat{Q}\boldsymbol{x}\|_2 = \|\boldsymbol{x}\|_2$. More generally, for any $n \times l$ matrix $\hat{A}$ we have $\|\hat{Q}\hat{A}\|_{\text{F}} = \|\hat{A}\|_{\text{F}}$.[20]

4.2 Singular Value Decomposition (SVD)

For clearer geometrical images, let us consider a real $m \times n$ matrix $\hat{A}$, and a corresponding linear transformation $\hat{A} : \mathbf{R}^n \to \mathbf{R}^m$. An image of a $(n-1)$-sphere $\mathbf{S}^{n-1} \subset \mathbf{R}^n$ is an [hyper]ellipsoid (which could be degenerate) in $\mathbf{R}^m$, whose principal sizes (or the lengths of the principal semi-axes) and orientation are important characteristics of $\hat{A}$. Let us denote the jth, $1 \leq j \leq m$, principal semi-axis as $\sigma_j \boldsymbol{u}_j$, where $\|\boldsymbol{u}_j\|_2 = 1$, so the semi-axis length is $\sigma_j \geq 0$. Geometrically, all the semi-axes, *i.e.*, $\boldsymbol{u}$-vectors, are orthogonal to each other. (Obviously, no more than n σ’s are non-zero.)

Definition / Theorem 4: Any [real or complex] $m \times n$ matrix $\hat{A}$ has a *[reduced] Singular Value Decomposition* (SVD) $\hat{A} = \hat{U}\hat{\Sigma}\hat{V}^\dagger$, where $m \times l$ matrix $\hat{U}$ and $l \times n$ matrix $\hat{V}$ are unitary, while $l \times l$ matrix $\hat{\Sigma}$ is real diagonal, with non-negative [and non-increasing] diagonal entries. Here $l = \min(m,n)$.[21] The diagonal entries of $\hat{\Sigma}$ are called *singular values*, while the columns of the matrix $\hat{U}$ / $\hat{V}$ are called *left / right singular vectors*.

Proof [and **Algorithm** $\text{SVD}_{\hat{A}^\dagger\hat{A}}$]:[22] Let us consider the case $m \geq n$ (otherwise construct SVD of $\hat{A}^\dagger$, and then Hermite conjugate). Construct the $n \times n$ matrix $\hat{A}^\dagger\hat{A}$. It is Hermitian, thus it is diagonalizable, with real eigenvalues and orthogonal eigenvectors. It is also positive definite,[23] so all its eigenvalues are non-negative. We can write $\hat{A}^\dagger\hat{A} = \hat{V}\hat{\Sigma}^2\hat{V}^\dagger$, where $\hat{V}$ is $n \times n$ unitary matrix; matrix $\hat{\Sigma}$ is $n \times n$ real diagonal. We can choose the diagonal entries of $\hat{\Sigma}$ to be non-negative. By permuting the columns of $\hat{V}$, we can reorder the diagonal entries of $\hat{\Sigma}$ in non-increasing order. The jth column of $\hat{U}$, the vector $\boldsymbol{u}_j$, is set to $\hat{A}\boldsymbol{v}_j/\sigma_j$ if $\sigma_j > 0$, or chosen arbitrarily from the orthogonal completion to the previous columns of $\hat{U}$ if $\sigma_j = 0$. We still have to prove two statements: 1) $\hat{U}$ is unitary; and 2) $\hat{A} = \hat{U}\hat{\Sigma}\hat{V}^\dagger$.

For 1) we need to show that columns of $\hat{U}$ (it is enough to consider only non-zero σ’s) are unit vectors that are orthogonal to each other. We have $\langle \boldsymbol{u}_i, \boldsymbol{u}_j \rangle = \boldsymbol{u}_i^\dagger \boldsymbol{u}_j = (\hat{A}\boldsymbol{v}_i)^\dagger \hat{A}\boldsymbol{v}_j/\sigma_i\sigma_j = \boldsymbol{v}_i^\dagger \hat{A}^\dagger\hat{A}\boldsymbol{v}_j/\sigma_i\sigma_j = \boldsymbol{v}_i^\dagger \hat{V}\hat{\Sigma}^2\hat{V}^\dagger\boldsymbol{v}_j/\sigma_i\sigma_j = \boldsymbol{e}_i^\dagger\hat{\Sigma}^2\boldsymbol{e}_j/\sigma_i\sigma_j = \delta_{ij}$ (as $\hat{\Sigma}^2$ is diagonal).

[20] Whenever the product $\hat{U}\hat{B}\hat{V}^\dagger$ is defined, and the matrices $\hat{U}$ and $\hat{V}$ are unitary, we have $\|\hat{U}\hat{B}\hat{V}^\dagger\|_{\text{F}} = \|\hat{B}\|_{\text{F}}$.

[21] Not so rarely considered *full* SVD is a factorization $\hat{A} = \hat{U}\hat{\Sigma}\hat{V}^\dagger$, where square $m \times m$ matrix $\hat{U}$ and $n \times n$ matrix $\hat{V}$ are unitary, and $m \times n$ matrix $\hat{\Sigma}$ is diagonal (with non-negative numbers on the main diagonal).

[22] For a different proof, see [TrBa97, Theorem 4.1, p. 29].

[23] For any vector $\boldsymbol{v}$ we have $\langle \boldsymbol{v}, \hat{A}^\dagger\hat{A}\boldsymbol{v} \rangle = \boldsymbol{v}^\dagger\hat{A}^\dagger\hat{A}\boldsymbol{v} = (\hat{A}\boldsymbol{v})^\dagger\hat{A}\boldsymbol{v} = \|\hat{A}\boldsymbol{v}\|_2^2 \geq 0$.

For 2), vectors $\boldsymbol{v}_j$, $j = 1, \ldots, n$, form a basis of $\mathbf{C}^n$ [why?], so any vector $\boldsymbol{v} \in \mathbf{C}^n$ is a [unique] linear combination of them. It is enough to check $\hat{A}\boldsymbol{v}_j = \hat{U}\hat{\Sigma}\hat{V}^\dagger\boldsymbol{v}_j = \hat{U}\sigma_j\boldsymbol{e}_j = \sigma_j\boldsymbol{u}_j$ for all $1 \le j \le n$. If $\sigma_j > 0$, then we have $\hat{A}\boldsymbol{v}_j = \sigma_j\boldsymbol{u}_j$ by construction of $\boldsymbol{u}_j$. If $\sigma_j = 0$, then $\boldsymbol{v}_j^\dagger\hat{A}^\dagger\hat{A}\boldsymbol{v}_j = \sigma_j^2 = 0 = \|\hat{A}\boldsymbol{v}_j\|_2^2$, thus $\hat{A}\boldsymbol{v}_j = \mathbf{0} = \sigma_j\boldsymbol{u}_j$.

Once the singular values are ordered, the matrix $\hat{\Sigma}$ is unique. For any j we can multiply $\boldsymbol{v}_j$ by any number c with absolute value $|c| = 1$ (only $c = -1$ is interesting in real case), and both $\|\boldsymbol{v}_j\|_2$ and $\|\hat{A}\boldsymbol{v}_j\|_2$ are not going to change. For some matrices (when singular values coincide) the choice for $\hat{V}$ is even richer. *I.e.*, for any square unitary matrix $\hat{V}$ we have $\hat{I} = \hat{V}^\dagger\hat{V}$ as the SVD of the identity matrix $\hat{I}$. Generally, we can make an arbitrary unitary rotation of $\hat{V}$'s (and simultaneously of $\hat{U}$'s) part that corresponds to the same singular value.

Example 4: Consider the matrix on the right. Both of its eigenvalues are equal to 1. Let us proceed with the SVD of $\hat{A}$ in Octave[24] and on paper:

$$\hat{A} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

```
[...]/teaching/2019-4/math_575a/notes/Octave$ octave-cli
GNU Octave, version 4.2.1
[... copyright notice and links ...]
octave:1> format long
octave:2> A = [1, 1; 0, 1]
A =

   1   1
   0   1

octave:3> [U, S, V] = svd(A)
U =

   0.850650808352040  -0.525731112119134
   0.525731112119134   0.850650808352040

S =

Diagonal Matrix

   1.618033988749895                   0
                   0   0.618033988749895

V =

   0.525731112119134  -0.850650808352040
   0.850650808352040   0.525731112119134

octave:4> B = U * S * V'
B =

   1.00000000000000e+00   1.00000000000000e+00
   1.11022302462516e-16   1.00000000000000e+00

octave:5>
```

$$\hat{A}^\dagger\hat{A} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$

$$\det\left(\lambda\hat{I}_2 - \hat{A}^\dagger\hat{A}\right) = \lambda^2 - 3\lambda + 1$$

$$\sigma_{1,2}^2 = \lambda_{1,2} = \tfrac{1}{2}\left(3 \pm \sqrt{5}\right)$$

$$\sigma_{1,2} = \sqrt{\lambda_{1,2}} = \tfrac{1}{2}\left(\sqrt{5} \pm 1\right) = \varphi^{\pm 1}$$

$$\hat{A}^\dagger\hat{A}\underbrace{\begin{bmatrix} 1 \\ \varphi \end{bmatrix}}_{\sqrt{1+\varphi^2}\,\boldsymbol{v}_1} = \left(\varphi^2 = \sigma_1^2\right)\begin{bmatrix} 1 \\ \varphi \end{bmatrix}$$

note: $1 + \varphi = \varphi^2$

$$\boldsymbol{u}_1 = \frac{\hat{A}\boldsymbol{v}_1}{\sigma_1} = \frac{\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ \varphi \end{bmatrix}}{\varphi\sqrt{1+\varphi^2}} = \frac{\begin{bmatrix} \varphi \\ 1 \end{bmatrix}}{\sqrt{1+\varphi^2}}$$

find $\boldsymbol{v}_2$ and $\boldsymbol{u}_2$ in a similar way, or from orthogonality

[24] 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.

The Algorithm $\mathrm{SVD}_{\hat{A}^\dagger\hat{A}}$ is good for pen and paper calculations, but it is not numerically stable. The problem of diagonalization of $\hat{A}^\dagger\hat{A}$ has condition number $\kappa^2(\hat{A})$, instead of $\kappa(\hat{A})$.

We will not discuss numerical algorithms for the computation of SVD in detail. See, *e.g.*, [GoVa96, Sec. 8.6] and [Dem97, Sec. 5.4].

Here are some properties of SVD (here $\hat{A}$ is an $m\times n$ matrix, and $l=\min(m,n)$):

$$\hat{A}=\sum_{i=1}^{l}\sigma_i\boldsymbol{u}_i\boldsymbol{v}_i^\dagger,\qquad \|\hat{A}\|_{\mathrm{F}}^2=\sum_{i=1}^{l}\sigma_i^2,\qquad \|\hat{A}\|_2=\sigma_1,\qquad \big\langle\sigma_1,\boldsymbol{u}_1,\boldsymbol{v}_1\big\rangle=\operatorname*{arg\,min}_{\langle\sigma_1,\boldsymbol{u}_1,\boldsymbol{v}_1\rangle}\|\hat{A}-\sigma_1\boldsymbol{u}_1\boldsymbol{v}_1^\dagger\|_{\mathrm{F}}$$

$$\hat{A}_\downarrow:=\hat{A}-\sigma_1\boldsymbol{u}_1\boldsymbol{v}_1^\dagger,\qquad \big(\hat{A}_\downarrow\big)_\downarrow=\hat{A}-\sigma_1\boldsymbol{u}_1\boldsymbol{v}_1^\dagger-\sigma_2\boldsymbol{u}_2\boldsymbol{v}_2^\dagger,\qquad \sigma_1(\hat{A}_\downarrow)=\sigma_2(\hat{A})$$

$\sum_{i=1}^{r}\sigma_i\boldsymbol{u}_i\boldsymbol{v}_i^\dagger$ is the best approximation (in $\|\cdot\|_2$ and $\|\cdot\|_{\mathrm{F}}$ norms) of $\hat{A}$ by any matrix of rank $r\le l$

$$\big|\det\hat{A}\big|=\prod_{i=1}^{l=m=n}\sigma_i,$$ if $\hat{A}=\hat{A}^\dagger$, then its diagonalization is almost (up to signs of λ's) its SVD

4.3 Condition number of multiplication by a matrix

Consider a fixed $m\times n$ matrix $\hat{A}$. What is the [relative] condition number $\kappa(\hat{A}\cdot)$ of the problem of multiplying a vector by $\hat{A}$?[25]

If $n>m$ or the matrix $\hat{A}$ is not of full rank, then it has a non-trivial null space. Any errors in the computation of $\hat{A}\boldsymbol{x}=\boldsymbol{0}$ for $\boldsymbol{x}\in\mathrm{null}(\hat{A})$ would be considered infinite in relative sense, and thus $\kappa(\hat{A}\cdot)=\infty$ by definition. Otherwise we have

$$\kappa(\hat{A}\cdot,\boldsymbol{x})=\underbrace{\overbrace{\max_{\boldsymbol{\varepsilon}}\frac{\|\hat{A}(\boldsymbol{x}+\boldsymbol{\varepsilon})-\hat{A}\boldsymbol{x}\|}{\|\boldsymbol{\varepsilon}\|}}^{\text{sensitivity to }\boldsymbol{x}}}_{\|\hat{A}\|=\sigma_1}\frac{\|\boldsymbol{x}\|}{\|\hat{A}\boldsymbol{x}\|},\qquad \kappa(\hat{A}\cdot):=\max_{\boldsymbol{x}}\kappa(\hat{A}\cdot,\boldsymbol{x})=\frac{\sigma_1}{\sigma_n}$$

The denominator σ_n came from maximizing the ratio $\|\boldsymbol{x}\|/\|\hat{A}\boldsymbol{x}\|$.

Problems and exercises

1. Two norms $\|\cdot\|_{\mathrm{I}}$ and $\|\cdot\|_{\mathrm{II}}$ are called *equivalent* if there exist constants $0<C_1\le C_2$ such that $C_1\|\boldsymbol{x}\|_{\mathrm{I}}\le\|\boldsymbol{x}\|_{\mathrm{II}}\le C_2\|\boldsymbol{x}\|_{\mathrm{I}}$ for all $\boldsymbol{x}$. (a) Show that any two norms in $\mathbf{R}^m$ are equivalent. (b) Find constants C_1 and C_2 for any pair from $\|\cdot\|_1$, $\|\cdot\|_2$, and $\|\cdot\|_\infty$ norms.

2. Consider two operators $\hat{A}:X\to Y$ and $\hat{B}:Y\to Z$, where the vector spaces X, Y, and Z are normed. For induced norms, show that $\|\hat{B}\hat{A}\|\le\|\hat{A}\|\,\|\hat{B}\|$.

3. Consider "dot product with $\boldsymbol{u}$" operator $\boldsymbol{u}^\dagger:\mathbf{C}^n\to\mathbf{C}$, where $\boldsymbol{x}\mapsto\boldsymbol{u}\cdot\boldsymbol{x}=\boldsymbol{u}^\dagger\boldsymbol{x}$. Find its (a) induced L^2-norm and (b) Frobenius norm.

4. Consider "multiplying by $\boldsymbol{u}$" operator $\boldsymbol{u}:\mathbf{C}\to\mathbf{C}^m$, where $x\mapsto x\boldsymbol{u}$. Find its (a) induced L^2-norm and (b) Frobenius norm.

5. Let $\hat{A}$ be an $m\times n$ matrix, with an SVD $\hat{A}=\hat{U}\hat{\Sigma}\hat{V}^\dagger$. Find the SVD of $\hat{A}^\dagger$.

6. Let $\hat{A}$ be an $m\times n$ matrix. Consider $(m+n)\times(m+n)$ Hermitian matrix $\hat{B}:=\begin{bmatrix}\hat{O}_{n,n} & \hat{A}^\dagger\\ \hat{A} & \hat{O}_{m,m}\end{bmatrix}$. How the singular values of $\hat{A}$ and the eigenvalues of $\hat{B}$ are connected?

7. Show that $m\times n$, $m\ge n$, matrix is unitary if and only if all of its singular values are equal to 1.

[25] The so called *condition number* $\kappa(\hat{A})$ of a matrix $\hat{A}$ is defined in a similar but slightly different way: $\kappa(\hat{A})=\|\hat{A}\|_2\|\hat{A}^+\|_2$, where $\hat{A}^+$ is the Moore–Penrose pseudoinverse of $\hat{A}$.

5 Systems of linear equations

Consider a system of linear equations $\hat{A}\boldsymbol{x} = \boldsymbol{b}$, with $m \times n$ matrix $\hat{A}$. A way to interpret the system, which is often good while thinking theoretically, is: $\hat{A}$ is the matrix of a linear transformation $\mathbf{C}^n \to \mathbf{C}^m$, and we try to find such vector $\boldsymbol{x} \in \mathbf{C}^n$ that is transformed to $\boldsymbol{b}$, *i.e.*, $\hat{A} : \boldsymbol{x} \mapsto \boldsymbol{b}$.

Let us assume for simplicity that $\hat{A}$ is a full rank $m \times m$ matrix. Interpreting the system in way I), the solution could be found by computing the inverse matrix and reversing the transformation $\hat{A}$:

Algorithm $\hat{A}^{-1}\boldsymbol{b}$: Input: $\hat{A}$ and $\boldsymbol{b}$. Output: $\boldsymbol{x} := \hat{A}^{-1}\boldsymbol{b}$, *i.e.*, you compute the inverse matrix $\hat{A}^{-1}$ and multiply the r.h.s. $\boldsymbol{b}$ by it.

Example 5: "Failure" of $\hat{A}^{-1}\boldsymbol{b}$ in case of poorly conditioned matrix $\hat{A}$.

```
octave:1> format long
octave:2> A = [1 sqrt(2); sqrt(3) sqrt(6) + 1.e-13]
A =

   1.000000000000000   1.414213562373095
   1.732050807568877   2.449489742783278
```

$$\hat{A} = \begin{bmatrix} 1 & \sqrt{2} \\ \sqrt{3} & \sqrt{6}+10^{-13} \end{bmatrix}$$

```
octave:3> cond(A)
ans =  119564591877097.7
octave:4> B = inverse(A)
B =

   24496352586638.59  -14142975760059.20
  -17321537028348.06   10000594066094.83
```

$\sigma_1 \sim 1$ and $\sigma_1\sigma_2 = \det\hat{A} = 10^{-13}$

```
octave:5> b = [sqrt(3); 3]
b =

   1.732050807568877
   3.000000000000000
```

$$\boldsymbol{b} = \begin{bmatrix} \sqrt{3} \\ 3 \end{bmatrix}$$

```
octave:6> x = A \ b
x =

   1.721172241704469e+00
   7.692307692307693e-03
```

solution by $\backslash$ (or `mldivide`) operator

```
octave:7> y = B * b
y =

   1.726562500000000e+00
   3.906250000000000e-03
```

solution by Algorithm $\hat{A}^{-1}\boldsymbol{b}$

```
octave:8> A * x - b
ans =

  -2.220446049250313e-16
   0.000000000000000e+00

octave:9> A * y - b
ans =
```

```
   3.596415914275397e-05
   6.229175088678929e-05
```

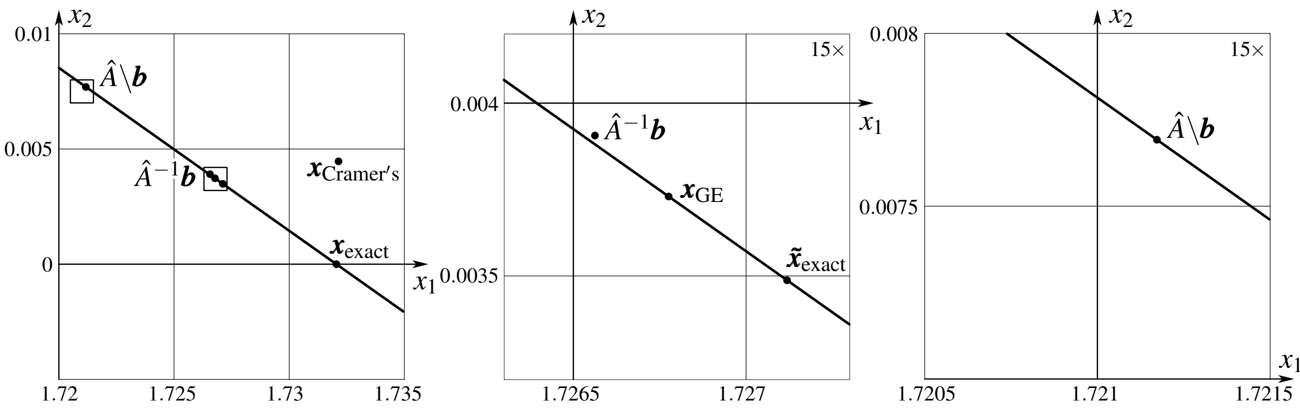
residual $\hat{A}(\hat{A}^{-1}\boldsymbol{b}) - \boldsymbol{b}$ is not too small

```
octave:10> A * [sqrt(3); 0] - b
ans =

   0.000000000000000e+00
  -4.440892098500626e-16

octave:11>
```

$\boldsymbol{x} = \begin{bmatrix} \sqrt{3} \\ 0 \end{bmatrix}$ is exact solution

$\hat{A}^{-1}\boldsymbol{b}$ solution is numerically closer to it than solution by \, but $\hat{A}^{-1}\boldsymbol{b}$ provides orders of magnitude larger residual

This is a [forward] stable, but not a backward stable algorithm. The vector $\boldsymbol{x}$ is not far from the exact solution, but the discrepance is such that it is not very plausible:



Here the thick line is $\boldsymbol{x}_{\text{exact}} + t\boldsymbol{v}_2$, where $\boldsymbol{v}_1 \approx \begin{bmatrix} 1 & \sqrt{2} \end{bmatrix}^{\mathrm{T}}/\sqrt{3}$ and $\boldsymbol{v}_2 \approx \begin{bmatrix} \sqrt{2} & -1 \end{bmatrix}^{\mathrm{T}}/\sqrt{3}$ are the right singular vectors of $\hat{A}$. The vector $\hat{A}^{-1}\boldsymbol{b} - \boldsymbol{x}_{\text{exact}}$ has some $\boldsymbol{v}_1$ component, which gives rise to not so small residual $\hat{A}(\hat{A}^{-1}\boldsymbol{b}) - \boldsymbol{b}$.

Let us take the matrix $\hat{A}$ and the r.h.s. $\boldsymbol{b}$, put them into a computer with double precision (also solve $\hat{A}\boldsymbol{x} = \boldsymbol{b}$ by Gaussian elimination for illustration), and then *exactly* solve the system of 2 linear equations with floating point number representations of $\hat{A}$ and $\boldsymbol{b}$, thinking about them as *exact* input:

```
[...]/teaching/2019-4/math_575a/notes$ cat C/A_x_eq_b.c
#include <stdio.h>
#include <math.h>
void print_bits(double x, char* s) { int i; unsigned char c;
  for (i = 0; i < 8; i++) for (c = 128; c != 0; c >>= 1)
    if (*((char *)(&x) + 7 - i) & c) printf("1"); else printf("0");
  printf("  %s\n", s); }
int main() { int i; double A[2][2], b[2], x[2], det, G[2][2], x52;
  A[0][0] = 1.; A[0][1] = sqrt(2.); /* Example 5 */
  A[1][0] = sqrt(3.); A[1][1] = sqrt(6.) + 1.e-13;
  print_bits(A[0][0], "a_11"); print_bits(A[0][1], "a_12");
  print_bits(A[1][0], "a_21"); print_bits(A[1][1], "a_22");
  b[0] = sqrt(3.); b[1] = 3.;
  print_bits(b[0], "b_1"); print_bits(b[1], "b_2");
printf("|\\___  _____/\\_________________________  _______________________/\n");
printf("|    \\/                                    \\/\n");
printf("sign  exponent (11 bits)               fraction (52 bits)\n\n");
  x52 = 1.; for (i = 0; i < 52; i++) x52 *= 2.; /* x52 = 2^52 */
  printf("a_12 = %18.1f / 2^52,   a_21 = b_1 = %18.1f / 2^52\n",
```

```
    x52 * A[0][1], x52 * A[1][0]);
  printf("a_11 = 1,   b_2 = 3,   a_22 = %18.1f / 2^52\n\n", x52 * A[1][1]);

  printf("solution by Cramer's rule:\n");
  det = 1. / (A[0][0] * A[1][1] - A[0][1] * A[1][0]);
  x[0] = (b[0] * A[1][1] - b[1] * A[0][1]) * det;
  x[1] = (A[0][0] * b[1] - A[1][0] * b[0]) * det;
  printf("   x = [%22.16e %22.16e]^T\n", x[0], x[1]);
  printf("   residual A * x - b: [%e %e]^T\n", x[0] + A[0][1] * x[1] - b[0],
    A[1][0] * x[0] + A[1][1] * x[1] - b[1]);

  printf("solution A^(-1) b:\n");
  G[0][0] = A[1][1] * det; G[0][1] = -A[0][1] * det;
  G[1][0] = -A[1][0] * det; G[1][1] = A[0][0] * det;
  x[0] = G[0][0] * b[0] + G[0][1] * b[1];
  x[1] = G[1][0] * b[0] + G[1][1] * b[1];
  printf("   x = [%22.16e %22.16e]^T\n", x[0], x[1]);
  printf("   residual A * x - b: [%e %e]^T\n", x[0] + A[0][1] * x[1] - b[0],
    A[1][0] * x[0] + A[1][1] * x[1] - b[1]);

  printf("solution by Gaussian elimination with complete pivoting:\n");
  G[1][0] = A[1][0] / A[1][1]; x[1] = b[1] / A[1][1];
  G[0][0] = A[0][0] - A[0][1] * G[1][0]; x[0] = b[0] - A[0][1] * x[1];
  x[0] /= G[0][0]; x[1] -= G[1][0] * x[0];
  printf("   x = [%22.16e %22.16e]^T\n", x[0], x[1]);
  printf("   residual A * x - b: [%e %e]^T\n", x[0] + A[0][1] * x[1] - b[0],
    A[1][0] * x[0] + A[1][1] * x[1] - b[1]);
  return 0; }
[...]/teaching/2019-4/math_575a/notes$ cc C/A_x_eq_b.c -lm ; ./a.out
0011111111110000000000000000000000000000000000000000000000000000  a_11
0011111111110110101000001001111001100110011111110011101111001101  a_12
0011111111111011101101100111101011101000010110000100110010101010  a_21
0100000000000011100110001000111000010100000010010010001000001111  a_22
0011111111111011101101100111101011101000010110000100110010101010  b_1
0100000000001000000000000000000000000000000000000000000000000000  b_2
|\___  _____/\________________________  _______________________/
|    \/                                \/
sign  exponent (11 bits)                fraction (52 bits)

a_12 = 6369051672525773.0 / 2^52,   a_21 = b_1 = 7800463371553962.0 / 2^52
a_11 = 1,   b_2 = 3,   a_22 = 11031521092846622.0 / 2^52

solution by Cramer's rule:
   x = [1.7321428571428570e+00 4.4642857142857140e-03]^T
   residual A * x - b: [6.405503e-03 1.109466e-02]^T
solution A^(-1) b:
   x = [1.7265625000000000e+00 3.9062500000000000e-03]^T
   residual A * x - b: [3.596416e-05 6.229175e-05]^T
solution by Gaussian elimination with complete pivoting:
   x = [1.7267759562841529e+00 3.7298831131791221e-03]^T
   residual A * x - b: [0.000000e+00 0.000000e+00]^T
[...]/teaching/2019-4/math_575a/notes$ gp
[... technical stuff ...]
                  GP/PARI CALCULATOR Version 2.11.1 (released)
```

```
[... copyright notice and links, technical parameters ...]
? a11 = 2^52; a12 = 6369051672525773; a21 = b1 = 7800463371553962; a22 = 1103152
1092846622; b2 = 3 * a11; det = a11 * a22 - a12 * a21
%1 = 2021703648790801886
? x1 = (b1 * a22 - b2 * a12) / det
%2 = 102697713703618710/59461872023258879
? x2 = (a11 * b2 - a21 * b1) / det
%3 = 3525341537980302/1010851824395400943
? 1. * x1
%4 = 1.7271187436454719105197015986197581099
? 1. * x2
%5 = 0.0034874958454854040546168999582954249611
```

The *exact* solution, $\tilde{\boldsymbol{x}}_{\text{exact}} \approx \left[\begin{array}{cc} 1.7271 & 0.0035 \end{array}\right]^{\mathrm{T}}$, of the "computerized" system (*i.e.*, how it looks like *after* the input data $\hat{A}$ and $\boldsymbol{b}$ are put into $\mathbf{FP}_{52}$), is as different from non-computerized $\boldsymbol{x}_{\text{exact}} = \left[\begin{array}{cc} \sqrt{3} & 0 \end{array}\right]^{\mathrm{T}}$ as solutions obtained by Gaussian elimination, $\boldsymbol{x}_{\text{GE}} \approx \left[\begin{array}{cc} 1.7268 & 0.0037 \end{array}\right]^{\mathrm{T}}$, or `mldivide` operator, $\hat{A}\backslash\boldsymbol{b} \approx \left[\begin{array}{cc} 1.7212 & 0.0077 \end{array}\right]^{\mathrm{T}}$.

5.1 System with orthogonal matrix

Consider a system $\hat{Q}\boldsymbol{x} = \boldsymbol{b}$, where $\hat{Q}$ is an $m \times m$ orthogonal or unitary matrix. The solution is $\boldsymbol{x} \coloneqq \hat{Q}^{\dagger}\boldsymbol{b}$, which could be viewed as the one obtained by the Algorithm $\hat{A}^{-1}\boldsymbol{b}$ (we have $\hat{Q}^{-1} = \hat{Q}^{\dagger}$). Here the matrix of the system $\hat{Q}$ is well conditioned though, $\kappa(\hat{Q}\cdot) = 1$. The L^2-norm of the residual is small: $\left\|\hat{Q}\big(\text{num}(\hat{Q}^{\dagger}) \odot \text{num}(\boldsymbol{b})\big) - \boldsymbol{b}\right\|_2 = \left\|\big(\text{num}(\hat{Q}^{\dagger}) \odot \text{num}(\boldsymbol{b})\big) - \hat{Q}^{\dagger}\boldsymbol{b}\right\|_2$ — we have $\hat{Q}\hat{Q}^{\dagger} = \hat{I}_m$, and multiplying a vector by unitary matrix $\hat{Q}^{\dagger}$ doesn't change its L^2-norm.

5.2 System with triangular matrix (see [TrBa97, Lec. 17])

Consider a system $\hat{R}\boldsymbol{x} = \boldsymbol{b}$, where $\hat{R}$ is an $n \times n$ upper triangular matrix, *i.e.*, $r_{ij} = 0$ whenever $i > j$. We assume that $\det\hat{R} = r_{11}r_{22}...r_{nn} \neq 0$, *i.e.*, the solution does exist and is unique. It can be found by a procedure called *back substitution* (the matrix of the system is already in echelon form, *i.e.*, most of the work of excluding variables is done): the last equation means $r_{nn}x_n = b_n$, so we immediately find $x_n \coloneqq b_n/r_{nn}$. In the equation $r_{n-1,n-1}x_{n-1} + r_{n-1,n}x_n = b_{n-1}$ only x_{n-1} is unknown, so we immediately find it: $x_{n-1} = (b_{n-1} - r_{n-1,n}x_n)/r_{n-1,n-1}$. Next we find x_{n-2}, and so on, till we finally find x_1.

Algorithm "Back Substitution": Input: upper triangular matrix $\hat{R}$ and the r.h.s. $\boldsymbol{b}$. Output: vector $\boldsymbol{x}$, computed according to the following pseudo-code:

`for` $i = n, n-1, ..., 2, 1$ `do`
  $B \coloneqq b_i$
  `for` $j = n, n-1, ..., i+1$ `do` (do nothing if $n < i+1$, *i.e.*, if $i = n$)
    $B \coloneqq B \ominus (r_{ij} \odot x_j)$
  $x_i \coloneqq B \oslash r_{ii}$
`return` $\boldsymbol{x}$

Alternatively, one can go from $i+1$ to n in `for` loop over j. There are $n(n-1)/2$ multiplications $r \odot x$, $n(n-1)/2$ subtractions $B \ominus (rx)$, and n divisions $B \oslash r$; overall n^2 floating point operations.

The pseudo-code could be written in a different form (which ruins the input vector $\boldsymbol{b}$, though):

```
for i = n, n − 1, ..., 2, 1 do
    x_i := b_i ⊘ r_ii
    for j = i − 1, i − 2, ..., 1 do        (do nothing if i − 1 < 1, i.e., if i = 1)
        b_j := b_j ⊖ (r_ji ⊙ x_i)
return x
```

Theorem 5: The Algorithm "Back Substitution" is backward stable. Moreover, we can interpret the output as the exact solution for the problem $(\hat{R}+\Delta\hat{R})\boldsymbol{x}=\boldsymbol{b}$ with $\|\Delta\hat{R}\| \sim \varepsilon_{\text{machine}}\|\hat{R}\|$, *i.e.*, we need to slightly change only the matrix $\hat{R}$, with the r.h.s. $\boldsymbol{b}$ being untouched. Moreover, we can show that $\Delta r_{ij} \sim \varepsilon_{\text{machine}} r_{ij}$, *i.e.*, we can choose the matrix $\Delta\hat{R}$ in such a way, that each of its matrix elements is a small, $\sim \varepsilon_{\text{machine}}$, change of the corresponding matrix element of $\hat{R}$. More specifically, $\left|(\Delta r_{ij})/r_{ij}\right| \le (n+1-j)\varepsilon_{\text{machine}} + O(\varepsilon^2_{\text{machine}})$.[26]

Proof: Consider the case $n=3$ and the calculation of x_1:

$$x_1 := \Big(\big(b_1 \ominus (r_{13} \odot x_3)\big) \ominus (r_{12} \odot x_2)\Big) \oslash r_{11}$$

$$x_1 = \Big(\big(b_1 - r_{13}x_3(1+\varepsilon_{13})\big)(1+\delta_{13}) - r_{12}x_2(1+\varepsilon_{12})\Big)(1+\delta_{12}) \cdot \frac{(1+\eta_1)}{r_{11}}$$

Here $|\varepsilon_{ij}|$, $|\delta_{ij}|$, and $|\eta_i|$ are all not greater than $\varepsilon_{\text{machine}}$: ε's are numerical errors introduced in calculation $r_{ij}x_j$, δ's are introduced while doing subtractions, and η's are due to divisions. The formula for x_1 could be rewritten as

$$x_1 = \frac{b_1 - r_{13}x_3(1+\varepsilon_{13}) - r_{12}x_2(1+\varepsilon_{12})(1+\delta_{13})^{-1}}{r_{11}(1+\eta_1)^{-1}(1+\delta_{12})^{-1}(1+\delta_{13})^{-1}}$$

Whenever we are doing subtraction, we push the numerical error as a change in r's on the left. Each non-diagonal r has potential need for change due to rx multiplications. Diagonal r's have additional potential need to change due to divisions.

Whenever we present the matrix $\hat{A}$ of the system $\hat{A}\boldsymbol{x}=\boldsymbol{b}$ as the product of unitary or triangular (partial case: diagonal) matrices, $\hat{A}=\hat{A}_1\hat{A}_2...\hat{A}_k$, we can find $\boldsymbol{x}$ as follows: First, solve $\hat{A}_1\boldsymbol{y_1}=\boldsymbol{b}$. This is easy, we have $\boldsymbol{y}_1=\hat{A}_1^\dagger\boldsymbol{b}$, if $\hat{A}_1$ is unitary; or we find $\boldsymbol{y}_1$ by back ($\hat{A}_1$ is upper triangular) or by forward ($\hat{A}_1$ is lower triangular, you subsequently find components of $\boldsymbol{y}_1$ from the first to the last) substitution. Then we solve $\hat{A}_2\boldsymbol{y}_2=\boldsymbol{y}_1$, $\hat{A}_3\boldsymbol{y}_3=\boldsymbol{y}_2$, and so on. Finally, we deal with $\hat{A}_k\boldsymbol{x}=\boldsymbol{y}_{k-1}$. We have

$$\boldsymbol{b} = \hat{A}_1\Big(\boldsymbol{y}_1 = \hat{A}_2\Big(\boldsymbol{y_2} = \hat{A}_3\big(\boldsymbol{y}_3 = ...\hat{A}_{k-1}(\boldsymbol{y}_{k-1} = \hat{A}_k\boldsymbol{x})\big)\Big)\Big), \qquad \boldsymbol{b} = \hat{A}_1\hat{A}_2...\hat{A}_k\boldsymbol{x} = \hat{A}\boldsymbol{x}$$

The commonly used factorizations of $\hat{A}$ are:

QR factorization :	$\hat{A}=\hat{Q}\hat{R}$,	$\hat{Q}$ is unitary, $\hat{R}$ is upper triangular
with column pivoting :	$\hat{A}=\hat{Q}\hat{R}\hat{P}$,	$\hat{P}$ is [column] permutation matrix
Gaussian elimination, LU factorization :	$\hat{A}=\hat{L}\hat{U}$,	$\hat{L}/\hat{U}$ is lower / upper triangular
with partial pivoting :	$\hat{A}=\hat{P}\hat{L}\hat{U}$,	$\hat{P}$ is [row] permutation matrix
with complete pivoting :	$\hat{A}=\hat{P}\hat{L}\hat{U}\hat{Q}$,	$\hat{Q}$ is [column] permutation matrix

[26] This is a different pattern or $\varepsilon_{\text{machine}}$-perturbations of $\hat{R}$ than the one given in [TrBa97, p. 126], because there $r_{ij}x_j$ terms are subtracted from b_i in different order (from left to right).

Problems and exercises

1. Consider the system $\hat{A}\boldsymbol{x} = \boldsymbol{b}$, where

$$\hat{A} = \hat{U}\hat{\Sigma}\hat{V}^{\dagger} = \begin{bmatrix} 0.8 & -0.6 \\ 0.6 & 0.8 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \varepsilon \end{bmatrix} \begin{bmatrix} 0.6 & 0.8 \\ -0.8 & 0.6 \end{bmatrix}, \qquad \boldsymbol{b} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$$

Find $\boldsymbol{x}$ analytically. Solve the system numerically for $\varepsilon = k \cdot 10^{-14}$, $k = 1, 2, ..., 10$, by 1) computing $\hat{A}$, then applying the Cramer's rule; 2) $\boldsymbol{x} := \hat{V}\hat{\Sigma}^{-1}\hat{U}^{\dagger}\boldsymbol{b}$. Plot all 20 numerical solutions on one plot.

6 QR factorization

Definition/Theorem 6: Any [real or complex] $m \times n$, $m \geq n$, matrix $\hat{A}$ has a *[reduced] QR factorization* $\hat{A} = \hat{Q}\hat{R}$, where $m \times n$ matrix $\hat{Q}$ is unitary, while $n \times n$ matrix $\hat{R}$ is upper triangular.[27] If $\hat{A}$ is of full rank, *i.e.*, rank$\hat{A} = n$, then there is only one QR factorization with [strictly] positive diagonal entries of $\hat{R}$.

Proof: Matrices $\hat{Q}$ and $\hat{R}$ could be produced by the Gram–Schmidt process. In case of full rank, the columns of $\hat{Q}$ are defined up to a multiplicative factor with absolute value equal to 1, which is uniquely set if one requires $r_{ii} > 0$ for all $1 \leq i \leq n$.

There are two major approaches for obtaining the QR factorization of a matrix (here $\hat{A}$ is an $m \times n$ matrix with $m \geq n$, on schematic pictures the long/short sides have lengths m/n). One strategy, employed in Gram–Schmidt process, is to apply "upper triangular" column operations to matrix $\hat{A}$, in order to make it unitary:

$$\hat{A}\ \hat{I}_n \longrightarrow \underbrace{\hat{Q}_{k-1}\ \hat{U}_{k-1}}_{\hat{Q}_k}\ \underbrace{\hat{U}_k^{-1}\ \hat{R}_{k-1}}_{\hat{R}_k} \longrightarrow \hat{Q}\ \hat{R}$$

$$\hat{Q} = \hat{A}\hat{U}_1\hat{U}_2\hat{U}_3 ... \qquad \hat{R} = ...\hat{U}_3^{-1}\hat{U}_2^{-1}\hat{U}_1^{-1}$$

Another strategy, employed in Householder reflections and Givens rotations methods, is to act by unitary matrices $\hat{V}_1$, $\hat{V}_2$, $\hat{V}_3$, ..., on the matrix $\hat{A}$ until it becomes upper triangular:

$$\hat{I}_m\ \hat{A} \longrightarrow \underbrace{\hat{Q}_{k-1}\ \hat{V}_k^{\dagger}}_{\hat{Q}_k}\ \underbrace{\hat{V}_k\ \hat{R}_{k-1}}_{\hat{R}_k} \longrightarrow \hat{Q}\ \hat{R}$$

$$\hat{Q} = \hat{V}_1^{\dagger}\hat{V}_2^{\dagger}\hat{V}_3^{\dagger} ... \qquad \hat{R} = ...\hat{V}_3\hat{V}_2\hat{V}_1\hat{A}$$

[27] A *full* QR factorization is $\hat{A} = \hat{Q}\hat{R}$, where square $m \times m$ matrix $\hat{Q}$ is unitary, and $m \times n$ matrix $\hat{R}$ is upper triangular (with zero matrix elements below the main diagonal).

6.1 Gram–Schmidt process

Algorithms "Classical / Modified Gram–Schmidt": Input: $m\times n$ matrix $\hat{A}$, $m\geq n$. Output: $m\times n$ unitary matrix $\hat{Q}$ and $n\times n$ upper triangular matrix $\hat{R}$, such that $\hat{A}=\hat{Q}\hat{R}$, according to the following pseudo-code:[28]

classical Gram–Schmidt

```
for j = 1, 2, ..., n do
    q_j := a_j
    for i = 1, 2, ..., j-1 do
        r_ij := q_i† a_j
        q_j := q_j - r_ij q_i
    r_jj := ||q_j||_2
    q_j := q_j / r_jj
return Q̂, R̂
```

$\longrightarrow$ modification $r_{ij}:=\boldsymbol{q}_i^\dagger\boldsymbol{q}_j$ $\longrightarrow$

modified Gram–Schmidt[29]

```
for i = 1, 2, ..., n do
    r_ii := ||a_i||_2
    a_i := a_i / r_ii
    for j = i+1, i+2, ..., n do
        r_ij := a_i† a_j
        a_j := a_j - r_ij a_i
return Â, R̂
```

On the right is the pseudo-code for Modified Gram–Schmidt that is equivalent (just operations are done in different order) to the algorithm after $r_{ij}:=\boldsymbol{q}_i^\dagger\boldsymbol{a}_j\to\boldsymbol{q}_i^\dagger\boldsymbol{q}_j$ modification.

Example 6: Let us compute the QR factorization of $\hat{A}$ using both classical (cGS) and modified (mGS) Gram–Schmidt methods. We will assume that $\varepsilon^2\ll\varepsilon_{\text{machine}}\ll\varepsilon$ in our calculations (*i.e.*, we will drop ε^2 terms when added to something of the order of 1).

$$\hat{A}=\begin{bmatrix}1&1&1\\ \varepsilon&0&0\\ 0&\varepsilon&0\\ 0&0&\varepsilon\end{bmatrix}$$

The 1st column is already as good as normalized ($\|\boldsymbol{a}_1\|_2^2=1+\varepsilon^2\approx 1$), so $r_{11}=1$ and $\boldsymbol{q}_1=\boldsymbol{a}_1$. We have $r_{12}=\boldsymbol{q}_1^\dagger\boldsymbol{a}_2=1$, and $r_{22}=\sqrt{2}\varepsilon$, $\boldsymbol{q}_2=\begin{bmatrix}0&-1&1&0\end{bmatrix}^{\mathrm{T}}/\sqrt{2}$ — so far there is no difference between classical and modified versions of the Gram–Schmidt processes. Now in classical version we have $r_{13}=\boldsymbol{q}_1^\dagger\boldsymbol{a}_3=\boldsymbol{a}_1^\dagger\boldsymbol{a}_3=1$ and $r_{23}=\boldsymbol{q}_2^\dagger\boldsymbol{a}_3=0$, so we get $r_{33}=\sqrt{2}\varepsilon$, $\boldsymbol{q}_3=\begin{bmatrix}0&-1&0&1\end{bmatrix}^{\mathrm{T}}/\sqrt{2}$. The cGS QR factorization is

$$\hat{Q}_{\text{cGS}}=\begin{bmatrix}1&0&0\\ \varepsilon&-1/\sqrt{2}&-1/\sqrt{2}\\ 0&1/\sqrt{2}&0\\ 0&0&1/\sqrt{2}\end{bmatrix},\qquad \hat{R}_{\text{cGS}}=\begin{bmatrix}1&1&1\\ 0&\sqrt{2}\varepsilon&0\\ 0&0&\sqrt{2}\varepsilon\end{bmatrix}$$

Indeed, we have $\hat{Q}_{\text{cGS}}\hat{R}_{\text{cGS}}=\hat{A}$, but the matrix $\hat{Q}_{\text{cGS}}$ is far from unitary: $\boldsymbol{q}_{\text{cGS},2}\cdot\boldsymbol{q}_{\text{cGS},3}=1/2\neq 0$.

In modified version we have $r_{13}=1$, and the column $\boldsymbol{a}_3$ is then orthogonalized to $\boldsymbol{q}_1$, becoming $\begin{bmatrix}0&-\varepsilon&0&\varepsilon\end{bmatrix}^{\mathrm{T}}$. Only after that it is attempted to be orthogonalized to $\boldsymbol{q}_2$: we have $r_{23}=\varepsilon/\sqrt{2}$, the

[28] If for some $1\leq i\leq n$ the matrix element r_{ii} is computed to be 0, then $\boldsymbol{q}_i$ is arbitrarily chosen from unit vectors orthogonal to $\boldsymbol{q}_1,\boldsymbol{q}_2,\ldots,\boldsymbol{q}_{i-1}$. *E.g.*,
$$\begin{bmatrix}4&5.6\\3&4.2\end{bmatrix}=\begin{bmatrix}0.8&-0.6\\0.6&0.8\end{bmatrix}\begin{bmatrix}5&7\\0&0\end{bmatrix}$$
Here, as $\boldsymbol{a}_2=7\boldsymbol{q}_1$, the matrix element r_{22} ends up to be zero, so $\boldsymbol{q}_2$ is chosen to be orthogonal to $\boldsymbol{q}_1$.

[29] If we do not want to overwrite $\hat{A}$, then we can copy $\hat{A}$ at the start of the algorithm and do calculations with the copy.

subtraction of $r_{23}\boldsymbol{q}_2$ brings the 3rd column to $\begin{bmatrix} 0 & -\varepsilon/2 & -\varepsilon/2 & \varepsilon \end{bmatrix}^{\mathrm{T}}$. Finally, we get

$$\hat{Q}_{\mathrm{mGS}} = \begin{bmatrix} 1 & 0 & 0 \\ \varepsilon & -1/\sqrt{2} & -1/\sqrt{6} \\ 0 & 1/\sqrt{2} & -1/\sqrt{6} \\ 0 & 0 & 2/\sqrt{6} \end{bmatrix}, \qquad \hat{R}_{\mathrm{mGS}} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & \sqrt{2}\varepsilon & \varepsilon/\sqrt{2} \\ 0 & 0 & \sqrt{3/2}\varepsilon \end{bmatrix}$$

The matrix $\hat{Q}_{\mathrm{mGS}}$ is much closer to be unitary than $\hat{Q}_{\mathrm{cGS}}$, the dot products $\boldsymbol{q}_{\mathrm{mGS},1}\cdot\boldsymbol{q}_{\mathrm{mGS},2}$ and $\boldsymbol{q}_{\mathrm{mGS},1}\cdot\boldsymbol{q}_{\mathrm{mGS},3}$ are small, of the order of ε, but still non-zero.

6.2 Householder reflections

Consider the following problem: You have a vector $\boldsymbol{x}\in\mathbf{C}^m$. Find a unitary transformation $\hat{V}$ would transform $\boldsymbol{x}$ to a vector $\boldsymbol{y}$ with only non-zero component being the 1st one? (We have $|y_1| = \|\boldsymbol{x}\|_2$ then.)

There are many such transformations, and some of them are [Householder] reflections through a hyperplane orthogonal to $\boldsymbol{v} = \boldsymbol{x} - \mathrm{e}^{\mathrm{i}\theta}\|\boldsymbol{x}\|\boldsymbol{e}_1$, *i.e.*, $\hat{V} = \hat{I}_m - 2\boldsymbol{v}\boldsymbol{v}^\dagger/\|\boldsymbol{v}\|^2$. We have $\hat{V}^\dagger = \hat{V}$, and

$$\hat{V}^\dagger\hat{V} = (\hat{I}_m - 2\boldsymbol{v}\boldsymbol{v}^\dagger/\|\boldsymbol{v}\|^2)(\hat{I}_m - 2\boldsymbol{v}\boldsymbol{v}^\dagger/\|\boldsymbol{v}\|^2) = \hat{I}_m - 4\boldsymbol{v}\boldsymbol{v}^\dagger/\|\boldsymbol{v}\|^2 + 4\boldsymbol{v}\boldsymbol{v}^\dagger\boldsymbol{v}\boldsymbol{v}^\dagger/\|\boldsymbol{v}\|^4 = \hat{I}_m$$

Thus, $\hat{V}$ is unitary. It transforms vector $\boldsymbol{x}$ to

$$\hat{V}\boldsymbol{x} = \boldsymbol{x} - 2\boldsymbol{v}\boldsymbol{v}^\dagger\boldsymbol{x}/\|\boldsymbol{v}\|^2 = \boldsymbol{x} - 2\boldsymbol{v}(\|\boldsymbol{x}\|^2 - \mathrm{e}^{-\mathrm{i}\theta}\|\boldsymbol{x}\|x_1)/\|\boldsymbol{v}\|^2 =$$
$$= \Big(\underbrace{\big(2\|\boldsymbol{x}\|^2 - x_1^*\mathrm{e}^{\mathrm{i}\theta}\|\boldsymbol{x}\| - x_1\mathrm{e}^{-\mathrm{i}\theta}\|\boldsymbol{x}\|\big)}_{\|\boldsymbol{v}\|^2}\boldsymbol{x} - 2\underbrace{\big(\boldsymbol{x} - \mathrm{e}^{\mathrm{i}\theta}\|\boldsymbol{x}\|\boldsymbol{e}_1\big)}_{\boldsymbol{v}}\underbrace{\big(\|\boldsymbol{x}\|^2 - \mathrm{e}^{-\mathrm{i}\theta}\|\boldsymbol{x}\|x_1\big)}_{\boldsymbol{v}^\dagger\boldsymbol{x}}\Big)/\|\boldsymbol{v}\|^2$$

In order for $\hat{V}\boldsymbol{x}$ to be proportional to $\boldsymbol{e}_1$, we need to have $\mathrm{e}^{2\mathrm{i}\theta} = x_1/x_1^*$, *i.e.*, $\theta = \arg x_1$ or $\theta = \arg x_1 + \pi$.

Definition 6: A *Householder reflection* for vector $\boldsymbol{x}\in\mathbf{C}^m$ is one of the two unitary transformations $\hat{H}_\pm(\boldsymbol{x}) \coloneqq \hat{I}_m - 2\boldsymbol{v}_\pm\boldsymbol{v}_\pm^\dagger/\|\boldsymbol{v}_\pm\|^2$, where $\boldsymbol{v}_\pm \coloneqq \boldsymbol{x} \pm \big(x_1/|x_1|\big)\|\boldsymbol{x}\|\boldsymbol{e}_1$.[30] We have $\hat{H}_\pm(\boldsymbol{x})\boldsymbol{x} = \mp\big(x_1/|x_1|\big)\|\boldsymbol{x}\|\boldsymbol{e}_1$.

Householder QR factorization

for $i = 1, 2, ..., n$ do

  $\boldsymbol{x} \coloneqq \hat{A}_{i:m,i}$ — $(m-i+1)\times 1$ matrix or a vector

  $\boldsymbol{v}_i \coloneqq \boldsymbol{x} \pm \big(x_1/|x_1|\big)\|\boldsymbol{x}\|\boldsymbol{e}_1$ — $+$ sign is better for numerical stability

  $\boldsymbol{v}_i \coloneqq \boldsymbol{v}_i/\|\boldsymbol{v}_i\|_2$ — normalize, so we don't need to divide by $\|\boldsymbol{v}_i\|_2^2$ later

  $\hat{A}_{i:m,i:n} \coloneqq \hat{A}_{i:m,i:n} - 2\boldsymbol{v}_i(\boldsymbol{v}_i^\dagger\hat{A}_{i:m,i:n})$

return $\hat{A}, \boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_n$

The vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_n$ could be used to reconstruct the matrix $\hat{Q}$.

Example 6, continued: Let us now compute the QR factorization of $\hat{A}$ by Householder reflections. For the 1st column the Householder reflector H_- is formed from the vector $\boldsymbol{v}_- = \begin{bmatrix} 1 & \varepsilon & 0 & 0 \end{bmatrix}^{\mathrm{T}} - \sqrt{1+\varepsilon^2}\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 0 & \varepsilon & 0 & 0 \end{bmatrix}^{\mathrm{T}}$. Thus $\hat{H}_-(\boldsymbol{a}_1)$ reflection is just changing the sign of the 2nd component. We have $\hat{H}_-(\boldsymbol{a}_1)\boldsymbol{a}_1 = \begin{bmatrix} 1 & -\varepsilon & 0 & 0 \end{bmatrix}^{\mathrm{T}}$, *i.e.*, not all components below the 1st one become zero. That is because in our computation the small vector $\boldsymbol{v}_-$ is resulted in almost cancellation of two close vectors, so the direction of $\boldsymbol{v}_-$ suffers from large numerical errors.

[30] If $x_1 = 0$, then $x_1/|x_1|$ could be set to any number with absolute value 1 (*e.g.*, the number 1).

We have $\boldsymbol{v}_+ = \begin{bmatrix} 2 & \varepsilon & 0 & 0 \end{bmatrix}^{\mathrm{T}}$, and

$$\hat{H}_+(\boldsymbol{a}_1)\hat{A} = \left(\hat{I}_4 - 2\frac{\boldsymbol{v}_+\boldsymbol{v}_+^\dagger}{\|\boldsymbol{v}_+\|^2 = 4}\right)\hat{A} = \underbrace{\begin{bmatrix} -1 & -\varepsilon & 0 & 0 \\ -\varepsilon & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\hat{V}_1} \begin{bmatrix} 1 & 1 & 1 \\ \varepsilon & 0 & 0 \\ 0 & \varepsilon & 0 \\ 0 & 0 & \varepsilon \end{bmatrix} = \underbrace{\begin{bmatrix} -1 & -1 & -1 \\ 0 & -\varepsilon & -\varepsilon \\ 0 & \varepsilon & 0 \\ 0 & 0 & \varepsilon \end{bmatrix}}_{\hat{R}_1}$$

Then, for the 2nd column we have $\boldsymbol{x} = \begin{bmatrix} -\varepsilon & \varepsilon & 0 \end{bmatrix}^{\mathrm{T}}$, $\boldsymbol{v}_+ = \varepsilon\begin{bmatrix} -(\sqrt{2}+1) & 1 & 0 \end{bmatrix}^{\mathrm{T}}$, and

$$\hat{V}_2\hat{V}_1\hat{A} = \hat{V}_2\hat{R}_1 = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\hat{V}_2} \underbrace{\begin{bmatrix} -1 & -1 & -1 \\ 0 & -\varepsilon & -\varepsilon \\ 0 & \varepsilon & 0 \\ 0 & 0 & \varepsilon \end{bmatrix}}_{\hat{R}_1} = \underbrace{\begin{bmatrix} -1 & -1 & -1 \\ 0 & \sqrt{2}\varepsilon & \varepsilon/\sqrt{2} \\ 0 & 0 & -\varepsilon/\sqrt{2} \\ 0 & 0 & \varepsilon \end{bmatrix}}_{\hat{R}_2}$$

Finally, for the 3rd column we have $\boldsymbol{x} = \begin{bmatrix} -\varepsilon/\sqrt{2} & \varepsilon \end{bmatrix}^{\mathrm{T}}$, $\boldsymbol{v}_+ = \varepsilon\begin{bmatrix} -(\sqrt{3}+1)/\sqrt{2} & 1 \end{bmatrix}^{\mathrm{T}}$, and

$$\hat{V}_3\hat{V}_2\hat{V}_1\hat{A} = \hat{V}_3\hat{R}_2 = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1/\sqrt{3} & \sqrt{2/3} \\ 0 & 0 & \sqrt{2/3} & 1/\sqrt{3} \end{bmatrix}}_{\hat{V}_3} \underbrace{\begin{bmatrix} -1 & -1 & -1 \\ 0 & \sqrt{2}\varepsilon & \varepsilon/\sqrt{2} \\ 0 & 0 & -\varepsilon/\sqrt{2} \\ 0 & 0 & \varepsilon \end{bmatrix}}_{\hat{R}_2} = \underbrace{\begin{bmatrix} -1 & -1 & -1 \\ 0 & \sqrt{2}\varepsilon & \varepsilon/\sqrt{2} \\ 0 & 0 & \sqrt{3/2}\varepsilon \\ 0 & 0 & 0 \end{bmatrix}}_{\hat{R}}$$

6.3 Givens rotations

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $r = \sqrt{a^2+b^2}$, and $(c,s) = (a,b)/r$. We have $c = \cos\varphi$ and $s = \sin\varphi$, where $\varphi = \mathtt{atan2}(b,a)$.

Problems and exercises

1. Compute the QR factorization of $\hat{A}$ using both classical and modified Gram–Schmidt methods, and by Householder reflections. Assume that $\varepsilon^2 \ll \varepsilon_{\text{machine}} \ll \varepsilon$ in your calculations (*i.e.*, drop ε^2 terms when added to something of the order of 1).

$$\hat{A} = \begin{bmatrix} 3 & 3+4\varepsilon & 7 \\ 4 & 4-3\varepsilon & 1 \\ 5\varepsilon & -7\varepsilon & -12 \end{bmatrix}$$

2. Write programs that compute the QR factorization $\hat{A} = \hat{Q}\hat{R}$ of a matrix by classical and modified Gram–Schmidt methods, and by Householder reflections (with generation of the matrix $\hat{Q}$). Test them (how big is the residual $\hat{A} - \hat{Q}\hat{R}$, how close is $\hat{Q}^\dagger\hat{Q}$ to the identity matrix) on 10×10 Hilbert matrix $\hat{H}_{ij} = 1/(i+j-1)$.

3. Consider 1001×5 matrix $\hat{A}$ with $A_{ij} = x_i^{j-1}\exp(-x_i^2/2)$, where $x_i = 0.01(i-501)$, $1 \le i \le 1001$, $1 \le j \le 5$. Find the QR factorization $\hat{A} = \hat{Q}\hat{R}$ by Gram–Schmidt method, and plot the vectors $\boldsymbol{q}_j/\sqrt{0.01} = 10\boldsymbol{q}_j$, $1 \le j \le 5$ as functions of x_i.[31]

[31] The result is related to Hermite functions [and Hermite polynomials].

4. Consider Lie groups of all complex/real invertible $n \times n$ matrices $\mathrm{GL}(n,\mathbf{C})/\mathrm{GL}(n,\mathbf{R})$. They contain subgroups of all unitary/orthogonal matrices $\mathrm{U}(n)/\mathrm{O}(n)$ and all complex/real upper triangular matrices with strictly positive diagonal entries $\mathrm{T}_+(n,\mathbf{C})/\mathrm{T}_+(n,\mathbf{R})$. Find [real] dimensions of all the 6 mentioned Lie groups. Calculate $\dim \mathrm{GL}(n,\mathbf{C}) - \dim \mathrm{U}(n) - \dim \mathrm{T}_+(n,\mathbf{C})$ and $\dim \mathrm{GL}(n,\mathbf{R}) - \dim \mathrm{O}(n) - \dim \mathrm{T}_+(n,\mathbf{R})$.

7 Gaussian elimination, LU factorization

$$\boxed{\hat{I}_n}\ \boxed{\hat{A}} \longrightarrow \underbrace{\boxed{\hat{L}_{j-1}}\ \boxed{\hat{T}_j^{-1}}}_{\hat{L}_j}\ \underbrace{\boxed{\hat{T}_j}\ \boxed{\hat{U}_{j-1}}}_{\hat{U}_j} \longrightarrow \boxed{\hat{L}}\ \boxed{\hat{U}}$$

$$\hat{L} = \hat{T}_1^{-1}\hat{T}_2^{-1}\hat{T}_3^{-1}... \qquad \hat{U} = ...\hat{T}_3\hat{T}_2\hat{T}_1\hat{A}$$

The process is similar to QR factorization by Householder reflections, but instead of unitary transformation $\hat{V}_k$ the lower triangular $\hat{T}_j$ is employed here. Column by column of $\hat{A}$ we make its content below the main diagonal being 0. Here is an example of a pseudo-code that produces LU factorization:

```
for j = 1, 2, ..., n-1 do
    for i = j+1, j+2, ..., n do                       application of T̂_j
        l_ij := a_ij / a_jj
        for k = j, j+1, ..., n do                      row operation Â_{i,j:n} := Â_{i,j:n} - l_ij Â_{j,j:n}
            a_ik := â_ik - l_ij a_jk                   a_ij becomes a_ij - (l_ij = a_ij/a_jj) a_jj = 0
return L̂, Â
```

The matrices $\hat{T}_j$ and $\hat{T}_j^{-1}$ look like (empty spaces correspond to zero matrix elements)[32]

$$\hat{T}_j = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & -l_{j+1,j} & 1 & & \\ & & -l_{ij} & & 1 & \\ & & -l_{nj} & & & 1 \end{bmatrix}, \qquad \hat{T}_j^{-1} = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & l_{j+1,j} & 1 & & \\ & & l_{ij} & & 1 & \\ & & l_{nj} & & & 1 \end{bmatrix}$$

$$\hat{T}_j \begin{bmatrix} \cdot \\ \cdot \\ a_{jj} \\ a_{j+1,j} \\ a_{ij} \\ a_{nj} \end{bmatrix} = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & -l_{j+1,j} & 1 & & \\ & & -l_{ij} & & 1 & \\ & & -l_{nj} & & & 1 \end{bmatrix} \begin{bmatrix} \cdot \\ \cdot \\ a_{jj} \\ a_{j+1,j} \\ a_{ij} \\ a_{nj} \end{bmatrix} = \begin{bmatrix} \cdot \\ \cdot \\ a_{jj} \\ -(a_{j+1,j}/a_{jj})a_{jj} + a_{j+1,j} \\ -(a_{ij}/a_{jj})a_{jj} + a_{ij} \\ -(a_{nj}/a_{jj})a_{jj} + a_{nj} \end{bmatrix} = \begin{bmatrix} \cdot \\ \cdot \\ a_{jj} \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

[32] The simplest case to self-check the formula for $\hat{T}_j^{-1}$ is $\begin{bmatrix} 1 & 0 \\ l & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ -l & 1 \end{bmatrix} = \begin{bmatrix} 1\cdot 1 + 0\cdot(-l) & 1\cdot 0 + 0\cdot 1 \\ l\cdot 1 + 1\cdot(-l) & l\cdot 0 + 1\cdot 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.

$$\hat{L}_j = \underbrace{\begin{bmatrix} 1 & & & & & \\ l_{21} & 1 & & & & \\ l_{j1} & l_{j,j-1} & 1 & & & \\ l_{j+1,1} & l_{j+1,j-1} & & 1 & & \\ l_{i1} & l_{i,j-1} & & & 1 & \\ l_{n1} & l_{n,j-1} & & & & 1 \end{bmatrix}}_{\hat{L}_{j-1}=\hat{T}_1^{-1}\hat{T}_2^{-1}\ldots\hat{T}_{j-1}^{-1}} \underbrace{\begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & l_{j+1,j} & 1 & & \\ & & l_{ij} & & 1 & \\ & & l_{nj} & & & 1 \end{bmatrix}}_{\hat{T}_j^{-1}} = \underbrace{\begin{bmatrix} 1 & & & & & \\ l_{21} & 1 & & & & \\ l_{j1} & l_{j,j-1} & 1 & & & \\ l_{j+1,1} & l_{j+1,j-1} & l_{j+1,j} & 1 & & \\ l_{i1} & l_{i,j-1} & l_{ij} & & 1 & \\ l_{n1} & l_{n,j-1} & l_{nj} & & & 1 \end{bmatrix}}_{\hat{L}_j}$$

Example 7: Consider the matrix $\hat{A}$ and its LU factorization:[33]

$$\hat{A} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 4 & 1 & -5 & 0 \\ 3 & -4 & 0 & 1 \\ 1 & -2 & 4 & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & & & \\ 4 & 1 & & \\ 3 & & 1 & \\ 1 & & & 1 \end{bmatrix}}_{\hat{L}_1=\hat{T}_1^{-1}} \overbrace{\underbrace{\begin{bmatrix} 1 & & & \\ -4 & 1 & & \\ -3 & & 1 & \\ -1 & & & 1 \end{bmatrix}}_{\hat{T}_1} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 4 & 1 & -5 & 0 \\ 3 & -4 & 0 & 1 \\ 1 & -2 & 4 & 1 \end{bmatrix}}^{\hat{U}_1} =$$

$$= \begin{bmatrix} 1 & & & \\ 4 & 1 & & \\ 3 & & 1 & \\ 1 & & & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & -4 & 3 & 1 \\ 0 & -2 & 5 & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & & & \\ 4 & 1 & & \\ 3 & -4 & 1 & \\ 1 & -2 & & 1 \end{bmatrix}}_{\hat{L}_2=\hat{L}_1\hat{T}_2^{-1}} \overbrace{\underbrace{\begin{bmatrix} 1 & & & \\ & 1 & & \\ & 4 & 1 & \\ & 2 & & 1 \end{bmatrix}}_{\hat{T}_2} \begin{bmatrix} 1 & 0 & -1 & 0 \\ & 1 & -1 & 0 \\ & -4 & 3 & 1 \\ & -2 & 5 & 1 \end{bmatrix}}^{\hat{U}_2} =$$

$$= \begin{bmatrix} 1 & & & \\ 4 & 1 & & \\ 3 & -4 & 1 & \\ 1 & -2 & & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 & 0 \\ & 1 & -1 & 0 \\ & 0 & -1 & 1 \\ & 0 & 3 & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & & & \\ 4 & 1 & & \\ 3 & -4 & 1 & \\ 1 & -2 & -3 & 1 \end{bmatrix}}_{\hat{L}_3=\hat{L}_2\hat{T}_3^{-1}} \overbrace{\underbrace{\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & 3 & 1 \end{bmatrix}}_{\hat{T}_3} \begin{bmatrix} 1 & 0 & -1 & 0 \\ & 1 & -1 & 0 \\ & & -1 & 1 \\ & & 3 & 1 \end{bmatrix}}^{\hat{U}_3} =$$

$$= \begin{bmatrix} 1 & & & \\ 4 & 1 & & \\ 3 & -4 & 1 & \\ 1 & -2 & -3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 & 0 \\ & 1 & -1 & 0 \\ & & -1 & 1 \\ & & 0 & 4 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & & & \\ 4 & 1 & & \\ 3 & -4 & 1 & \\ 1 & -2 & -3 & 1 \end{bmatrix}}_{\hat{L}} \underbrace{\begin{bmatrix} 1 & 0 & -1 & 0 \\ & 1 & -1 & 0 \\ & & -1 & 1 \\ & & & 4 \end{bmatrix}}_{\hat{U}} = \hat{L}\hat{U}$$

[33] The matrix $\hat{A}$ is chosen in such a way that the matrices $\hat{L}$ and $\hat{U}$ end up being integer. Here $\kappa(\hat{A}) \approx 129.1$, $\kappa(\hat{L}) \approx 414.4$, and $\kappa(\hat{U}) \approx 8.18$.

LU factorization with partial pivoting:[34]

$$
\hat{A} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 4 & 1 & -5 & 0 \\ 3 & -4 & 0 & 1 \\ 1 & -2 & 4 & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} & 1 & & \\ 1 & & & \\ & & 1 & \\ & & & 1 \end{bmatrix}}_{\hat{P}_1} \begin{bmatrix} 4 & 1 & -5 & 0 \\ 1 & 0 & -1 & 0 \\ 3 & -4 & 0 & 1 \\ 1 & -2 & 4 & 1 \end{bmatrix} =
$$

$$
= \underbrace{\begin{bmatrix} & 1 & & \\ 1 & & & \\ & & 1 & \\ & & & 1 \end{bmatrix}}_{\hat{P}_1} \underbrace{\begin{bmatrix} 1 & & & \\ \frac{1}{4} & 1 & & \\ \frac{3}{4} & & 1 & \\ \frac{1}{4} & & & 1 \end{bmatrix}}_{\hat{L}_1 = \hat{T}_1^{-1}} \underbrace{\begin{bmatrix} 4 & 1 & -5 & 0 \\ 0 & -\frac{1}{4} & \frac{1}{4} & 0 \\ 0 & -\frac{19}{4} & \frac{15}{4} & 1 \\ 0 & -\frac{9}{4} & \frac{21}{4} & 1 \end{bmatrix}}_{\hat{U}_1} =
$$

$$
= \underbrace{\begin{bmatrix} & 1 & & \\ 1 & & & \\ & & 1 & \\ & & & 1 \end{bmatrix}}_{\hat{P}_1} \underbrace{\begin{bmatrix} 1 & & & \\ \frac{1}{4} & 1 & & \\ \frac{3}{4} & & 1 & \\ \frac{1}{4} & & & 1 \end{bmatrix}}_{\hat{L}_1 = \hat{T}_1^{-1}} \begin{bmatrix} 1 & & & \\ & & 1 & \\ & 1 & & \\ & & & 1 \end{bmatrix} \begin{bmatrix} 4 & 1 & -5 & 0 \\ & -\frac{19}{4} & \frac{15}{4} & 1 \\ & -\frac{1}{4} & \frac{1}{4} & 0 \\ & -\frac{9}{4} & \frac{21}{4} & 1 \end{bmatrix} =
$$

$$
= \underbrace{\begin{bmatrix} & & 1 & \\ 1 & & & \\ & 1 & & \\ & & & 1 \end{bmatrix}}_{\hat{P}_2} \begin{bmatrix} 1 & & & \\ \frac{3}{4} & 1 & & \\ \frac{1}{4} & & 1 & \\ \frac{1}{4} & & & 1 \end{bmatrix} \underbrace{\begin{bmatrix} 1 & & & \\ & 1 & & \\ & \frac{1}{19} & 1 & \\ & \frac{9}{19} & & 1 \end{bmatrix}}_{\hat{T}_2^{-1}} \underbrace{\begin{bmatrix} 4 & 1 & -5 & 0 \\ & -\frac{19}{4} & \frac{15}{4} & 1 \\ & 0 & \frac{1}{19} & -\frac{1}{19} \\ & 0 & \frac{66}{19} & \frac{10}{19} \end{bmatrix}}_{\hat{U}_2} =
$$

$$
= \begin{bmatrix} & & 1 & \\ 1 & & & \\ & 1 & & \\ & & & 1 \end{bmatrix} \underbrace{\begin{bmatrix} 1 & & & \\ \frac{3}{4} & 1 & & \\ \frac{1}{4} & \frac{1}{19} & 1 & \\ \frac{1}{4} & \frac{9}{19} & & 1 \end{bmatrix}}_{\hat{L}_2} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix} \begin{bmatrix} 4 & 1 & -5 & 0 \\ & -\frac{19}{4} & \frac{15}{4} & 1 \\ & & \frac{66}{19} & \frac{10}{19} \\ & & \frac{1}{19} & -\frac{1}{19} \end{bmatrix} =
$$

$$
= \underbrace{\begin{bmatrix} & & & 1 \\ 1 & & & \\ & 1 & & \\ & & 1 & \end{bmatrix}}_{\hat{P}_3} \begin{bmatrix} 1 & & & \\ \frac{3}{4} & 1 & & \\ \frac{1}{4} & \frac{9}{19} & 1 & \\ \frac{1}{4} & \frac{1}{19} & & 1 \end{bmatrix} \underbrace{\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & \frac{1}{66} & 1 \end{bmatrix}}_{\hat{T}_3^{-1}} \begin{bmatrix} 4 & 1 & -5 & 0 \\ & -\frac{19}{4} & \frac{15}{4} & 1 \\ & & \frac{66}{19} & \frac{10}{19} \\ & & 0 & -\frac{2}{33} \end{bmatrix} =
$$

$$
= \underbrace{\begin{bmatrix} & & & 1 \\ 1 & & & \\ & 1 & & \\ & & 1 & \end{bmatrix}}_{\hat{P}} \underbrace{\begin{bmatrix} 1 & & & \\ \frac{3}{4} & 1 & & \\ \frac{1}{4} & \frac{9}{19} & 1 & \\ \frac{1}{4} & \frac{1}{19} & \frac{1}{66} & 1 \end{bmatrix}}_{\hat{L}} \underbrace{\begin{bmatrix} 4 & 1 & -5 & 0 \\ & -\frac{19}{4} & \frac{15}{4} & 1 \\ & & \frac{66}{19} & \frac{10}{19} \\ & & & -\frac{2}{33} \end{bmatrix}}_{\hat{U}}
$$

[34] We get $\kappa(\hat{L}) \approx 2.46$ and $\kappa(\hat{U}) \approx 144.8$.

LU factorization with complete pivoting:[35]

$$\hat{A}=\begin{bmatrix}1&0&-1&0\\4&1&-5&0\\3&-4&0&1\\1&-2&4&1\end{bmatrix}=\underbrace{\begin{bmatrix}&1&&\\1&&&\\&&1&\\&&&1\end{bmatrix}}_{\hat{P}_1}\begin{bmatrix}-5&1&4&0\\-1&0&1&0\\0&-4&3&1\\4&-2&1&1\end{bmatrix}\underbrace{\begin{bmatrix}&&1&\\&1&&\\1&&&\\&&&1\end{bmatrix}}_{\hat{Q}_1}=$$

$$=\underbrace{\begin{bmatrix}&1&&\\1&&&\\&&1&\\&&&1\end{bmatrix}}_{\hat{P}_1}\underbrace{\begin{bmatrix}1&&&\\\frac{1}{5}&1&&\\0&&1&\\-\frac{4}{5}&&&1\end{bmatrix}}_{\hat{L}_1=\hat{T}_1^{-1}}\underbrace{\begin{bmatrix}-5&1&4&0\\0&-\frac{1}{5}&\frac{1}{5}&0\\0&-4&3&1\\0&-\frac{6}{5}&\frac{21}{5}&1\end{bmatrix}}_{\hat{U}_1}\underbrace{\begin{bmatrix}&&1&\\&1&&\\1&&&\\&&&1\end{bmatrix}}_{\hat{Q}_1}=$$

$$=\underbrace{\begin{bmatrix}&1&&\\1&&&\\&&1&\\&&&1\end{bmatrix}}_{\hat{P}_1}\underbrace{\begin{bmatrix}1&&&\\\frac{1}{5}&1&&\\0&&1&\\-\frac{4}{5}&&&1\end{bmatrix}}_{\hat{L}_1=\hat{T}_1^{-1}}\begin{bmatrix}1&&&\\&&&1\\&&1&\\&1&&\end{bmatrix}\begin{bmatrix}-5&4&1&0\\&\frac{21}{5}&-\frac{6}{5}&1\\&3&-4&1\\&\frac{1}{5}&-\frac{1}{5}&0\end{bmatrix}\begin{bmatrix}1&&&\\&&1&\\&1&&\\&&&1\end{bmatrix}\underbrace{\begin{bmatrix}&&1&\\&1&&\\1&&&\\&&&1\end{bmatrix}}_{\hat{Q}_1}=$$

$$=\underbrace{\begin{bmatrix}&&&1\\1&&&\\&&1&\\&1&&\end{bmatrix}}_{\hat{P}_2}\begin{bmatrix}1&&&\\-\frac{4}{5}&1&&\\0&&1&\\\frac{1}{5}&&&1\end{bmatrix}\begin{bmatrix}-5&4&1&0\\&\frac{21}{5}&-\frac{6}{5}&1\\&3&-4&1\\&\frac{1}{5}&-\frac{1}{5}&0\end{bmatrix}\underbrace{\begin{bmatrix}&&1&\\1&&&\\&1&&\\&&&1\end{bmatrix}}_{\hat{Q}_2}=$$

$$=\underbrace{\begin{bmatrix}&&&1\\1&&&\\&&1&\\&1&&\end{bmatrix}}_{\hat{P}_2}\underbrace{\begin{bmatrix}1&&&\\-\frac{4}{5}&1&&\\0&\frac{5}{7}&1&\\\frac{1}{5}&\frac{1}{21}&&1\end{bmatrix}}_{\hat{L}_2}\underbrace{\begin{bmatrix}-5&4&1&0\\&\frac{21}{5}&-\frac{6}{5}&1\\&0&-\frac{22}{7}&\frac{2}{7}\\&0&-\frac{1}{7}&-\frac{1}{21}\end{bmatrix}}_{\hat{U}_2}\underbrace{\begin{bmatrix}&&1&\\1&&&\\&1&&\\&&&1\end{bmatrix}}_{\hat{Q}_2}=$$

$$=\underbrace{\begin{bmatrix}&&&1\\1&&&\\&&1&\\&1&&\end{bmatrix}}_{\hat{P}=\hat{P}_2}\underbrace{\begin{bmatrix}1&&&\\-\frac{4}{5}&1&&\\0&\frac{5}{7}&1&\\\frac{1}{5}&\frac{1}{21}&\frac{1}{22}&1\end{bmatrix}}_{\hat{L}}\underbrace{\begin{bmatrix}-5&4&1&0\\&\frac{21}{5}&-\frac{6}{5}&1\\&&-\frac{22}{7}&\frac{2}{7}\\&&0&-\frac{2}{33}\end{bmatrix}}_{\hat{U}}\underbrace{\begin{bmatrix}&&1&\\1&&&\\&1&&\\&&&1\end{bmatrix}}_{\hat{Q}=\hat{Q}_2}=$$

Problems and exercises

1. Write a program that solves the square system $\hat{A}\boldsymbol{x}=\boldsymbol{b}$ by Gaussian elimination with partial pivoting. Test it on 10×10 Hilbert matrix $\hat{H}_{ij}=1/(i+j-1)$: First compute the r.h.s. vector $\boldsymbol{b}=\hat{H}\begin{bmatrix}1&2&3&4&5&6&7&8&9&10\end{bmatrix}^{\mathrm{T}}$, and then solve the system $\hat{H}\boldsymbol{x}=\boldsymbol{b}$.

[35] We get $\kappa(\hat{L})\approx 2.92$ and $\kappa(\hat{U})\approx 121.2$.

8 Eigenvalues

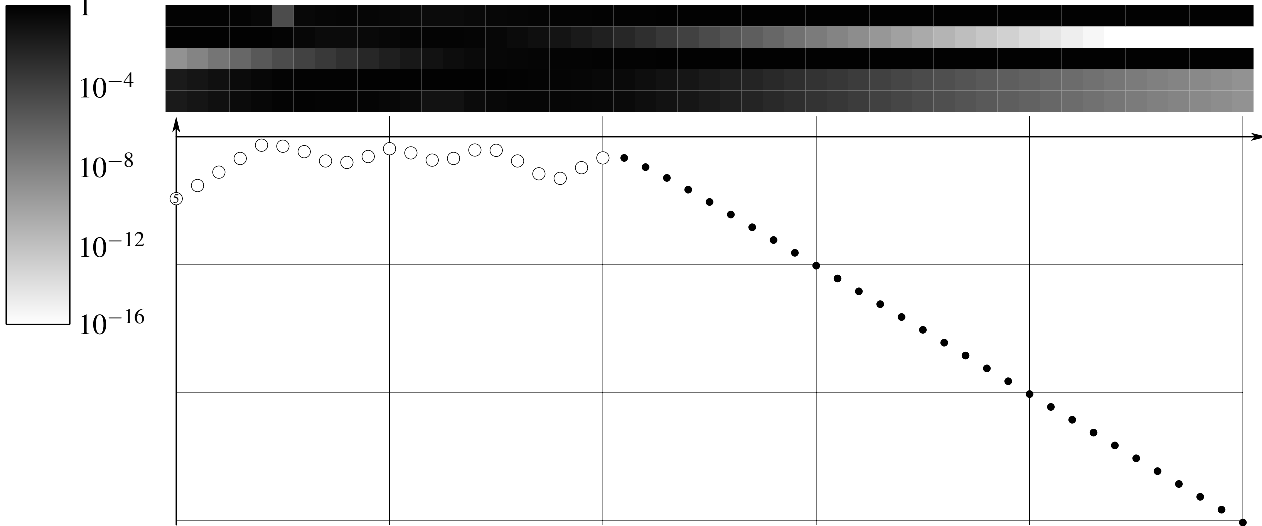
Algorithm $\det(\lambda\hat{I}-\hat{A})=0$: Calculate the characteristic polynomial of the matrix, then find its roots. This is an unstable algorithm.

Algorithm "power iteration": Choose arbitrarily a vector $\boldsymbol{x}_0$, then calculate $\boldsymbol{x}_n := \hat{A}\boldsymbol{x}_{n-1}$ for $n=1$, 2, ... Renormalize the vector $\boldsymbol{x}_n$ when needed. This method finds an eigenvector for the eigenvalue with the largest absolute value.

Exapmpe 8: Consider the following matrix:

$$\hat{A}=\begin{bmatrix} 8 & 7 & -8 & -7 & -1 \\ 6 & 7 & 6 & -3 & 3 \\ -8 & -8 & 8 & 8 & 0 \\ -2 & -2 & -2 & 6 & 2 \\ -6 & -6 & -6 & 6 & 2 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 1 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 1 \\ -1 & -1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & -1 & 0 \end{bmatrix}}_{\hat{V}} \underbrace{\begin{bmatrix} 16 & & & & \\ & 8 & & & \\ & & 4 & & \\ & & & 2 & \\ & & & & 1 \end{bmatrix}}_{\hat{D}} \underbrace{\begin{bmatrix} 1 & 1 & 0 & -1 & 0 \\ -1 & -1 & -1 & 1 & 0 \\ 2 & 2 & 2 & -1 & 1 \\ -1 & -1 & -1 & 1 & -1 \\ -2 & -1 & -2 & 1 & -1 \end{bmatrix}}_{\hat{V}^{-1}}$$

The columns of $\hat{V}$ are the eigenvectors of $\hat{A}$: $\hat{A}\boldsymbol{v}_j = 2^{5-j}\boldsymbol{v}_j$, $j=1, 2, 3, 4, 5$. Let us choose $\boldsymbol{x}_0 := \boldsymbol{v}_1 + 2^{16}\boldsymbol{v}_2 + 2^{28}\boldsymbol{v}_3 + 2^{36}\boldsymbol{v}_2 + 2^{40}\boldsymbol{v}_5$, normalize it $\boldsymbol{x}_0 := \boldsymbol{x}_0/\|\boldsymbol{x}_0\|_2$, and then do power iterations $\boldsymbol{x}_n = \hat{A}\boldsymbol{x}_{n-1}$, $\boldsymbol{x}_n := \boldsymbol{x}_n/\|\boldsymbol{x}_n\|_2$ for $n = 1, 2, 3, ..., 50$:



The graph shows the smallest angle to one the eigenvalues $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, $\boldsymbol{v}_3$, $\boldsymbol{v}_4$, $\boldsymbol{v}_5$. Here the vector $\boldsymbol{x}_0$ is chosen in such a way, that each eigenvector is dominant at some iteration. Eventually the iterations converge to $\boldsymbol{v}_1$, the eigenvector with the largest in absolute value eigenvalue.

Algorithm "inverse iteration": Choose a number μ. Do power iteration for $(\hat{A}-\mu\hat{I})^{-1}$. This method finds an eigenvector for the eigenvalue closest to μ.

Definition 8.1: Let $\hat{A}$ be an $n\times n$ matrix. The *Rayleigh quotient* of a vector $\boldsymbol{x}$ is the $R(\hat{A},\boldsymbol{x}) := \big(\boldsymbol{x}^\dagger\hat{A}\boldsymbol{x}\big)/(\boldsymbol{x}^\dagger\boldsymbol{x})$. If $\boldsymbol{x}$ is an eigenvector of the matrix $\hat{A}$ with eigenvalue λ, then $R(\hat{A},\boldsymbol{x})=\lambda$.

Algorithm "Rayleigh quotient iteration": Choose arbitrarily a vector $\boldsymbol{x}_0$, then calculate $\boldsymbol{x}_n := \big(\hat{A} - R(\hat{A},\boldsymbol{x}_{n-1})\hat{I}\big)^{-1}\boldsymbol{x}_{n-1}$ for $n=1$, 2, ... Renormalize the vector $\boldsymbol{x}_n$ when needed.

Definition/Theorem 8.2: Any square $n\times n$ matrix $\hat{A}$ has a *Schur decomposition* $\hat{A}=\hat{Q}\hat{T}\hat{Q}^\dagger$, where matrix $\hat{Q}$ is unitary, while matrix $\hat{T}$ is upper triangular. This is a similarity transformation, and diagonal elements of $\hat{T}$ are the eigenvalues of $\hat{A}$.

Definition 8.3: A *Hessenberg decomposition* of a square $n \times n$ matrix $\hat{A}$ is $\hat{A} = \hat{Q}\hat{H}\hat{Q}^\dagger$, where matrix $\hat{Q}$ is unitary, and matrix $\hat{H}$ is such that $H_{ij} = 0$ if $i > j+1$. This is a similarity transformation, and the eigenvalues of $\hat{A}$ and of $\hat{H}$ are the same.

The Hessenberg form of a matrix $\hat{A}$ can be ontained with Householder reflections $O(n^3)$ operations. The algorithm is simialar to QR factorization, but one leaves one more component non-zero. This allows the zeros of the formed matrix being not destroyed by multiplying by $\hat{V}^\dagger$ from the right.

Problems and exercises

1. Consider the 100×100 matrix $\hat{A}$ with $a_{ii} = -2$, and $a_{i+1,i} = a_{i-1,i} = 1$ for all $i = 1, 2, \ldots 100$. The values $0/101$ of the index are identified with $100/1$. The matrix is cyclic shift invariant ($a_{ij} = a_{i+k,j+k}$), so discrete Fourier transform diagonalizes it. Consider $\boldsymbol{x}_0 = \boldsymbol{e}_{50}$. (a) Do power iterations $\boldsymbol{x}_n := \hat{A}\boldsymbol{x}_{n-1}$, $\boldsymbol{x}_n := \boldsymbol{x}_n/\|\boldsymbol{x}_n\|_2$. Plot $\boldsymbol{x}_{100}$, $\boldsymbol{x}_{1000}$, and $\boldsymbol{x}_{10000}$. Guess the maximal in absolute value eigenvalue $\lambda_{\max}$ and the eigenvector $\boldsymbol{x}_{\max}$ corresponding to it. (b) Let $\mu = -4.001$. Plot $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{x}_3$, and $\boldsymbol{x}_4$ for the inverse power iteration $\boldsymbol{x}_n := (\hat{A} - \mu\hat{I})^{-1}\boldsymbol{x}_{n-1}$, $\boldsymbol{x}_n := \boldsymbol{x}_n/\|\boldsymbol{x}_n\|_2$ ($\boldsymbol{x}_n$ is found as the solution of the system $(\hat{A} - \mu\hat{I})\boldsymbol{x}_n = \boldsymbol{x}_{n-1}$). (c) Do Rayleigh quotient iterations $\mu_0 = -4.001$, $\boldsymbol{x}_n := (\hat{A} - \mu_{n-1}\hat{I})^{-1}\boldsymbol{x}_{n-1}$, $\boldsymbol{x}_n := \boldsymbol{x}_n/\|\boldsymbol{x}_n\|_2$, $\mu_n := R(\hat{A}, \boldsymbol{x}_n) = \boldsymbol{x}_n^\dagger \hat{A}\boldsymbol{x}_n$ up to $n = 5$. Print μ_n and $\|\boldsymbol{x}_n \pm \boldsymbol{x}_{n-1}\|_2$, $n = 1, 2, \ldots, 5$.

2. Consider the matrix $\hat{A}$ on the right. Its eigenvalues are complex. (a) Do 100 and 101 QR iterations $\hat{Q}\hat{R} := \hat{A}$, $\hat{A} := \hat{R}\hat{Q}$. Can you easily extract eigenvalues of $\hat{A}$, *e.g.*, from $\hat{A}_{1:2,1:2}$? (b) Do 100 shifted QR iterations $\hat{Q}\hat{R} := \hat{A} - \hat{I}_4$, $\hat{A} := \hat{R}\hat{Q} + \hat{I}_4$. Calculate the eigenvalues of the upper left $\hat{A}_{1:2,1:2}$ and lower right $\hat{A}_{3:4,3:4}$ 2×2 corners of the resulted matrix, compare them with the eigenvalues of $\hat{A}$. (c) Do 100 shifted QR iterations $\hat{Q}\hat{R} := \hat{A} - \mu\hat{I}_4$, $\hat{A} := \hat{R}\hat{Q} + \mu\hat{I}_4$ with $\mu = (1+\mathrm{i})/2$. Is $\hat{A}$ close to being an upper triangular?

$$\hat{A} = \begin{bmatrix} 30000 & -29999 & -29999 & 30000 \\ 30001 & -30000 & -30000 & 30001 \\ 9999 & -10000 & 30000 & -29999 \\ 10000 & -10001 & 30001 & -30000 \end{bmatrix}$$

Part III
Systems of nonlinear equations

Consider you are to solve the equation $f(x) = 0$, where f is the continuous real function of one real variable. This problem could be solved by simple but powerful *bisection method*. The idea is the following: If you find such $x_{\text{left}} < x_{\text{right}}$ that $f(x_{\text{left}})$ and $f(x_{\text{right}})$ are of different sign, then there is such x_*, $x_{\text{left}} < x_* < x_{\text{right}}$, that $f(x_*) = 0$. Try $x = (x_{\text{left}} + x_{\text{right}})/2$. If $f(x) = 0$, then you solved the equation. Otherwise check which $f(x_{\text{left}})$ or $f(x_{\text{right}})$ is of different sign with $f(x)$ and narrow the interval [inside which at least one solution lies in] $(x_{\text{left}}, x_{\text{right}})$ to either (x_{left}, x) or (x, x_{right}). We can systematically reduce, each time by factor or 2, the width of the interval containing a solution, until the width of the interval [or uncertainty in solution] is small enough.

Here is how the solution of $x = \cos x$ equation inside the interval $[0, 1]$ is found (we define $f(x) = x - \cos(x)$, we have $f(0) = -1 < 0$ and $f(1) = 1 - \cos(1) > 0$):

```
[...]/teaching/2019-4/math_575a/notes/Python$ cat x_eq_cos_x.py
from math import cos
xl, xr = 0., 1.
print('{0:.8e}                    {1:.8e}'.format(xl, xr))
while (xr - xl > 1.e-6):
```

```
    xm = 0.5 * (xl + xr)
    if (cos(xm) > xm):
        print('               {0:.15e} {1:.8e}'.format(xm, xr))
        xl = xm
    else:
        print('{0:.8e} {1:.15e}'.format(xl, xm))
        xr = xm
[...]/teaching/2019-4/math_575a/notes/Python$ python3 x_eq_cos_x.py
0.00000000e+00                       1.00000000e+00
               5.000000000000000e-01 1.00000000e+00
5.00000000e-01 7.500000000000000e-01
               6.250000000000000e-01 7.50000000e-01
               6.875000000000000e-01 7.50000000e-01
               7.187500000000000e-01 7.50000000e-01
               7.343750000000000e-01 7.50000000e-01
7.34375000e-01 7.421875000000000e-01
               7.382812500000000e-01 7.42187500e-01
7.38281250e-01 7.402343750000000e-01
7.38281250e-01 7.392578125000000e-01
               7.387695312500000e-01 7.39257812e-01
               7.390136718750000e-01 7.39257812e-01
7.39013672e-01 7.391357421875000e-01
               7.390747070312500e-01 7.39135742e-01
7.39074707e-01 7.391052246093750e-01
7.39074707e-01 7.390899658203125e-01
               7.390823364257812e-01 7.39089966e-01
7.39082336e-01 7.390861511230469e-01
               7.390842437744141e-01 7.39086151e-01
7.39084244e-01 7.390851974487305e-01
[...]/teaching/2019-4/math_575a/notes/Python$
```

9 Functional iteration

Quite often a general system of non-linear equations arises (or can be rewritten) in the form $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$, where $\boldsymbol{f}(\cdot)$ is a n-component vector function, with an n-component vector as an argument.

A common method of solving such a system is by iterations $\boldsymbol{x}^{(n+1)} := \boldsymbol{f}\left(\boldsymbol{x}^{(n)}\right)$ starting from some initial guess $\boldsymbol{x}^{(0)}$. If such iterations do converge, then they converge to a solution. Here is how the equation $x = \cos(x)$ is solved by functional iteration, starting from initial guess $x^{(0)} = 0$:

```
[...]/teaching/2019-4/math_575a/notes/C$ cat x_eq_cos_x.c
#include <stdio.h>
#include <math.h>
int main() { int i; double x;
  for (x = 0., i = 0; i <= 34; i++, x = cos(x)) printf("%8.6f ", x);
  printf("\n"); return 0; }
[...]/teaching/2019-4/math_575a/notes/C$ cc x_eq_cos_x.c -lm
[...]/teaching/2019-4/math_575a/notes/C$ ./a.out
0.000000 1.000000 0.540302 0.857553 0.654290 0.793480 0.701369 0.763960 0.722102
 0.750418 0.731404 0.744237 0.735605 0.741425 0.737507 0.740147 0.738369 0.73956
7 0.738760 0.739304 0.738938 0.739184 0.739018 0.739130 0.739055 0.739106 0.7390
71 0.739094 0.739079 0.739089 0.739082 0.739087 0.739084 0.739086 0.739085
[...]/teaching/2019-4/math_575a/notes/C$
```

```
octave:1> format long
octave:2> options = optimset('TolX', 1.e-13, 'TolFun', 1.e-13);
octave:3> x = fsolve(@(x) (x - cos(x)), 0., options)
x =    7.390851332151714e-01
octave:4> y = cos(x)
y =    7.390851332151533e-01
```

The stability of the functional iterations at the solution $\boldsymbol{x}_* = \boldsymbol{f}(\boldsymbol{x}_*)$ could be obtained from the linearization of $\boldsymbol{f}$ at $\boldsymbol{x} = \boldsymbol{x}_*$.

10 Newton–Raphson method

Let us try to solve $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$ system of equations. It is complicated as the vector function $\boldsymbol{f}$ is non-linear. Consider we have a guess for a solution, $\boldsymbol{x}_0$. We can approximate $\boldsymbol{f}$ by its tangent line approximation: $\boldsymbol{f}(\boldsymbol{x} = \boldsymbol{x}_0 + \boldsymbol{\delta}) \approx \boldsymbol{f}(\boldsymbol{x}_0) + (\nabla \boldsymbol{f})(\boldsymbol{x}_0) \cdot \boldsymbol{\delta}$. The truncation of the Taylor series in $\boldsymbol{\delta}$ is motivated by our expectation that $\boldsymbol{\delta}$ is small. Our system of equations becomes $\boldsymbol{f}(\boldsymbol{x}) \approx \boldsymbol{f}(\boldsymbol{x}_0) + (\nabla \boldsymbol{f})(\boldsymbol{x}_0) \cdot \boldsymbol{\delta} = \boldsymbol{0}$, which is a system of linear equations for the components of vector $\boldsymbol{\delta}$. We have

$$\boldsymbol{x} = \boldsymbol{x}_0 + \boldsymbol{\delta} \approx \boldsymbol{x}_0 + \big(\text{solution of the } (\nabla \boldsymbol{f})(\boldsymbol{x}_0) \cdot \boldsymbol{\delta} = -\boldsymbol{f}(\boldsymbol{x}_0) \text{ system}\big) = \boldsymbol{x}_0 - \big((\nabla \boldsymbol{f})(\boldsymbol{x}_0)\big)^{-1} \boldsymbol{f}(\boldsymbol{x}_0)$$

The method of solving the system of equations $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$ based on iterations $\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \big((\nabla \boldsymbol{f})(\boldsymbol{x}_n)\big)^{-1} \boldsymbol{f}(\boldsymbol{x}_n)$ is called the Newton–Raphson method.

There is no guarantee that these iterations are going to converge, but when they do they converge very fast: $\boldsymbol{f}(\boldsymbol{x}_{n+1}) = \boldsymbol{f}\Big(\boldsymbol{x}_n - \big((\nabla \boldsymbol{f})(\boldsymbol{x}_n)\big)^{-1} \boldsymbol{f}(\boldsymbol{x}_n)\Big) = \boldsymbol{f}(\boldsymbol{x}_n) - (\nabla \boldsymbol{f})(\boldsymbol{x}_n) \cdot \big((\nabla \boldsymbol{f})(\boldsymbol{x}_n)\big)^{-1} \boldsymbol{f}(\boldsymbol{x}_n) + \frac{1}{2} (\nabla\nabla \boldsymbol{f})(\boldsymbol{x}_n) \cdot \big(\big((\nabla \boldsymbol{f})(\boldsymbol{x}_n)\big)^{-1} \boldsymbol{f}(\boldsymbol{x}_n)\big)^2 \propto (\boldsymbol{f}(\boldsymbol{x}_n))^2$. The "error" in the next step is the square of the error in previous step.

Example 10.1: Let us compute $\sqrt{2}$. We may construct a function $f(x)$ such that $f(\sqrt{2}) = 0$, and then find the root by the Newton–Raphson method. Let $f(x) := x^2 - 2$. Then $f'(x) = 2x$, and the update rule reads as $x_{n+1} = x_n - (x_n^2 - 2)/2x_n = x_n/2 + 1/x_n$. If we start from $x_0 = 1$ or $x_0 = 2$, we have $x_1 = 1/2 + 1/1 = 2/2 + 1/2 = 3/2$. Then $x_2 = x_1/2 + 1/x_1 = 3/4 + 2/3 = (9+8)/12 = 17/12 = 1.41666...$ (notice that $17^2 = 289 \approx 288 = 2 \cdot 12^2$). We have $x_3 = 17/24 + 12/17 = (17^2 + 24 \cdot 12)/24 \cdot 17 = 577/408 = 1.41421568...$ (notice that $577^2 = 332929 \approx 332928 = 2 \cdot 408^2$). Next $x_4 = 665857/470832 = 1.41421356237468...$, while $\sqrt{2} = 1.41421356237309...$ The Newton–Raphson iterations very quickly converge to $\sqrt{2}$, at each iteration the number of correct significant digits is doubled.

Example 10.2: Consider the system $y = x^2$, $xy = 1$. We can write it as $\boldsymbol{f}(x,y)$ by setting $f_1(x,y) = y - x^2$ and $f_2(x,y) = xy - 1$. The matrix $\nabla \boldsymbol{f}$ and the iterations look like

$$\begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix} = \begin{bmatrix} -2x & 1 \\ y & x \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} \mapsto \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} -2x & 1 \\ y & x \end{bmatrix}^{-1} \begin{bmatrix} y - x^2 \\ xy - 1 \end{bmatrix} = \begin{bmatrix} y - x^2 \\ xy - 1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \frac{1}{2x^2 + y} \begin{bmatrix} x & -1 \\ -y & -2x \end{bmatrix} \begin{bmatrix} y - x^2 \\ xy - 1 \end{bmatrix} =$$
$$= \frac{1}{2x^2 + y} \begin{bmatrix} 1 + xy + x^3 \\ x(2 + xy) \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \frac{\alpha}{3 + 4\alpha + \beta + 2\alpha^2} \begin{bmatrix} \beta + \alpha + \alpha^2 \\ 2\beta - \alpha + \alpha\beta \end{bmatrix}$$

where $x = 1 + \alpha$, $y = 1 + \beta$. We have $x = y = 1$ being a solution, with the deviation from it being about squared in magnitude with each iteration:

```
octave:1> format long
octave:2> f = @(x) ([1 + x(1) * x(2) + (x(1))^3, x(1) * (2 + x(1) * x(2))] / (2
* (x(1))^2 + x(2)));
octave:3> x0 = [2, 0];
octave:4> [x0 ; f(x0); f(f(x0)); f(f(f(x0))); f(f(f(f(x0)))); f(f(f(f(f(x0)))))]
ans =

   2.000000000000000e+00   0.000000000000000e+00
   1.125000000000000e+00   5.000000000000000e-01
   9.851804123711341e-01   9.510309278350515e-01
   1.000325727687672e+00   1.000422180897834e+00
   1.000000081169684e+00   1.000000056293719e+00
   1.000000000000004e+00   1.000000000000001e+00
```

Problems and exercises

1. Consider the system $y = x^2$, $xy = 1$ (Example 10.2). Find all of its solutions analytically. Do several Newton-Raphson iterations starting from $x_0 = -1 + \mathrm{i}$, $y_0 = 0$. Do you converge to a solution, and if yes, to which one?

2. Consider the [transcendental] equation $\mathrm{e}^x = kx$. When $k > \mathrm{e}$, there are two solutions, $x_{\mathrm{small}}(k) < 1$ and $x(k) > 1$. For $\mathrm{e}^2 \leq k \leq \mathrm{e}^{10}$ log-log plot the solution $x(k)$ found by (a) bisection method, (b) functional iteration (you need to rewrite the equation in $x = F(x)$ form with iterations being converging), and (c) Newton–Raphson method.

Part IV
Numerical ODEs

Suggested reading: [AsPe98].

11 Interpolation, basic integration schemes

Consider you are to compute $I = \int_a^b \mathrm{d}x\, f(x)$. Here we think of generic (*i.e.*, not specified) function $f(x)$. We would like to 1) compute I accurately enough, and 2) spend less of an effort (which we will measure in at how many points the function $f(\cdot)$ is computed).[36] In a general recipe, where $f(x)$ is not specified, [due to linearity of integration] an algorithm of computing I

Example 11: Let us compute $\int_0^{\pi/2} \mathrm{d}x\, \sin(x)$. We will do it in several ways:
(a) Analytical: $\int_0^{\pi/2} \mathrm{d}x\, \sin(x) = -\cos(x)\big|_0^{\pi/2} = \cos(0) - \cos(\pi/2) = 1$. This is the exact answer.
(b) Taylor series: $\displaystyle\int_0^{\pi/2} \mathrm{d}x\, \sin(x) = \int_0^{\pi/2} \mathrm{d}x \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} = \sum_{n=0}^{\infty} \frac{(-1)^n (\pi/2)^{2n+2}}{(2n+2)!}$. Let us calculate this series numerically:

[36] It is not necessary that the integral is estimated through values of $f(\cdot)$ at some points. Possible situations could be using an analytical formula for I, or presenting f as a linear combination of functions

```
[...]/teaching/2019-4/math_575a/notes/Python$ cat int_sin_b.py
from math import pi
taylor, sum, n, pi2f = 1., 0., 0, -1.
while taylor != sum:
    pi2f = -pi2f * (pi / 2.)**2 / ((2. * n + 1.) * 2. * (n + 1.))
    taylor, sum, n = sum, sum + pi2f, n + 1
print('series = {0:.15e}, {1:d} terms are summed'.format(taylor, n))
[...]/teaching/2019-4/math_575a/notes/Python$ python3 int_sin_b.py
series = 9.999999999999999e-01, 11 terms are summed
[...]/teaching/2019-4/math_575a/notes/Python$
```

(c) Midpoint rule: for large N we have $\int_0^{\pi/2} \mathrm{d}x\, \sin(x) \approx \underbrace{\frac{\pi/2}{N}}_{\Delta x} \sum_{i=0}^{N-1} \sin\Big(\underbrace{\frac{\pi(2i+1)}{4N}}_{x_i}\Big)$:

```
[...]/teaching/2019-4/math_575a/notes/Python$ cat int_sin_c.py
from math import pi, sin
for N in [10, 100, 1000]:
    S = sum(map(sin, [(pi / 2.) * (i + 0.5) / N for i in range(0, N)]))
    print('N = {0:4d}, midpoint = {1:.15e}'.format(N, (pi / 2.) * S / N))
[...]/teaching/2019-4/math_575a/notes/Python$ python3 int_sin_c.py
N =   10, midpoint = 1.001028824142709e+00
N =  100, midpoint = 1.000010280911905e+00
N = 1000, midpoint = 1.000000102808387e+00
[...]/teaching/2019-4/math_575a/notes/Python$
```

(d) As in (c), but Simpson's rule is used: here N is necessarily even, $\Delta x := (b-a)/N$, $x_j := a + j\Delta x$ (so $x_0 = a$ and $x_N = b$), and $\int_a^b \mathrm{d}x\, f(x) \approx \frac{\Delta x}{3} \sum_{j=0}^{N/2-1} \Big(f(x_j) + 4f(x_{j+1}) + f(x_{j+2})\Big) = \Delta x\big(f(a) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \ldots + 2f(x_{N-2}) + 4f(x_{N-1}) + f(b)\big)/3$:

```
[...]/teaching/2019-4/math_575a/notes/Python$ cat int_sin_d.py
from math import pi, sin
for N in [10, 100, 1000]:
    S, w = sin(0.) + sin(pi / 2.), 4.
    for i in range(1, N):
        S, w = S + w * sin((pi / 2.) * i / N), 6. - w
    print('N = {0:4d}, Simpson = {1:.15e}'.format(N, pi * S / (6. * N)))
[...]/teaching/2019-4/math_575a/notes/Python$ python3 int_sin_d.py
N =   10, Simpson = 1.000003392220900e+00
N =  100, Simpson = 1.000000000338236e+00
N = 1000, Simpson = 1.000000000000033e+00
[...]/teaching/2019-4/math_575a/notes/Python$
```

[37]

Problems and exercises

[37] Chebfun — computational software using Chebyshev nodes.

1. Consider the integral $1 = \frac{1}{e-1}\int_0^1 dx\, \exp(x)$. Compute it by left-, right-sum, trapezoidal, mid-point, and Simpson's rules. Log-log plot errors vs. Δx, find the order of accuracy of these numerical integration schemes.

2. (a) Compute the integral $\int_0^1 dx\, \left(1-x^2\right)^2$ using trapezoidal rule. (b) Compute the integral $\int_0^1 \frac{dx}{1+x^2}$ using Simpson's rule. In both (a) and (b) log-log plot errors vs. Δx, speculate about the order of accuracy.

3. Compute $\int_0^1 \frac{dx}{\sqrt{x+x^3}}$ up to 10 significant digits.

12 Euler method, stability

The solution of the system of ODEs $d\boldsymbol{x}/dt = \boldsymbol{f}\big(t,\boldsymbol{x}(t)\big)$ can be written as[38]

$$\boldsymbol{x}(t_{\text{end}}) = \boldsymbol{x}(t_{\text{start}}) + \int_{t_{\text{start}}}^{t_{\text{end}}} dt\, \underbrace{\boldsymbol{f}\big(t,\boldsymbol{x}(t)\big)}_{d\boldsymbol{x}/dt}$$

We assume $\boldsymbol{x}(t_{\text{start}})$ to be known, and our task is to find $\boldsymbol{x}(t_{\text{end}})$ or the whole trajectory $\boldsymbol{x}(t)$. We divide the interval of integration $[t_{\text{start}},t_{\text{end}}]$ into N subintervals

$$t_{\text{start}} = t_0 < t_1 < t_2 < \ldots < t_{N-1} < t_N = t_{\text{end}}, \qquad n^{\text{th}} \text{ step size } h_n = t_{n-1} - t_n$$

and approximate the integral of the r.h.s. $\boldsymbol{f}$ over $[t_{n-1},t_n]$ using some integration scheme. We will denote $\boldsymbol{x}(t_n)$ as $\boldsymbol{x}_n$. Estimating the integral of $\boldsymbol{f}$ over $[t_0,t_1]$ would give us the difference between $\boldsymbol{x}_1$ and $\boldsymbol{x}_0$, and (as $\boldsymbol{x}_0$ is known) that will give us the value of $\boldsymbol{x}_1$.[39] Next, from the esimation of the integral of $\boldsymbol{f}$ over $[t_1,t_2]$ we will find $\boldsymbol{x}_2$. This way, one by one we find all the values $\boldsymbol{x}_n$, $1 \le n \le N$.

If we estimate the integral of $\boldsymbol{f}$ over subinterval $[t_{n-1},t_n]$ using left sums rule (with just one subdivision) we get the [forward] Euler method:

$$\boldsymbol{x}(t+h) = \boldsymbol{x}(t) + h\boldsymbol{f}\big(t,\boldsymbol{x}(t)\big)$$

From the known $\boldsymbol{x}(t_{\text{start}}) = \boldsymbol{x}_0$ we calculate $\boldsymbol{x}_1 = \boldsymbol{x}_0 + h_1\boldsymbol{f}(t_0,\boldsymbol{x}_0)$. Note that this is a straightforward calculation, we get $\boldsymbol{x}_1$ right away. Such numerical schemes for solving ODEs are called *explicit*.

[38] Such a rewrite is not very much useful by itself, as the function $\boldsymbol{x}(t)$ inside the integrand $\boldsymbol{f}(\cdot,\cdot)$ is unknown.
[39] Estimation of the integral may depend on the value of $\boldsymbol{x}_1$, then finding $\boldsymbol{x}_1$ is not straightforward.

13 Runge–Kutta methods

A general Runge–Kutta method is typically defined by writing down its *Butcher tableau*:

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array} \qquad \begin{gathered} \boldsymbol{k}_i = h\boldsymbol{f}\Big(t + c_j h, \boldsymbol{x}(t) + \sum_{j=1}^{s} a_{ij}\boldsymbol{k}_j\Big) \\ \boldsymbol{x}(t+h) \coloneqq \boldsymbol{x}(t) + \sum_{j=1}^{s} b_j \boldsymbol{k}_j \end{gathered}$$

The number s is called a number of stages. The quantity $\boldsymbol{x}(t) + \sum_{j=1}^{s} a_{ij}\boldsymbol{k}_j$ could be thought as a preliminary estimation of $\boldsymbol{x}(t + c_i h)$. The method is explicit if $a_{ij} = 0$ whenever $i \le j$. In this case the preliminary data $\boldsymbol{k}_i$, $i = 1, 2, ..., s$ can be calculated in staightforward computation.

For a method to be at least 1st order of accuracy, we need to have $b_1 + b_2 + ... + b_s = 1$.

It is physically reasonable to have $c_i = \sum_{j=1}^{s} a_{ij}$, as it implies $\boldsymbol{k}_i = h\boldsymbol{f}\big(t + c_i h, \boldsymbol{x}(t + c_i h)\big) + O(h^3)$. Then an explicit method necessarily would have $c_1 = 0$, and $\boldsymbol{k}_1 = h\boldsymbol{f}\big(t, \boldsymbol{x}(t)\big)$, *i.e.*, the 1st stage in an explicit Runge–Kutta method is always a forward Euler step.

A celebrated classical Runge–Kutta method of the 4th order of accuracy (RK4) is given by

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} \qquad \begin{gathered} \boldsymbol{k}_1 \coloneqq h\boldsymbol{f}\big(t, \boldsymbol{x}(t)\big) \\ \boldsymbol{k}_2 \coloneqq h\boldsymbol{f}\big(t + \tfrac{h}{2}, \boldsymbol{x}(t) + \tfrac{1}{2}\boldsymbol{k}_1\big) \\ \boldsymbol{k}_3 \coloneqq h\boldsymbol{f}\big(t + \tfrac{h}{2}, \boldsymbol{x}(t) + \tfrac{1}{2}\boldsymbol{k}_2\big) \\ \boldsymbol{k}_4 \coloneqq h\boldsymbol{f}\big(t + h, \boldsymbol{x}(t) + \boldsymbol{k}_3\big) \\ \boldsymbol{x}(t+h) \coloneqq \boldsymbol{x}(t) + \tfrac{1}{6}\big(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4\big) \end{gathered}$$

It is an explicit method with 4 stages. Let us demonstrate that it is indeed of the 4th order of accuracy. It will be convenient to use the following notation: $\boldsymbol{F}^{(m,n)}_{j_1 j_2 \cdots j_n} \coloneqq h^{m+1} \frac{\partial^m}{\partial t^m} \frac{\partial^n}{\partial X_{j_1} \partial X_{j_2} \ldots \partial X_{j_n}} \boldsymbol{f}\big(t, \boldsymbol{X}\big)\big|_{\boldsymbol{X}=\boldsymbol{x}(t)}$. We will write $\boldsymbol{F}$ instead of $\boldsymbol{F}^{(0,0)} = h\boldsymbol{f}\big(t, \boldsymbol{x}(t)\big)$. For the dynamics $\mathrm{d}\boldsymbol{x}/\mathrm{d}t = \boldsymbol{f}\big(t, \boldsymbol{x}(t)\big)$, we would have (this

is general, and not about RK4)

$$
\begin{aligned}
\Delta\boldsymbol{x} &= \boldsymbol{x}(t+h)-\boldsymbol{x}(t) = \left(h\tfrac{\mathrm{d}}{\mathrm{d}t}+\tfrac{h^2}{2}\tfrac{\mathrm{d}^2}{\mathrm{d}t^2}+\tfrac{h^3}{6}\tfrac{\mathrm{d}^3}{\mathrm{d}t^3}+\tfrac{h^4}{24}\tfrac{\mathrm{d}^4}{\mathrm{d}t^4}\right)\boldsymbol{x}(t)+O(h^5) = \boldsymbol{F}+\\
&+\left(\tfrac{h^2}{2}\tfrac{\mathrm{d}}{\mathrm{d}t}+\tfrac{h^3}{6}\tfrac{\mathrm{d}^2}{\mathrm{d}t^2}+\tfrac{h^4}{24}\tfrac{\mathrm{d}^3}{\mathrm{d}t^3}\right)\boldsymbol{f}\big(t,\boldsymbol{x}(t)\big)+\ldots = \boldsymbol{F}+\tfrac{1}{2}\boldsymbol{F}^{(1,0)}+\tfrac{1}{2}\boldsymbol{F}^{(0,1)}_{j}F_j+\\
&+\left(\tfrac{h^3}{6}\tfrac{\mathrm{d}}{\mathrm{d}t}+\tfrac{h^4}{24}\tfrac{\mathrm{d}^2}{\mathrm{d}t^2}\right)\left(\tfrac{\partial \boldsymbol{f}(t,\boldsymbol{X})}{\partial t}+\tfrac{\partial \boldsymbol{f}(t,\boldsymbol{X})}{\partial X_j}f_j\big(t,\boldsymbol{X}\big)\right)\Big|_{\boldsymbol{X}=\boldsymbol{x}(t)} = \boldsymbol{F}+\tfrac{1}{2}\boldsymbol{F}^{(1,0)}+\tfrac{1}{2}\boldsymbol{F}^{(0,1)}_{j}F_j+\\
&+\underbrace{\tfrac{1}{6}\boldsymbol{F}^{(2,0)}+\tfrac{1}{6}\boldsymbol{F}^{(1,1)}_{j}F_j}_{\frac{h^3}{6}\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial}{\partial t}\boldsymbol{f}(t,\boldsymbol{X})\big|_{\boldsymbol{X}=\boldsymbol{x}(t)}}+\underbrace{\tfrac{1}{6}\boldsymbol{F}^{(1,1)}_{j}F_j+\tfrac{1}{6}\boldsymbol{F}^{(0,2)}_{jk}F_jF_k+\tfrac{1}{6}\boldsymbol{F}^{(0,1)}_{j}F^{(1,0)}_{j}+\tfrac{1}{6}\boldsymbol{F}^{(0,1)}_{j}F^{(0,1)}_{j;k}F_k}_{\frac{h^3}{6}\frac{\mathrm{d}}{\mathrm{d}t}f_j(t,\boldsymbol{X})\frac{\partial}{\partial X_j}\boldsymbol{f}(t,\boldsymbol{X})\big|_{\boldsymbol{X}=\boldsymbol{x}(t)}}+\\
&+\tfrac{h^4}{24}\tfrac{\mathrm{d}}{\mathrm{d}t}\Bigg[\tfrac{\partial^2\boldsymbol{f}(t,\boldsymbol{X})}{\partial t^2}+2\tfrac{\partial^2\boldsymbol{f}(t,\boldsymbol{X})}{\partial t\partial X_j}f_j\big(t,\boldsymbol{X}\big)+\tfrac{\partial^2\boldsymbol{f}(t,\boldsymbol{X})}{\partial X_j\partial X_k}f_j\big(t,\boldsymbol{X}\big)f_k\big(t,\boldsymbol{X}\big)+\tfrac{\partial\boldsymbol{f}(t,\boldsymbol{X})}{\partial X_j}\tfrac{\partial f_j(t,\boldsymbol{X})}{\partial t}+\\
&+\tfrac{\partial\boldsymbol{f}(t,\boldsymbol{X})}{\partial X_j}\tfrac{\partial f_j(t,\boldsymbol{X})}{\partial X_k}f_k\big(t,\boldsymbol{X}\big)\Bigg]\Bigg|_{\boldsymbol{X}=\boldsymbol{x}(t)} = \boldsymbol{F}+\tfrac{1}{2}\boldsymbol{F}^{(1,0)}+\tfrac{1}{2}\boldsymbol{F}^{(0,1)}_{j}F_j+\tfrac{1}{6}\boldsymbol{F}^{(2,0)}+\tfrac{1}{3}\boldsymbol{F}^{(1,1)}_{j}F_j+\tfrac{1}{6}\boldsymbol{F}^{(0,2)}_{jk}F_jF_k+\\
&+\tfrac{1}{6}\boldsymbol{F}^{(0,1)}_{j}F^{(1,0)}_{j}+\tfrac{1}{6}\boldsymbol{F}^{(0,1)}_{j}F^{(0,1)}_{j;k}F_k+\tfrac{1}{24}\boldsymbol{F}^{(3,0)}{}_{\boxed{1}}+\tfrac{1}{8}\boldsymbol{F}^{(2,1)}_{j}F_{j\boxed{12}}+\tfrac{1}{8}\boldsymbol{F}^{(1,2)}_{jk}F_jF_{k\boxed{23}}+\\
&+\tfrac{1}{24}\boldsymbol{F}^{(0,3)}_{jkl}F_jF_kF_{l\boxed{3}}+\tfrac{1}{8}\boldsymbol{F}^{(1,1)}_{j}F^{(1,0)}_{j}{}_{\boxed{24}}+\tfrac{1}{8}\boldsymbol{F}^{(1,1)}_{j}F^{(0,1)}_{j;k}F_{k\boxed{25}}+\tfrac{1}{8}\boldsymbol{F}^{(0,2)}_{jk}F^{(1,0)}_{j}F_{k\boxed{34}}+\\
&+\tfrac{1}{8}\boldsymbol{F}^{(0,2)}_{jk}F^{(0,1)}_{j;l}F_kF_{l\boxed{35}}+\tfrac{1}{24}\boldsymbol{F}^{(0,1)}_{j}F^{(2,0)}_{j}{}_{\boxed{4}}+\tfrac{1}{12}\boldsymbol{F}^{(0,1)}_{j}F^{(1,1)}_{j;k}F_{k\boxed{45}}+\tfrac{1}{24}\boldsymbol{F}^{(0,1)}_{j}F^{(0,2)}_{j;kl}F_kF_{l\boxed{5}}+\\
&+\tfrac{1}{24}\boldsymbol{F}^{(0,1)}_{j}F^{(0,1)}_{j;k}F^{(1,0)}_{k}{}_{\boxed{5}}+\tfrac{1}{24}\boldsymbol{F}^{(0,1)}_{j}F^{(0,1)}_{j;k}F^{(0,1)}_{k;l}F_{l\boxed{5}}
\end{aligned}
$$

The digits in boxes indicate which of the 5 terms in big square brackets do contribute to the adjacent part of the expression.

For the classical Runge–Kutta method we have $\boldsymbol{k}_1 = \boldsymbol{F}$, and

$$\boldsymbol{k}_2 = h\boldsymbol{f}\big(t+\tfrac{h}{2}, \boldsymbol{x}(t)+\tfrac{1}{2}\boldsymbol{F}\big) = \boldsymbol{F} + \underbrace{\tfrac{1}{2}\boldsymbol{F}^{(1,0)} + \tfrac{1}{2}\boldsymbol{F}_j^{(0,1)}F_j}_{O(h^2)\text{ terms}} + \underbrace{\tfrac{1}{8}\boldsymbol{F}^{(2,0)} + \tfrac{1}{4}\boldsymbol{F}_j^{(1,1)}F_j + \tfrac{1}{8}\boldsymbol{F}_{jk}^{(0,2)}F_jF_k}_{O(h^3)\text{ terms}} +$$
$$+ \underbrace{\tfrac{1}{48}\boldsymbol{F}^{(3,0)} + \tfrac{1}{16}\boldsymbol{F}_j^{(2,1)}F_j + \tfrac{1}{16}\boldsymbol{F}_{jk}^{(1,2)}F_jF_k + \tfrac{1}{48}\boldsymbol{F}_{jkl}^{(0,3)}F_jF_kF_l}_{O(h^4)\text{ terms}} + O(h^5)$$

$$\boldsymbol{k}_3 = h\boldsymbol{f}\big(t+\tfrac{h}{2}, \boldsymbol{x}(t)+\tfrac{1}{2}\boldsymbol{k}_2\big) = \boldsymbol{F} + \tfrac{1}{2}\boldsymbol{F}^{(1,0)} + \tfrac{1}{2}\boldsymbol{F}_j^{(0,1)}k_{2j} + \tfrac{1}{8}\boldsymbol{F}^{(2,0)} + \tfrac{1}{4}\boldsymbol{F}_j^{(1,1)}k_{2j} + \tfrac{1}{8}\boldsymbol{F}_{jk}^{(0,2)}k_{2j}k_{2k} +$$
$$+ \tfrac{1}{48}\boldsymbol{F}^{(3,0)} + \tfrac{1}{16}\boldsymbol{F}_j^{(2,1)}k_{2j} + \tfrac{1}{16}\boldsymbol{F}_{jk}^{(1,2)}k_{2j}k_{2k} + \tfrac{1}{48}\boldsymbol{F}_{jkl}^{(0,3)}k_{2j}k_{2k}k_{2l} + O(h^5) = \boldsymbol{F} + \tfrac{1}{2}\boldsymbol{F}^{(1,0)} +$$
$$+ \tfrac{1}{2}\boldsymbol{F}_j^{(0,1)}\Big(F_j + \tfrac{1}{2}F_j^{(1,0)} + \tfrac{1}{2}F_{j;k}^{(0,1)}F_k + \tfrac{1}{8}F_j^{(2,0)} + \tfrac{1}{4}F_{j;k}^{(1,1)}F_k + \tfrac{1}{8}F_{j;kl}^{(0,2)}F_kF_l + O(h^4)\Big) + \tfrac{1}{8}\boldsymbol{F}^{(2,0)} +$$
$$+ \tfrac{1}{4}\boldsymbol{F}_j^{(1,1)}\Big(F_j + \tfrac{1}{2}F_j^{(1,0)} + \tfrac{1}{2}F_{j;k}^{(0,1)}F_k + O(h^3)\Big) + \tfrac{1}{8}\boldsymbol{F}_{jk}^{(0,2)}\Big(F_jF_k + \tfrac{1}{2}F_j^{(1,0)}F_k + \tfrac{1}{2}F_{j;l}^{(0,1)}F_lF_k +$$
$$+ \tfrac{1}{2}F_jF_k^{(1,0)} + \tfrac{1}{2}F_jF_{k;l}^{(0,1)}F_l + \dots\Big) + \tfrac{1}{48}\boldsymbol{F}^{(3,0)} + \tfrac{1}{16}\boldsymbol{F}_j^{(2,1)}F_{2j} + \tfrac{1}{16}\boldsymbol{F}_{jk}^{(1,2)}F_jF_k + \tfrac{1}{48}\boldsymbol{F}_{jkl}^{(0,3)}F_jF_kF_l + \dots$$

$$\boldsymbol{k}_4 = h\boldsymbol{f}\big(t+h, \boldsymbol{x}(t)+\boldsymbol{k}_3\big) = \boldsymbol{F} + \boldsymbol{F}^{(1,0)} + \boldsymbol{F}_j^{(0,1)}k_{3j} + \tfrac{1}{2}\boldsymbol{F}^{(2,0)} + \boldsymbol{F}_j^{(1,1)}k_{3j} + \tfrac{1}{2}\boldsymbol{F}_{jk}^{(0,2)}k_{3j}k_{3k} +$$
$$+ \tfrac{1}{6}\boldsymbol{F}^{(3,0)} + \tfrac{1}{2}\boldsymbol{F}_j^{(2,1)}k_{3j} + \tfrac{1}{2}\boldsymbol{F}_{jk}^{(1,2)}k_{3j}k_{3k} + \tfrac{1}{6}\boldsymbol{F}_{jkl}^{(0,3)}k_{3j}k_{3k}k_{3l} + \dots = \boldsymbol{F} + \boldsymbol{F}^{(1,0)} +$$
$$+ \boldsymbol{F}_j^{(0,1)}\Big(F_j + \tfrac{1}{2}F_j^{(1,0)} + \tfrac{1}{2}F_{j;k}^{(0,1)}\big(F_k + \tfrac{1}{2}F_k^{(1,0)} + \tfrac{1}{2}F_{k;l}^{(0,1)}F_l\big) + \tfrac{1}{8}F_j^{(2,0)} + \tfrac{1}{4}F_{j;k}^{(1,1)}F_k + \tfrac{1}{8}F_{j;kl}^{(0,2)}F_kF_l\Big) +$$
$$+ \tfrac{1}{2}\boldsymbol{F}^{(2,0)} + \boldsymbol{F}_j^{(1,1)}\Big(F_j + \tfrac{1}{2}F_j^{(1,0)} + \tfrac{1}{2}F_{j;k}^{(0,1)}F_k\Big) + \tfrac{1}{2}\boldsymbol{F}_{jk}^{(0,2)}\Big(F_jF_k + \tfrac{1}{2}F_j^{(1,0)}F_k + \tfrac{1}{2}F_{j;l}^{(0,1)}F_lF_k +$$
$$+ \tfrac{1}{2}F_jF_k^{(1,0)} + \tfrac{1}{2}F_jF_{k;l}^{(0,1)}F_l\Big) + \tfrac{1}{6}\boldsymbol{F}^{(3,0)} + \tfrac{1}{2}\boldsymbol{F}_j^{(2,1)}F_j + \tfrac{1}{2}\boldsymbol{F}_{jk}^{(1,2)}F_jF_k + \tfrac{1}{6}\boldsymbol{F}_{jkl}^{(0,3)}F_jF_kF_l + \dots$$

	$\boldsymbol{F}$	$\boldsymbol{F}^{(1,0)}$	$\boldsymbol{F}_j^{(0,1)}F_j$	$\boldsymbol{F}^{(2,0)}$	$\boldsymbol{F}_j^{(1,1)}F_j$	$\boldsymbol{F}_{jk}^{(0,2)}F_jF_k$	$\boldsymbol{F}_j^{(0,1)}F_j^{(1,0)}$	$\boldsymbol{F}_j^{(0,1)}F_{j;k}^{(0,1)}F_k$	$\boldsymbol{F}^{(3,0)}$	$\boldsymbol{F}_j^{(2,1)}F_j$	$\boldsymbol{F}_{jk}^{(1,2)}F_jF_k$	$\boldsymbol{F}_{jkl}^{(0,3)}F_jF_kF_l$	$\boldsymbol{F}_j^{(1,1)}F_j^{(1,0)}$	$\boldsymbol{F}_j^{(1,1)}F_{j;k}^{(0,1)}F_k$	$\boldsymbol{F}_{jk}^{(0,2)}F_j^{(1,0)}F_k$	$\boldsymbol{F}_{jk}^{(0,2)}F_{j;l}^{(0,1)}F_kF_l$	$\boldsymbol{F}_j^{(0,1)}F_j^{(2,0)}$	$\boldsymbol{F}_j^{(0,1)}F_{j;k}^{(1,1)}F_k$	$\boldsymbol{F}_j^{(0,1)}F_{j;kl}^{(0,2)}F_kF_l$	$\boldsymbol{F}_j^{(0,1)}F_{j;k}^{(0,1)}F_k^{(1,0)}$	$\boldsymbol{F}_j^{(0,1)}F_{j;k}^{(0,1)}F_{k;l}^{(0,1)}F_l$
$\Delta\boldsymbol{x}$	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{24}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{24}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{24}$	$\frac{1}{12}$	$\frac{1}{24}$	$\frac{1}{24}$	$\frac{1}{24}$
$\boldsymbol{k}_1$	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\boldsymbol{k}_2$	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{8}$	0	0	$\frac{1}{48}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{48}$	0	0	0	0	0	0	0	0	0
$\boldsymbol{k}_3$	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{48}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{48}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{16}$	0	0
$\boldsymbol{k}_4$	1	1	1	$\frac{1}{2}$	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{4}$

We have $\Delta\boldsymbol{x} = \boldsymbol{x}(t+h) - \boldsymbol{x}(t) = (\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4)/6 + O(h^5)$, and the classical Runge–Kutta method RK4 is indeed of the 4th order of accuracy.

Problems and exercises

1. Consider the system of equations $\mathrm{d}x/\mathrm{d}t = v$, $\mathrm{d}v/\mathrm{d}t = -\sin(x) - 0.02v$. Solve the system with the initial condition $x(0) = 0$, $v(0) = 2.125$ numerically, using forward Euler, explicit midpoint (RK2), and classical Runge–Kutta (RK4) methods. Find out how the error in

$$\begin{bmatrix} x(20) & v(20) \end{bmatrix} = \begin{bmatrix} 6.8426504104428864014... & 1.7912033841288853138... \end{bmatrix}$$

scales with h.

2. Consider the system of equations $\mathrm{d}x/\mathrm{d}t = p$, $\mathrm{d}p/\mathrm{d}t = x - x^2$.[40] Solve the system with the initial condition $x(0) = 0.01$, $p(0) = 0.009$ numerically, using forward Euler, explicit midpoint (RK2), and classical Runge–Kutta (RK4) methods. Log-log plot the error in

$$\begin{bmatrix} x(20) & p(20) \end{bmatrix} = \begin{bmatrix} 0.48859294559329852479... & 0.40118050259290873684... \end{bmatrix}$$

vs. the step size h for all the three methods.

3. Consider the system of N ordinary differential equations $(1 < n < N)$

$$\frac{\mathrm{d}u_1(t)}{\mathrm{d}t} = \frac{-u_1(t) + u_2(t)}{2(\Delta x)^2}, \quad \frac{\mathrm{d}u_n(t)}{dt} = \frac{u_{n-1}(t) - 2u_n(t) + u_{n+1}(t)}{2(\Delta x)^2}, \quad \frac{\mathrm{d}u_N(t)}{\mathrm{d}t} = \frac{u_{N-1}(t) - u_N(t)}{2(\Delta x)^2}$$

where $\Delta x = 1/N$, and $N = 100$. (This is a discretization of $\partial u/\partial t = \frac{1}{2}\partial^2 u/\partial x^2$ diffusion equation on $0 < x < 1$ segment, with zero flux boundary at the walls located at $x = 0$ and $x = 1$. The quantity $u_n(t)$ could be thought as the concentration of diffusing particles inside $\big((n-1)/N, n/N\big)$ interval.) Consider the initial condition $u_n(0) \coloneqq C_N x_n^2(1 - x_n)$, where $x_n \coloneqq (n - \frac{1}{2})/N$ and $C_N \coloneqq N/\sum_{n=1}^{N} x_n^2(1 - x_n)$. Solve the system to find $u_n(1)$ by (a) forward Euler using the time step $\tau = 1/9992$; (b) forward Euler with $\tau = 1/10000$; (c) backward Euler, $\tau = 1/100$. Plot $u_n(t = 1)$ as a function of n.

14 Adaptive step size

To reduce computational cost / improve accuracy of computation we would like to increase / decrease the step size. The former reduces the number of steps, while the latter shrinks the local (and then the global) error in each step. We would like to go with large steps through dull, uninteresting parts of our dynamics, while it is desirable to make small steps in tricky parts of the dynamics in order not to lose accuracy. To do so, we constantly need to be aware of whether we are satisfied with the current, intantaneous quality of solution.

An easy way to estimate the accuracy of numerical solution is to compare it with another solution, of comparable or even better quality. With whatever numerical scheme you are using, one possibility is to compare $\boldsymbol{x}(t+h)$, obtained from $\boldsymbol{x}(t)$ by one full step h, with $\boldsymbol{x}(t+h)$ obtained from $\boldsymbol{x}(t+\frac{1}{2}h)$, which itself is an $(h/2)$-update of $\boldsymbol{x}(t)$. Then we compare the two versions of $\boldsymbol{x}(t+h)$, and if, let us say, it is greater than some tolerance level, we do not accept such an update of $\boldsymbol{x}(t)$. The next thing is to tune the size step in such a way that the predicted difference between the two versions of $\boldsymbol{x}\big(t + \text{new } h\big)$ would be close to the tolerance level:

$$\text{new } h \coloneqq \left(\text{frac} \cdot \frac{\text{tolerance level}}{\text{difference of the two } \boldsymbol{x}(t+h) \text{ versions}}\right)^{1/(p+1)} \cdot h$$

[40] This is a Hamiltonian system, with $\mathcal{H}(x,p) = \frac{1}{2}p^2 - \frac{1}{2}x^2 + \frac{1}{3}x^3$.

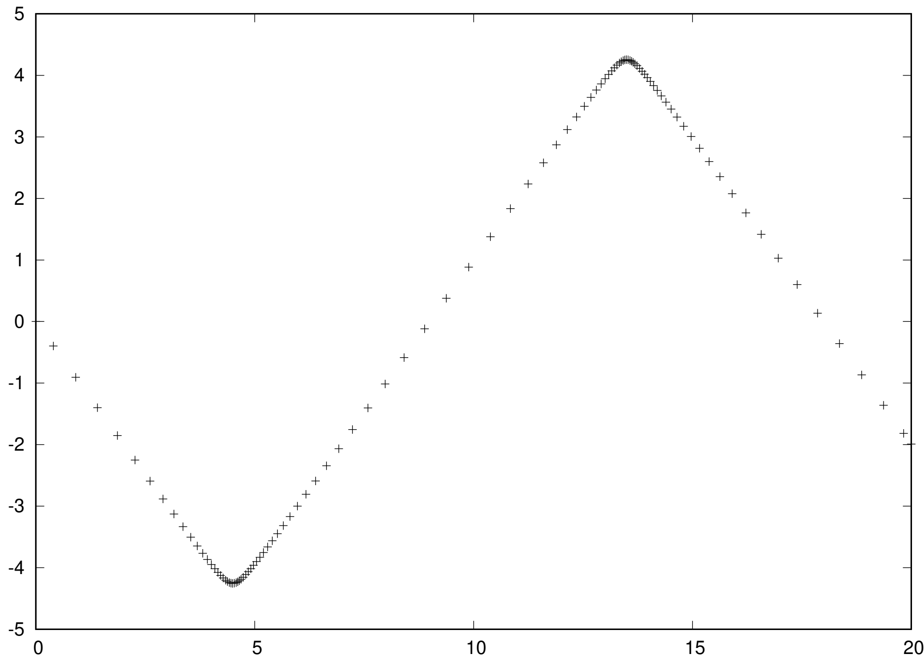
Here p is the order of accuracy of our scheme, and frac is the so called *safety fraction*.

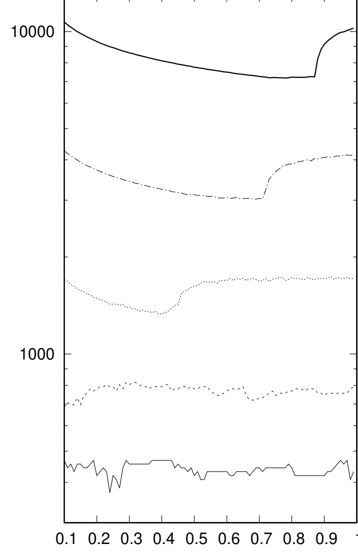
Example 14: Consider the following system of ODEs $\mathrm{d}x/\mathrm{d}t = p$, $\mathrm{d}p/\mathrm{d}t = -x/\sqrt{1+x^2}$ with initial condition $x(0) = 10$, $p(0) = 0$.[41] We would like to construct the trajectory $x(t)$, $p(t)$ for $0 \leq t \leq t_{\mathrm{end}} = 20$. Let us employ RK4 method and choose the step size adaptively, as described above:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define m 2
int counter; /* number of calculations of the r.h.s. f */
void RHS(double t, double *x, double *f)
  { f[0] = x[1]; f[1] = -x[0] / sqrt(1. + pow(x[0], 2.)); counter++; }

/* classical Runge--Kutta method (RK4) */
#define s 4
double X[s][m], K[s][m], a[4][4] = { {   0.,    0.,   0.,   0.},
                                     {1./2.,    0.,   0.,   0.},
                                     {   0., 1./2.,   0.,   0.},
                                     {   0.,    0.,   1.,   0.} },
  b[s] = {1./6., 1./3., 1./3., 1./6.}, c[s] = {0., 1./2., 1./2., 1.};

void explicit_Runge_Kutta(double h, double t, double *x0, double *x1) {
  int i, j, l;
  for (i = 0; i < s; RHS(t + c[i] * h, X[i], K[i]), i++) for (l = 0; l < m; l++)
    for (X[i][l] = x0[l], j = 0; j < i; j++) X[i][l] += h * a[i][j] * K[j][l];
  for (l = 0; l < m; l++)
    for (x1[l] = x0[l], i = 0; i < s; i++) x1[l] += h * b[i] * K[i][l]; }

int main(int argc, char **argv)
 {
  double t, dt, t_end = 20., x[m], xh[m], xhh1[m], xhh2[m];
  double local_error, tolerance = 15. * atof(argv[2]), frac = atof(argv[1]);

  x[0] = 10.; x[1] = 0.; t = 0.; dt = atof(argv[3]);
  printf("%22.16e % 22.16e % 22.16e 0\n", t, x[0], x[1]);
  for (counter = 0; t < t_end;)
   {
    if (t + dt > t_end) dt = t_end - t;
/* Runge--Kutta, full step */
    explicit_Runge_Kutta(dt, t, x, xh);
/* Runge--Kutta, two half steps */
    explicit_Runge_Kutta(0.5 * dt, t, x, xhh1);
    explicit_Runge_Kutta(0.5 * dt, t, xhh1, xhh2);
/* estimating new time step from the mismatch between the two updates */
    local_error = sqrt(pow(xh[0] - xhh2[0], 2.) + pow(xh[1] - xhh2[1], 2.));
/* checking whether the time step is accepted or rejected */
    if (local_error < tolerance) { t += dt; x[0] = xhh2[0]; x[1] = xhh2[1];
      printf("%22.16e % 22.16e % 22.16e %d\n", t, x[0], x[1], counter); }
    dt = dt * pow(frac * tolerance / local_error, 0.2);
   } return 0; }
```

[41] This system is Hamiltonian, with $\mathcal{H}(x,p) = \frac{1}{2}p^2 + \sqrt{1+x^2}$.

Here is the graph of $p(t)$, the tolerance level is 10^{-8}, the points on the graph are actual consecutive points of the obtained numerical solution (one can see that near $x = 0$, where the "velocity" $p(t)$is the maximal, the time step is a lot smaller):





On the right is the graph that shows the dependence of the total number of the r.h.s. $\boldsymbol{f}(t,\boldsymbol{x})$ calculations to reach the final time $t_{\text{end}} = 20$ as a function of frac. The curves correspond (from bottom to top) to tolerance levels 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, and 10^{-12}. Whenever frac is too small, we propagate forward with smaller steps, so we need more steps. On the other hand, if frac is too large, we expect rejections of the step to happen more ofter, so we'll still need many r.h.s. calculations.

Problems and exercises

1. Consider the system of ODEs $\mathrm{d}x/\mathrm{d}t = p$, $\mathrm{d}p/\mathrm{d}t = -x/\sqrt{1+x^2}$ with initial condition $x(0) = 10$, $p(0) = 0$ (Example 14). Find $x(20)$, $p(20)$ by Dormand–Prince method with adaptive step size. Plot how the number of the r.h.s. evaluations changes with the tolerance level.

15 Boundary value problems

15.1 Quasi-linearization

Discretizing the ODEs somehow, write down the BVP as a (linear or non-linear) system of equations, which then solve, *e.g.*, by Newton–Raphson method (in this case the method is called quasi-linearization).

15.2 Shooting method

Consider the BVP $\mathrm{d}\boldsymbol{x}/\mathrm{d}t = \boldsymbol{f}\big(t, \boldsymbol{x}(t)\big)$ with boundary conditions $\boldsymbol{g}_0\big(\boldsymbol{x}(t_0)\big) = \boldsymbol{0}$ and $\boldsymbol{g}_1\big(\boldsymbol{x}(t_1)\big) = \boldsymbol{0}$. Construct the system of equations for $\boldsymbol{x}(t_0)$ being ($\boldsymbol{g}_0 = \boldsymbol{0}$ and $\boldsymbol{g}_1 = \boldsymbol{0}$), in which the argument of $\boldsymbol{g}_1$, the vector $\boldsymbol{x}(t_1)$ is treated as a vector function of $\boldsymbol{x}(t_0)$ that is computed by an ODE solver. The resulting system of equations is solved by methhods of Part III. The parts of $\boldsymbol{x}(t_0)$ which are not immediately determined from $\boldsymbol{g}_0\big(\boldsymbol{x}(t_0)\big) = \boldsymbol{0}$ are called *shooting parameters*.

15.3 Petviashvili factor

15.4 Galerkin method

Problems and exercises

1. Consider the BVP $u'' + u^2 = 0$ with $u(\pm 1) = 0$ boundary conditions. There are two solutions: $u(x) \equiv 0$ and a "non-trivial" one. Find the latter solution by (a) quasi-linearization method $\boldsymbol{u}_{n+1} \coloneqq \boldsymbol{u}_n - \big((\nabla \boldsymbol{f})(\boldsymbol{u}_n)\big)^{-1} \boldsymbol{f}(\boldsymbol{u}_n)$:

$$\boldsymbol{u} = \begin{bmatrix} u(-1) \\ u(-1+h) \\ u(-1+2h) \\ \vdots \\ u(x) \\ \vdots \\ u(1-h) \\ u(1) \end{bmatrix}, \qquad \boldsymbol{f}(\boldsymbol{u}) = \begin{bmatrix} u(-1) \\ u(-1) - 2u(-1+h) + u(-1+2h) + h^2 u^2(-1+h) \\ u(-1+h) - 2u(-1+2h) + u(-1+3h) + h^2 u^2(-1+2h) \\ \vdots \\ u(x-h) - 2u(x) + u(x+h) + h^2 u^2(x) \\ \vdots \\ u(1-2h) - 2u(1-h) + u(1) + h^2 u^2(1-h) \\ u(1) \end{bmatrix}$$

(b) simple shooting method (there is only one shooting parameter here, *e.g.*, $u'(-1)$); and (c) functional iteration with Petviashvili factor:

$$v_n(x) \coloneqq A_n \cdot (x+1) - \int_{-1}^{x} \mathrm{d}\xi_1 \int_{-1}^{\xi_1} \mathrm{d}\xi_2 \; u_n^2(\xi_2), \quad \text{by construction } v_n'' = -u_n^2 \text{ and } v_n(-1) = 0$$

A_n is chosen to enforce $v_n(1) = 0$

$$u_{n+1}(x) \coloneqq v_n(x) \cdot \left(\frac{\int_{-1}^{1} \mathrm{d}\xi \, \big(-v_n''(\xi)\big)}{\int_{-1}^{1} \mathrm{d}\xi \; v_n^2(\xi)} \right)^{\alpha}, \qquad \alpha \coloneqq 1 \text{ for faster convergence}$$

References

[AsPe98] U. M. Ascher, L. R. Petzold, *Computer methods for ODEs and DAEs* (SIAM, 1998).

[Dem97] J. W. Demmel, *Applied numerical linear algebra* (SIAM, Philadelphia, 1997).

[GoVa96] G. H. Golub, C. F. Van Loan, *Matrix computations*, 3rd ed. (Johns Hopkins U. Press, Baltimore, 1996).

[Hig02] N. J. Higham, *Accuracy and stability of numerical algorithms*, 2nd ed. (SIAM, Philadelphia, 2002).

[IEEE85] *IEEE standard for binary floating-point arithmetic, ANSI/IEEE Standard 754-1985* (IEEE, New York, 1985).

[Knu98] D. E. Knuth, *The art of computer programming. Vol. 2. Seminumerical algorithms*, 3rd ed. (Addison Wesley Longman, 1998).

[TrBa97] L. N. Trefethen, D. Bau III, *Numerical linear algebra* (SIAM, 1997).

Index

adjoint, 11
algorithm
  backward stable, 4
  semistable, 5
  stable, 5

back substitution, 18
backward error, 4
backward stable algorithm, 4
bisection method, 29
Butcher tableau, 35

condition number, 8

diagonal matrix, 11
dot product, 11

equivalent norms, 14

Frobenius norm, 12

Hermitian conjugate, 11
Hermitian matrix, 11
Hessenberg decomposition, 29
Hilbert–Schmidt norm, 12
Householder reflection, 22

identity matrix, 11
implementation
  ε-valid, 4
  virtual, 2
induced norm, 12

machine epsilon, 2
matrix, 11
  diagonal, 11
  Hermitian, 11
  identity matrix, 11
  orthogonal, 11
  unitary, 11
  zero matrix, 11

norm
  L^0-"norm", 11
  L^2-norm, 11
  L^∞-norm, 11
  L^p-norm, 11
  Frobenius, 12
  Hilbert–Schmidt, 12
  induced, 12
  operator, 12
  weighted, 11

operator norm, 12
orthogonal matrix, 11
orthogonal vectors, 11

QR factorization, 20

Rayleigh quotient, 28

safety fraction, 38
scalar product, 11
Schur decomposition, 28
semistable algorithm, 5
shooting parameter, 41
singular value, 12
singular value decomposition, 12
singular vector, 12
stable algorithm, 5
SVD, 12

unit roundoff, 2
unit vector, 11
unitary matrix, 11

ε-valid implementation, 4
vector, 11
  unit, 11
virtual implementation, 2

weighted norm, 11

zero matrix, 11

MATH 575B Numerical Analysis
Spring 2020 class notes

Misha Stepanov

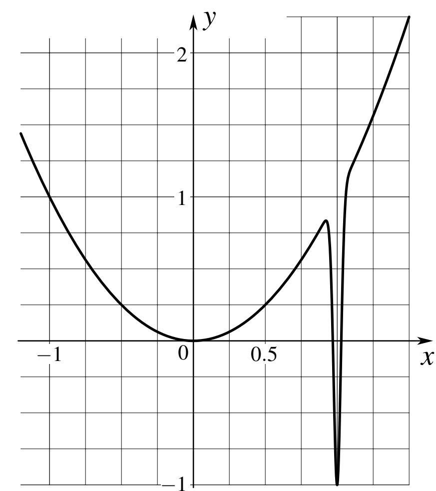
Department of Mathematics and Program in Applied Mathematics
University of Arizona, Tucson, AZ 85721, USA

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, $f(x) = x^2 - 2\exp\bigl(-(x-1)^2/2\sigma^2\bigr)$, with σ 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 $f(\boldsymbol{x})$ is the same as minimizing $-f(\boldsymbol{x})$.

16 Least squares problem

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

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

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

The equation $\nabla f(\boldsymbol{x}_*) = \boldsymbol{0}$ for the position of minimum $\boldsymbol{x}_*$ reads as $\hat{A}\boldsymbol{x}_* = \boldsymbol{b}$. The solution can be found by standard methods like Gaussian elimination, QR factorization, or (as $\hat{A}$ is real symmetric) using Cholesky factorization $\hat{A} = \hat{R}^{\mathrm{T}}\hat{R}$, where $\hat{R}$ 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 $\hat{A}\boldsymbol{x} = \boldsymbol{b}$, 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 $\boldsymbol{x}$ that minimizes the norm of the residual:

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

The minimized function is clearly bounded from below by 0, and if the matrix $\hat{A}$ is of full rank, *i.e.*, $\operatorname{rank}\hat{A} = n < m$, then the solution is unique: $\boldsymbol{x}_* = \left(\hat{A}^T\hat{A}\right)^{-1}\hat{A}^T\boldsymbol{b}$. 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 $\boldsymbol{x}_* = \hat{R}^{-1}\hat{Q}^\dagger\boldsymbol{b} = \hat{V}\hat{\Sigma}^{-1}\hat{U}^\dagger\boldsymbol{b}$ (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), $1 \le i \le N$, 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 $\mathscr{S}_5 = \left\{(-2,-3), (-1,-1), (0,5), (2,5), (3,1)\right\}$ by $y = ax + b$ line using the least squares method,[1] *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 $f(\boldsymbol{x})$. The gradient vector $\nabla f(\boldsymbol{x})$ is the direction of the fastest growth of the function $f(\boldsymbol{x})$, locally we have $f(\boldsymbol{x} + \boldsymbol{\varepsilon}) \approx f(\boldsymbol{x}) + \left(\nabla f(\boldsymbol{x})\right) \cdot \boldsymbol{\varepsilon}$. If we move against its gradient, $\mathrm{d}\boldsymbol{x}/\mathrm{d}t := -\nabla f(\boldsymbol{x})$, then the function $f\left(\boldsymbol{x}(t)\right)$ is non-increasing: $\mathrm{d}f(\boldsymbol{x})/\mathrm{d}t = \nabla f(\boldsymbol{x}) \cdot \mathrm{d}\boldsymbol{x}/\mathrm{d}t = -\|\nabla f\|^2 \le 0$. This gives an idea how to compute the position $\boldsymbol{x}_*$ of the minimum:

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

Some example functions that are going to be used:

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

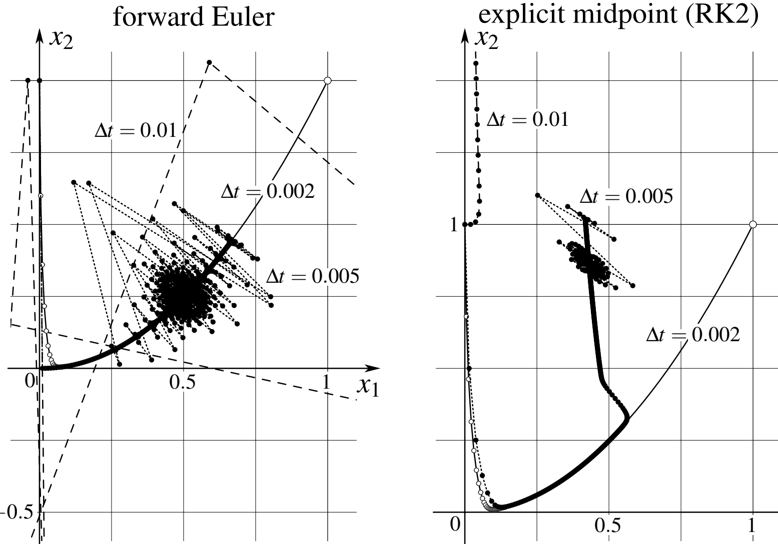
Rosenbrock function[2] $$R_n(x_1, x_2, \dots, x_n) = \sum_{i=1}^{n} (x_i - 1)^2 + A \sum_{i=1}^{n-1} \left(x_{i+1} - x_i^2\right)^2, \quad A \text{ is large, } e.g., A = 100$$

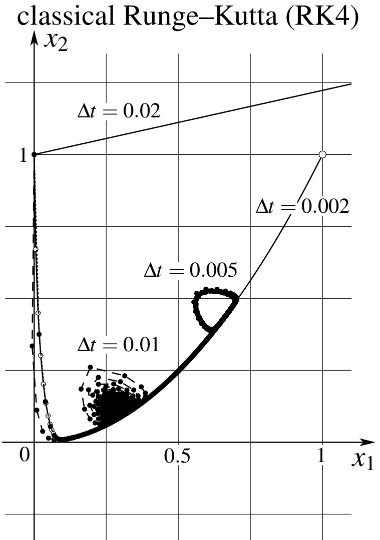
Here the lower index indicates the number of variables.

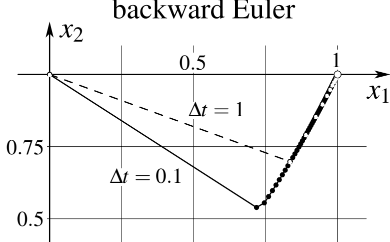
[1] See also the Anscombe's quartet data.

[2] 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 $\mathrm{d}\boldsymbol{x}/\mathrm{d}t = -\nabla R_2(\boldsymbol{x})$ is stiff, *e.g.*, the equation for x_2 is $\mathrm{d}x_2/\mathrm{d}t = -200(x_2 - x_1^2)$, *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 Δt 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 $\nabla f(\boldsymbol{x}_*) = \boldsymbol{0}$, 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:

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

The function in brackets is equal to 0 at $\boldsymbol{X} = \boldsymbol{x}(t)$, and $\|\boldsymbol{X} - \boldsymbol{x}(t)\| \geq 0$, so it is guaranteed that $f\big(\boldsymbol{x}(t + \Delta t)\big) \leq f\big(\boldsymbol{x}(t)\big)$. Still Algorithm $\dot{\boldsymbol{x}} = -\nabla f$ (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 $\nabla f(\boldsymbol{x}_*) = \boldsymbol{0}$, which one may want to try, *e.g.*, due to no good initial guess for $\boldsymbol{x}_*$.

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

```
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.
```

[3] 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 $\nabla \boldsymbol{f}$ being computed through finite differences. Solving the system $\nabla f(\boldsymbol{x}_*) = \boldsymbol{0}$ 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 $\boldsymbol{x}(t)$ 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 $\boldsymbol{x}$
`while` (stopping critedia is not met) `do`

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

pick direction $\Delta\boldsymbol{x}$

gradient descent: $\Delta\boldsymbol{x} \propto -\nabla f(\boldsymbol{x})$
steepest descent: $\Delta\boldsymbol{x} \coloneqq \arg\min_{\|\Delta\boldsymbol{x}\|=1} \nabla \boldsymbol{f} \cdot \Delta\boldsymbol{x}$
$\|\cdot\|$ could be, *e.g.*, L^1- or weighted norm

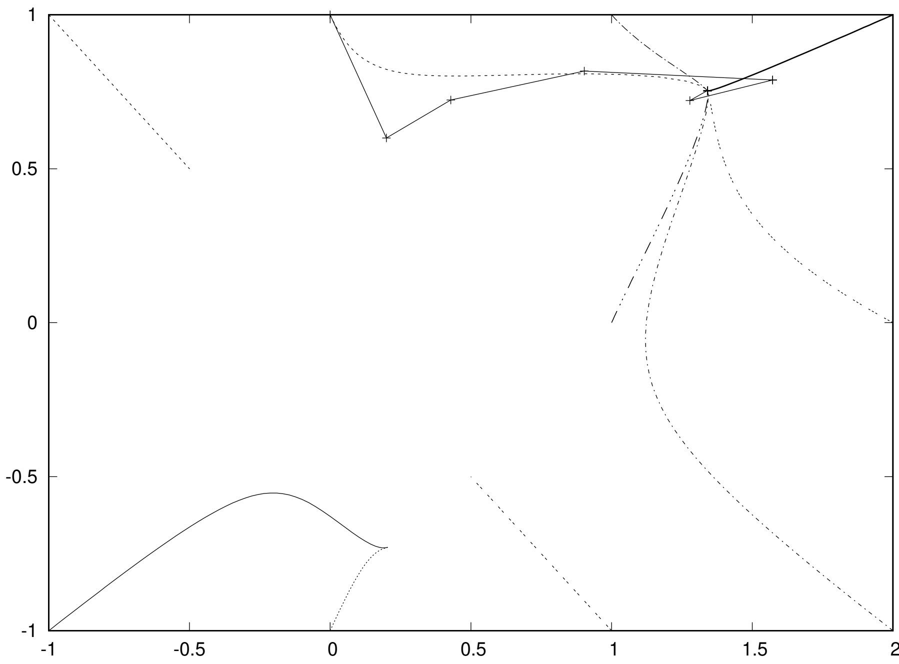
line search: choose step size t

exact line search: $t \coloneqq \arg\min_s f(\boldsymbol{x} + s\Delta\boldsymbol{x})$, *i.e.*, the step size t is found from *exact* one-dimensional minimization[4]
backtracking line search: start with some not too small t, then reduce t until $f(\boldsymbol{x}+t\Delta\boldsymbol{x}) < f(\boldsymbol{x}) + \alpha\nabla f \cdot \big(t\Delta\boldsymbol{x}\big)$, $0 \leq \alpha < 1$

update: $\boldsymbol{x} \coloneqq \boldsymbol{x} + t\Delta\boldsymbol{x}$
`return` $\boldsymbol{x}$

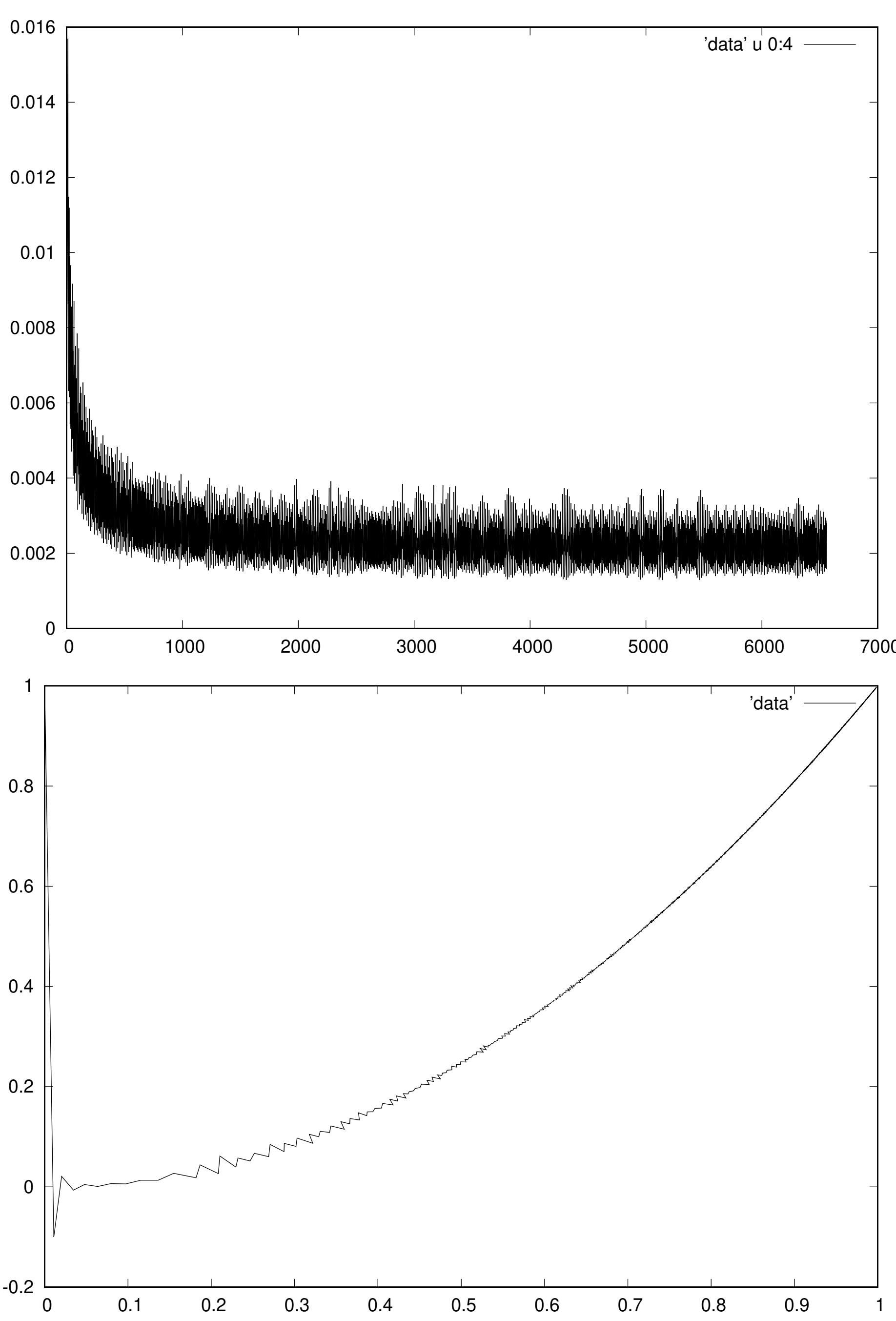
[4] This may be ...

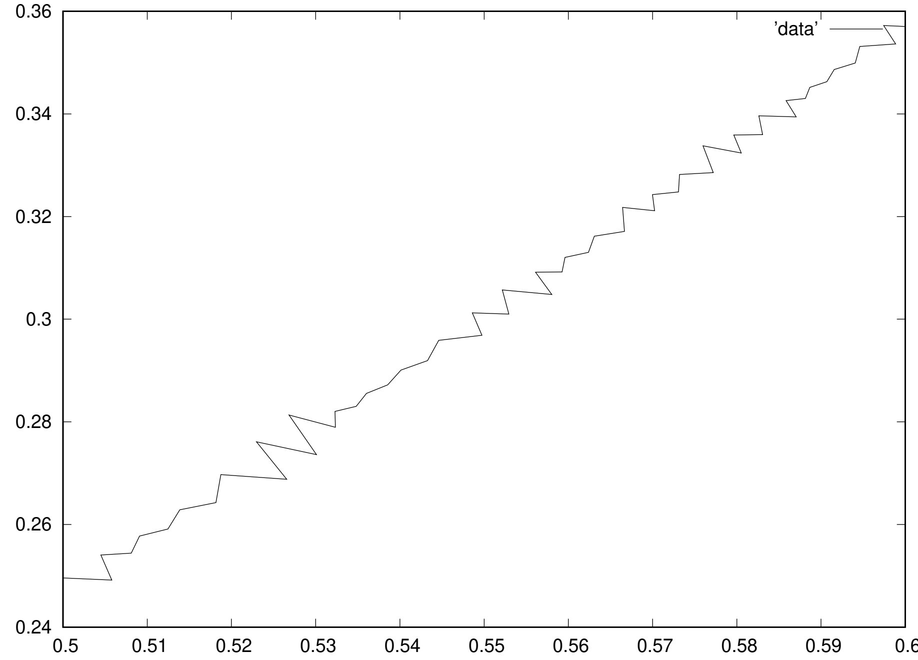
Example 17.1: Here is the gradient descent metod with backtraching, $\alpha = 0$, applied to $G_2(x,y)$:



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 $\boldsymbol{x} = \begin{bmatrix} 0 & 1 \end{bmatrix}^{\mathrm{T}}$):

```
import numpy as np
def f(x):
  return (x[0][0] - 1.)**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.]])
  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
```


'data' u 0:4
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
1000
2000
3000
4000
5000
6000
7000
'data'
1
0.8
0.6
0.4
0.2
-0.2
0.1
0.3
0.5
0.7
0.9



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

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

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

In particular,

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

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 $V_2(x,y) = (x+3)^2 + y^2 \mathrm{e}^{-2x}$. Find the minimum of (a) $V_2(x,y)$ and (b) $W_2(x,z) \coloneqq V_2(x, y = z/20)$ 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 $\|\Delta \boldsymbol{x}\|$ is used?

18 Newton's method

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

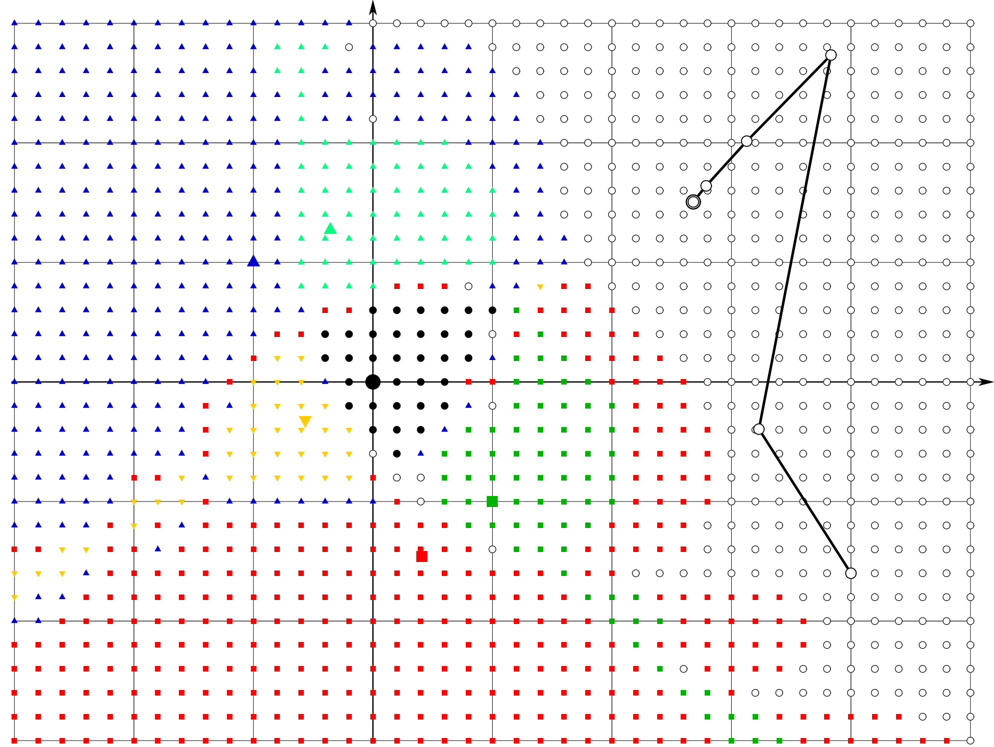
$$\boldsymbol{x} \longmapsto \boldsymbol{x} - (\nabla^2 f)^{-1} \nabla f$$

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

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

As the solutions of $\nabla f(\boldsymbol{x}_*) = \mathbf{0}$ 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 $\begin{bmatrix} x & y \end{bmatrix}^{\mathrm{T}}$ 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 $\boldsymbol{x}$ and $\boldsymbol{y}$ be connected by $\boldsymbol{x} = \hat{T}\boldsymbol{y}$. Consider a function $f(\boldsymbol{x})$ and its deformation $g(\boldsymbol{y}) = f(\hat{T}\boldsymbol{y})$. We have

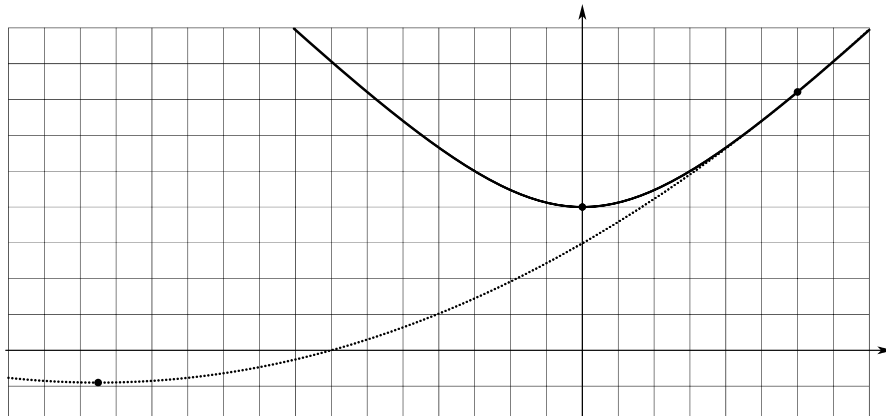
$$\frac{\partial g(\boldsymbol{y})}{\partial y_i} = \sum_j \frac{\partial f(\boldsymbol{x})}{\partial x_j}\bigg|_{\boldsymbol{x}=\hat{T}\boldsymbol{y}} \cdot \underbrace{\frac{\partial x_j}{\partial y_i}}_{T_{ji}} \qquad \text{or} \qquad \nabla_{\boldsymbol{y}} g(\boldsymbol{y}) = \hat{T}^{\mathrm{T}}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

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

$$\boldsymbol{x} = \hat{T}\boldsymbol{y} \longmapsto \hat{T}\Big(\boldsymbol{y} - \big(\nabla^2_{\boldsymbol{y}} g(\boldsymbol{y})\big)^{-1} \nabla_{\boldsymbol{y}} g(\boldsymbol{y})\Big) = \boldsymbol{x} - \hat{T}\Big(\hat{T}^{\mathrm{T}} \nabla^2_{\boldsymbol{x}} f(\boldsymbol{x}) \hat{T}\Big)^{-1} \hat{T}^{\mathrm{T}} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \boldsymbol{x} - \big(\nabla^2_{\boldsymbol{x}} f(\boldsymbol{x})\big)^{-1} \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$$

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

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



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 realiability of the Newton method, ...

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

Levenberg–Marquardt algorithm: $\boldsymbol{x} \longmapsto \boldsymbol{x} - t(\nabla^2 f + \mu\hat{I})^{-1}\nabla f$. In the limit $\mu \to 0 / +\infty$ 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[5]

$$\nabla f(x) = \begin{bmatrix} a \\ b \end{bmatrix}, \qquad \nabla^2 f(x) = \begin{bmatrix} c & 0 \\ 0 & 1 \end{bmatrix}, \qquad -\big(\nabla^2 f\big)^{-1}\nabla f = -\begin{bmatrix} c & 0 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} a \\ b \end{bmatrix} = -\begin{bmatrix} a/c \\ b \end{bmatrix}$$

$$(\nabla f) \cdot \Big(-\big(\nabla^2 f\big)^{-1}\nabla f\Big) = -\frac{a^2}{c} - b^2, \qquad (\nabla f) \cdot \Big(-\big(\nabla^2 f + \mu\hat{I}\big)^{-1}\nabla f\Big) = -\frac{a^2}{\mu+c} - \frac{b^2}{1+\mu}$$

[5] 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 $f : \mathbf{R} \to \mathbf{R}$, 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 trough finite differences. We can consider an update rule $x_{k+1} = x_k - f(x_k) \cdot (x_k - x_{k-1})/\big(f(x_k) - f(x_{k-1})\big)$. Geometrically this corresponds to forming a line that goes through the points $\big(x_{k-1}, f(x_{k-1})\big)$ and $\big(x_k, f(x_k)\big)$, 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
$\sqrt{2}$	1.4142135623730950488016887242096980...	

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 $\boldsymbol{x}_0$ and $\hat{C}_0$ (*e.g.*, $\hat{C}_0 = \hat{I}$)
`while` $(\|\nabla f(\boldsymbol{x}_k)\| > \varepsilon)$ `do`
  pick direction $\Delta\boldsymbol{x}_k := -\hat{C}_k \nabla f(\boldsymbol{x}_k)$
  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 $f(\boldsymbol{x}_k + t\Delta\boldsymbol{x}_k) < f(\boldsymbol{x}_k) + \alpha \nabla f(\boldsymbol{x}_k) \cdot \big(t\Delta\boldsymbol{x}_k\big)$, $0 \le \alpha < 1$
  update: $\boldsymbol{x}_{k+1} := \boldsymbol{x}_k + \big(\boldsymbol{d}_k := t\Delta\boldsymbol{x}_k\big)$
    $\boldsymbol{g}_k := \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k)$
    $\hat{C}_{k+1} := \left(\hat{I} - \dfrac{\boldsymbol{d}_k \boldsymbol{g}_k^{\mathrm{T}}}{\boldsymbol{g}_k^{\mathrm{T}} \boldsymbol{d}_k}\right) \hat{C}_k \left(\hat{I} - \dfrac{\boldsymbol{g}_k \boldsymbol{d}_k^{\mathrm{T}}}{\boldsymbol{g}_k^{\mathrm{T}} \boldsymbol{d}_k}\right) + \dfrac{\boldsymbol{d}_k \boldsymbol{d}_k^{\mathrm{T}}}{\boldsymbol{g}_k^{\mathrm{T}} \boldsymbol{d}_k}$
    $\hat{C}$ is an approximation of inverse Hessian, $\hat{C} \approx (\nabla^2 f)^{-1}$ and $\hat{C}_{k+1}$ is chosen from the condition $\boldsymbol{d}_k = \hat{C}_{k+1}\boldsymbol{g}_k$, which is an approximation of $\nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k) \approx (\nabla^2 f) \cdot \big(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\big)$
  $k \leftarrow k + 1$
`return` $\boldsymbol{x}_{\text{last}}$

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

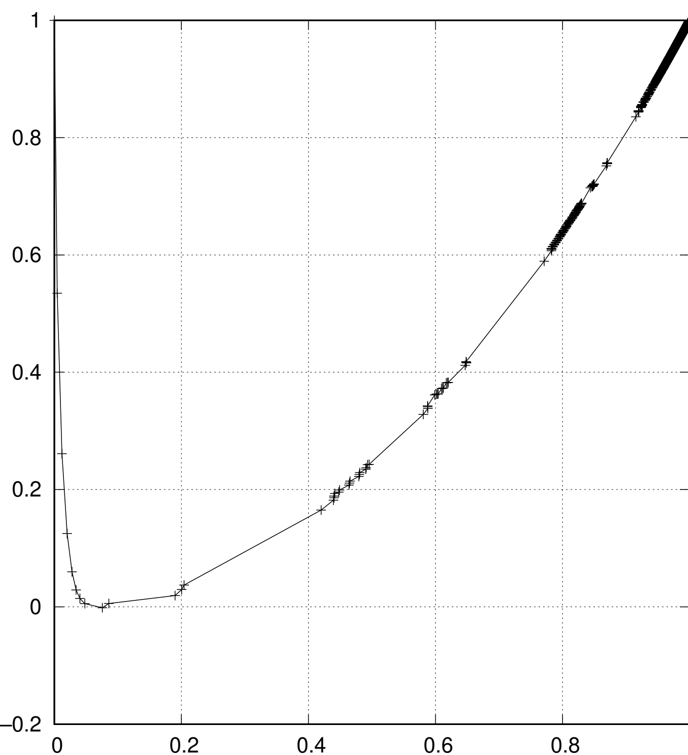
```
function [x] = BFGS_R2(x)
  f = @(x) ((x(1) - 1)^2 + 100 * (x(2) - (x(1))^2)^2);
  df = @(x) (2 * [x(1) - 1; 0] + 200 * (x(2) - (x(1))^2) * [-2 * x(1); 1]);
  F = f(x); dF = df(x); C = eye(length(x));
  while (norm(dF) > 1.e-12)
    d = -C * dF;
    while (f(x + d) >= F)
      d = d / 2.;
      if (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
 0.9999999999998126   0.9999999999995955
 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 $H_2(x,y) \coloneqq \exp(8x-13y+21)+\exp(21y-13x-34)+0.0001\exp(x+y)$. Is it convex/strictly convex? Minimize it, *i.e.*, find $(x_*,y_*) = \arg\min_{(x,y)} H_2(x,y)$.

2. Minimize a function $J_2(x,y) \coloneqq 3xy-2y+1000(x^2+y^2-1.1)\exp\bigl(10(x^2+y^2-1)\bigr)$.

3. Consider a 99-dimensional vector $\boldsymbol{x}$ with components x_1, x_2, ..., 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

$$E_{99}(\boldsymbol{x}) \coloneqq \frac{1}{2}\sum_{i=0}^{99}\bigl(x_{i+1}-x_i\bigr)^2+\frac{1}{16}\sum_{i=1}^{99}\bigl(1-x_i^2\bigr)^2, \qquad \boldsymbol{x}_* = \arg\min_{\boldsymbol{x}} E_{99}(\boldsymbol{x})$$

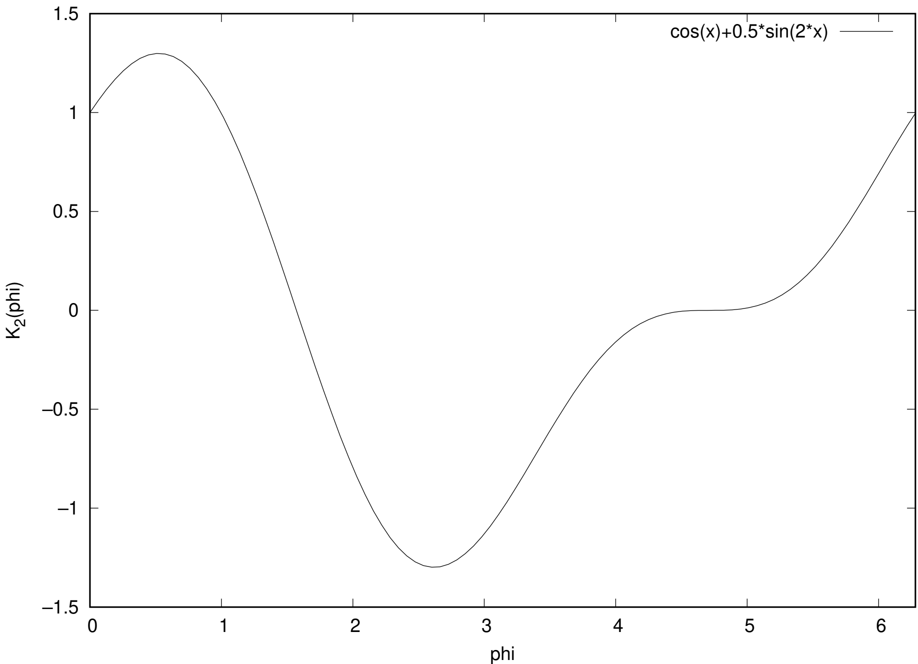
Find $E_{99}(\boldsymbol{x}_*)$. Plot the vector $\boldsymbol{x}_*$ (how the component x_{*i}, $0 \le i \le 100$, changes with i).

19 Equality constrained minimization

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

One method is to parametrize the set of points $\boldsymbol{x}$ satisfying the equality constraints by $n - n_{\text{eq}}$ coordinates $\boldsymbol{y} \in \mathbf{R}^{n-n_{\text{eq}}}$, and then solve unconstrained optimization problem in $\boldsymbol{y}$ 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 $(\cos\varphi, \sin\varphi)$, $0 \le \varphi < 2\pi$. Then we have $K_2(x,y) \to K_2(\varphi) = \cos\varphi + \cos\varphi \sin\varphi = \cos\varphi + \frac{1}{2}\sin 2\varphi$. 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 anf the dual function

$$\mathscr{L}(x,y;\nu) = x + xy + \nu(x^2 + y^2 - 1), \qquad g(\nu) = \inf_{x,y} \mathscr{L}(x,y;\nu) = \begin{cases} -4\nu^3/(4\nu^2 - 1), & \nu > 1/2 \\ -\infty, & \nu \le 1/2 \end{cases}$$

```
[...]/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 =
```

[6] 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   5.000000000000001e-01  -8.660254037844385e-01

octave:3> newton(@Lagrangian_K_2, [-0.3; 0.; 0.])'
counter =  32
ans =
  -8.660254037844431e-01   5.000000000000023e-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  -1.000000000000000e+00   5.960464637411177e-09

octave:5> newton(@Lagrangian_K_2, [-0.8; 0.3; 0.])'
counter =  5
ans =
  -8.660254039596367e-01   5.000000001235305e-01   8.660254033776313e-01
```

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

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

$$\begin{aligned} \hat{Q}\boldsymbol{x} + \hat{A}^{\mathrm{T}}\boldsymbol{\nu} &= \boldsymbol{r} \\ \hat{A}\boldsymbol{x} &= \boldsymbol{b} \end{aligned} \qquad \begin{pmatrix} \hat{Q} & \hat{A}^{\mathrm{T}} \\ \hat{A} & \hat{O} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\nu} \end{pmatrix} = \begin{pmatrix} \boldsymbol{r} \\ \boldsymbol{b} \end{pmatrix}$$

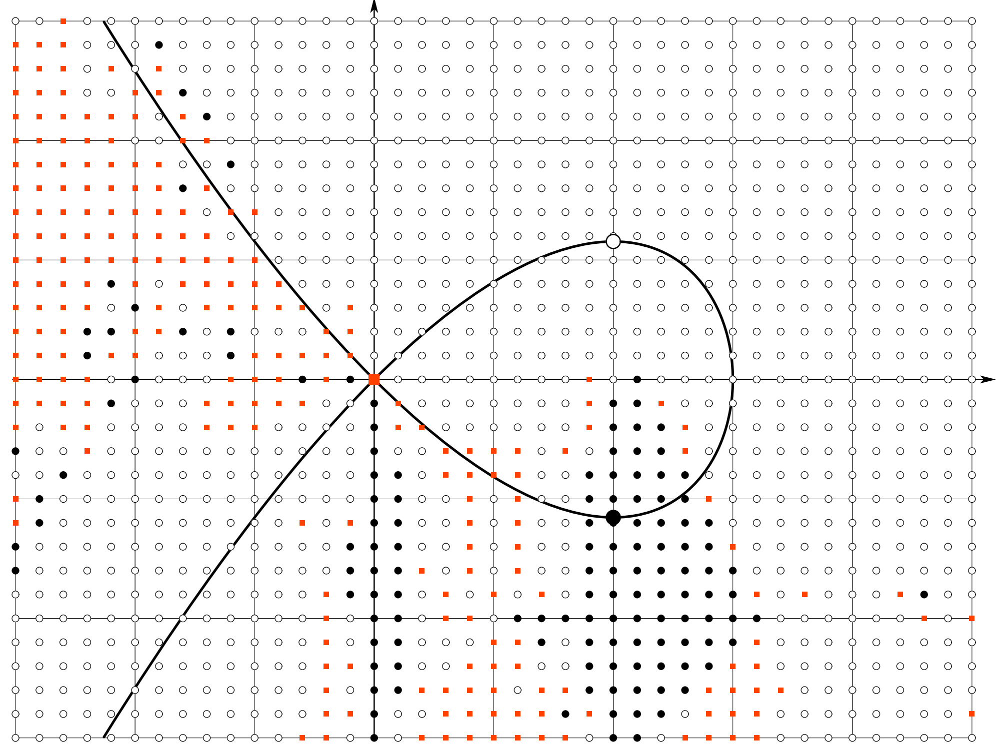
It is an $(n + n_{\mathrm{eq}}) \times (n + n_{\mathrm{eq}})$ system of linear equations, solving which would give the position of optimum $\boldsymbol{x}_*$ that automatically satisfy the condition $\hat{A}\boldsymbol{x}_* = \boldsymbol{b}$, due to the lower part of the system.

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 - 2 x - 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
```

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 $\mathscr{L}(\boldsymbol{x},\boldsymbol{\nu})$) 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 $(x, y, \nu = 0)$ 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 $f(\boldsymbol{x})$ subject to n_{eq} equality constraints $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}$. We form a function

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

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

When t is large, the function $f_t(\boldsymbol{x})$ has narrow valleys along the zero set of $\boldsymbol{h}(\boldsymbol{x})$, 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[7]

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

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

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

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

$$u_* := \arg\min_u \int \mathrm{d}x \Big(\underbrace{\tfrac{1}{2}(u - w)^2 + \lambda \big(u'(x)\big)^2}_{\mathscr{L}(u,u')} \Big)$$

[7] Adapted from [Cal20, Sec. 3.6].

The Euler–Lagrange equation would be linear: $(\partial\mathscr{L}/\partial u')' = \partial\mathscr{L}/\partial u$ 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 $\exp\bigl(-|x|/\sqrt{2\lambda}\bigr)/\sqrt{8\lambda}$.

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 $\nu = p$:

$$u \coloneqq \arg\min_u \Bigl(\tfrac{1}{2}\sum_{\mathsf{v}}\bigl(u_{\mathsf{v}} - w_{\mathsf{v}}\bigr)^2 + \lambda\sum_{\mathsf{e}=\mathsf{v}_1\mathsf{v}_2}\bigl|u_{\mathsf{v}_1} - u_{\mathsf{v}_2}\bigr|^p\Bigr)$$

(It is possible to assign different values of λ to edges, if desired.)

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

$$(u_{*1}, u_{*2}) \coloneqq \arg\min_{(u_1,u_2)}\Bigl(\tfrac{1}{2}(u_1-w_1)^2 + \tfrac{1}{2}(u_2-w_2)^2 + \lambda\sqrt{\varepsilon^2+(u_1-u_2)^2}\Bigr) = \arg\min_{(u_1,u_2)} f(u_1,u_2)$$

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

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

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 $u \coloneqq u_1 - u_2$.) In that case $-\mathrm{d}(\mathrm{d}u/\mathrm{d}t)/\mathrm{d}u = f''(u) = 2\lambda/\varepsilon$ is large, and we need small step size (not greater than ε/λ 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:

$$m\frac{d^2u}{\mathrm{d}t^2} + \frac{\mathrm{d}u}{\mathrm{d}t} = -f'(u) = -\frac{2\lambda u}{\sqrt{\varepsilon^2+u^2}}$$

You may imagine a particle of mass m moving in the potential $f(u)$, while the term $\mathrm{d}u/\mathrm{d}t$ provides a friction force. (In the limit $m, \lambda \to \infty$ with λ/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 $u \propto \mathrm{e}^{\gamma t}$ with $\gamma = -2\lambda/\varepsilon$ (which bounds the time step by $\Delta t \lesssim \varepsilon/\lambda$), we get $\gamma = \pm\sqrt{1/4m^2 - 2\lambda/\varepsilon m} - 1/2m \approx \pm\mathrm{i}\sqrt{2\lambda/\varepsilon m}$, so we need $\Delta t \lesssim \sqrt{\varepsilon m/\lambda}$, *i.e.*, Δt 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 γ 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 2nd 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 ε is small) valleys in a more decisive fashion.

The system of ODEs for the pixels' values looks like

$$\text{for all pixels } \mathsf{v} \qquad m\frac{d^2u_{\mathsf{v}}}{\mathrm{d}t^2} + \frac{\mathrm{d}u_{\mathsf{v}}}{\mathrm{d}t} = w_{\mathsf{v}} - u_{\mathsf{v}} - \lambda \sum_{\mathsf{v}'} \frac{u_{\mathsf{v}} - u_{\mathsf{v}'}}{\sqrt{\varepsilon^2 + (u_{\mathsf{v}} - u_{\mathsf{v}'})^2}}$$

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

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

The noise increases from left to right, and the values of λ 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 λ 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 λ 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, 64×64, 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 λ are shown, $\lambda = 4$ (top), 8, 16, 32, 64, 128, and 256 (bottom).

$\lambda = 4$

$\lambda = 8$

$\lambda = 16$

$\lambda = 32$

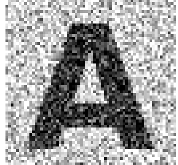
$\lambda = 64$

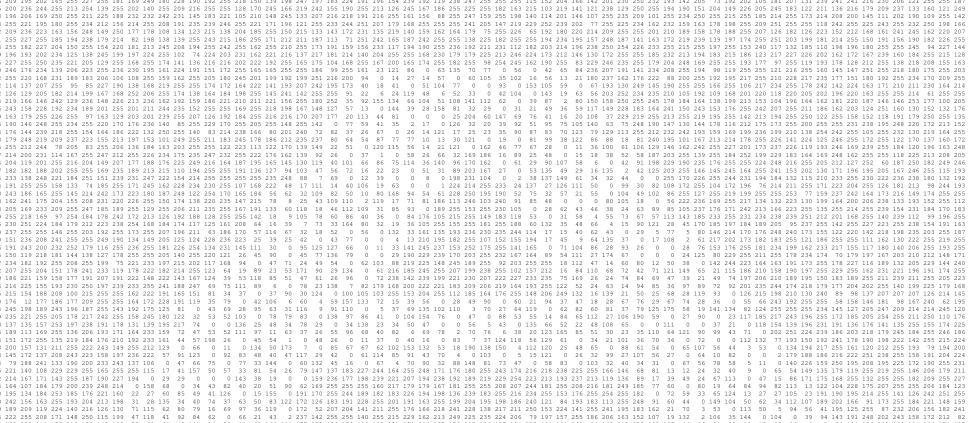
$\lambda = 128$

$\lambda = 256$

Problems and exercises

1. Consider the image below. You can think of it as 64×64 matrix written on the right. Denoise it, using $p = \nu = 1$, $\varepsilon = 0.1$, and some appropriate value of λ.





21 Barrier method

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

$$I_-(g) = \begin{cases} 0, & g \leq 0 \\ +\infty, & g > 0 \end{cases}$$

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 $f_t(\boldsymbol{x}) := f(\boldsymbol{x}) + \sum_i L_t\big(g_i(\boldsymbol{x})\big)$ is close to the solution of our [constrained] optimization problem.

Example 21.1: Consider minimizing $L_2(x,y) = (x-1)^2 + (y-1)^2$ subject to $y^2 - x^2 + 2x^3/3 \leq 0$. We introduce the barrier and find the minimum by damped Newtom method:

```
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
    X = X + d;
  end

octave:1> format long; format compact
octave:2> newton_L2_ineq([0.5; 0.], 1.)'
ans =
   1.000000000000117e+00   2.213803260348214e-01

octave:3> newton_L2_ineq([0.5; 0.], 10.)'
ans =
   1.000000000000000e+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 =
   1.000000000000000e+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 $f_t(x,y) = (x-1)^2 + (y-1)^2 - (1/t)\ln\big(x^2 - y^2 - 2x^3/3\big)$ 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 $y^2 + x^3 \le 0$.

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

22 Linear programming

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

```
[...]/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 */
```

[8] 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 =   0.000000000e+00 inf =   0.000e+00 (2)
*     2: obj =  -7.000000000e+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 $\mathscr{S}_5 = \{(-2,-3), (-1,-1), (0,5), (2,5), (3,1)\}$ by $y = ax + b$ line, with $\max_i |ax_i + b - y_i|$ being minimized. The problem of fitting in L^∞ sense could be written as a linear program

$$\text{minimize } d \text{ subject to } -d \le ax_i + b - y_i \le d \text{ for all } i$$

```
[...]/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 =  8.0000000000000016e-01
b =  1.8000000000000000e+00
[...]/teaching/2020-1/math_575b/notes/linear_programming$
```

Example 22.3: Consider fitting the cloud of 5 points $\mathscr{S}_5$ 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

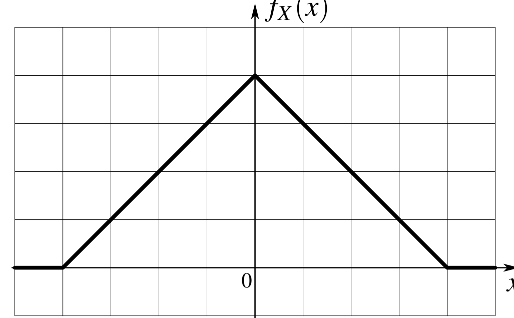
23 Generating pseudo-random numbers

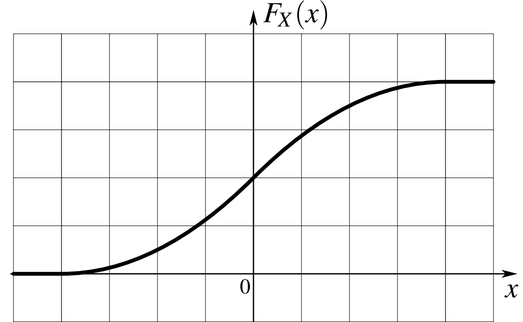
linear congruential generators, RANDU (bad pseudo-random numbers generator)
TestU01, KISS: a bit too simple
GNU GSL RNGs

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

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





The 1st idea is to use the expression for F_X^{-1}:

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

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

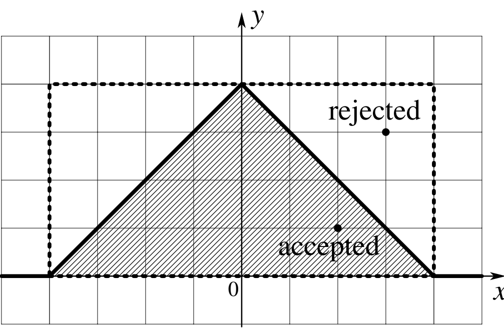
```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
int main(void) { gsl_rng * RNG; int i, j; double u[3], x;
 RNG = gsl_rng_alloc(gsl_rng_ranlux389);

 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; }
```

$$f_{X+Y}(z) = \int \mathrm{d}x\,\mathrm{d}y\; f_X(x) f_Y(y)\, \delta(x+y-z) = \big(f_X * f_Y\big)(z)$$

The 3rd 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; }
```

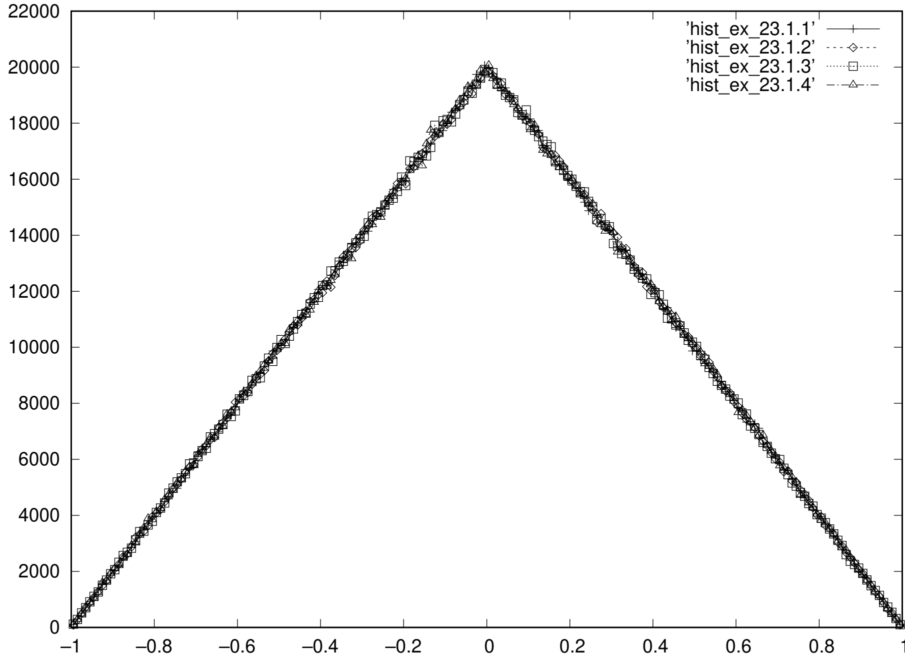
The 4th idea is the rejection method due to John von Neumann. Consider we have the desired density distribution function $f_X(x)$ confined in a rectangle, with the share of the area under the f_X being not small. We can throw a point uniformly distributed in the rectangle by throwing two uniform numbers 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 distrubution.

The 1st idea is to use the inverse cumulative distribution function. Here $F_X(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \mathrm{d}y\, \mathrm{e}^{-y^2/2}$ 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; }
```

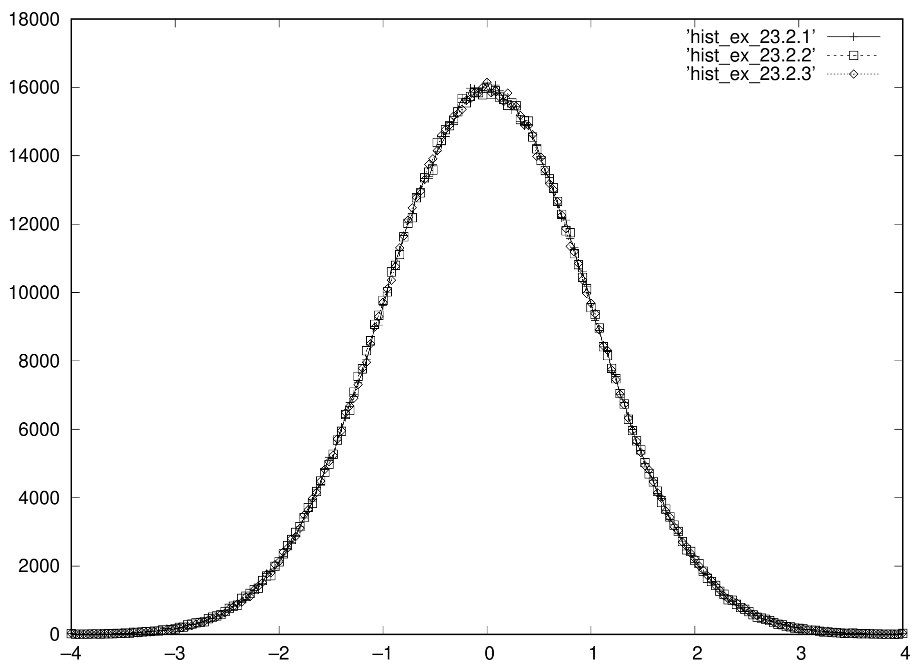
The 2nd idea uses the Cental Limit Theorem. Let us sum 100 (again, this is not too fast) uniformly distributed numbers, and then shift and rescale the result so that we get a number that is [almost] normally distrubuted, with zero mean and standard deviation being equal to 1:

```
#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 3rd idea is exploiting $\frac{1}{2\pi}\mathrm{e}^{-(x^2+y^2)/2}\,\mathrm{d}x\,\mathrm{d}y = \frac{1}{2\pi}\mathrm{e}^{-r^2/2}\,r\,\mathrm{d}r\,\mathrm{d}\theta = \mathrm{e}^{-s}\,\mathrm{d}s\,\frac{\mathrm{d}\theta}{2\pi}$, where $r = \sqrt{x^2+y^2}$ and $s = r^2/2$. We have exponential distribution for s and uniform one for θ. 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

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

24 Monte Carlo method

Imagine you have a stochastic differential equation $\mathrm{d}x/\mathrm{d}t = \xi(t)$, 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 $\lim_{t\to 0^+}\rho(t,x) = \delta(x)$. 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 τ), 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 ξ_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 ζ 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

$$\Big\langle \big(x(t+\tau) - x(t)\big)^2 \Big\rangle = \int_t^{t+\tau} \mathrm{d}t_1 \int_t^{t+\tau} \mathrm{d}t_2 \, \langle \xi(t_1)\xi(t_2)\rangle = \int_t^{t+\tau} \mathrm{d}t_1 \int_t^{t+\tau} \mathrm{d}t_2 \, 2D\delta(t_1 - t_2) = 2D\tau$$

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

$$K_{XX}(t_1,t_2) = \mathsf{E}\Big(\big(X(t_1) - \mathsf{E}X(t_1)\big)\big(X(t_2) - \mathsf{E}X(t_2)\big) \Big)$$

For stationary process $X(t)$ the quantity $\mathsf{E}X(t)$ 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 $K_{XX}(t_1,t_2) = 0$.[9] The smaller $K_{XX}(\tau = t_1 - t_2)$, the less dependence we expect between the values of X separated by τ in time.

Example 24.1: Consider a stochastic differential equation $\mathrm{d}x(t)/\mathrm{d}t = -x(t) + \xi(t)$, 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 ξ, otherwise $x(t)$ exponentially

[9] The reverse is not true. Consider, *e.g.*, X and Y to be uniformly distributed on the circle $X^2 + Y^2 = 1$. Then $\mathsf{E}X = \mathsf{E}Y = \mathsf{E}XY = 0$ (so linear correlation coefficient between X and Y is zero), but X and Y are not independet, they are even functionally dependent.

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

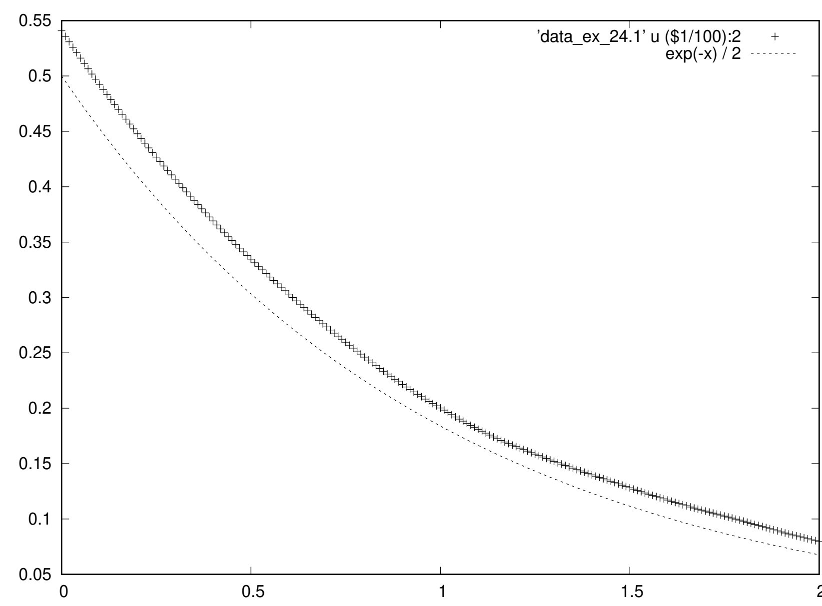
$$K_{xx}(t_1,t_2) = \mathsf{E}x(t_1)x(t_2) = \int_{-\infty}^{t_1} \mathrm{d}t_1' \int_{-\infty}^{t_2} \mathrm{d}t_2'\, \exp\bigl(-(t_1-t_1')-(t_2-t_2')\bigr)\, \underbrace{\mathsf{E}\xi(t_1')\xi(t_2')}_{\delta(t_1'-t_2')}$$

$$= \int_{-\infty}^{\min(t_1,t_2)} \mathrm{d}t'\, \exp\bigl(2t'-t_1-t_2\bigr) = \tfrac{1}{2}\exp\bigl(-|t_1-t_2|\bigr)$$

The process $x(t)$ is stationary, and $K_{xx}(t_1,t_2)$ depends just on t_1-t_2.[10]

The equation can be solved numerically using the update rule $x_{i+1} = \mathrm{e}^{-\tau}x_i + \sqrt{\tau}\zeta_{i+1/2}$, here τ is the time step. Here index in ζ just distinguished the independent trials of standard normal distribution. The statistics of $-\zeta$ is the same as of ζ, and $\mathsf{E}x_i = 0$. As $\mathsf{E}\zeta = 0$, we have $\mathsf{E}x_{i+1}x_i = \mathrm{e}^{-\tau}\mathsf{E}x_i^2$, or $K(\tau) = \mathrm{e}^{-\tau}K(0)$. We have $K(2) = \mathsf{E}x_{i+2}x_i = \mathsf{E}\bigl(\mathrm{e}^{-\tau}x_{i+1} + \sqrt{\tau}\zeta_{i+3/2}\bigr)x_i = \mathrm{e}^{-\tau}K(1) = \mathrm{e}^{-2\tau}K(0)$. It can be shown that $K(t) = \mathrm{e}^{-t}K(0)$, *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])
```



[10] 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 $\mathrm{d}x(t)/\mathrm{d}t = x(t)\,\xi(t)$, 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:

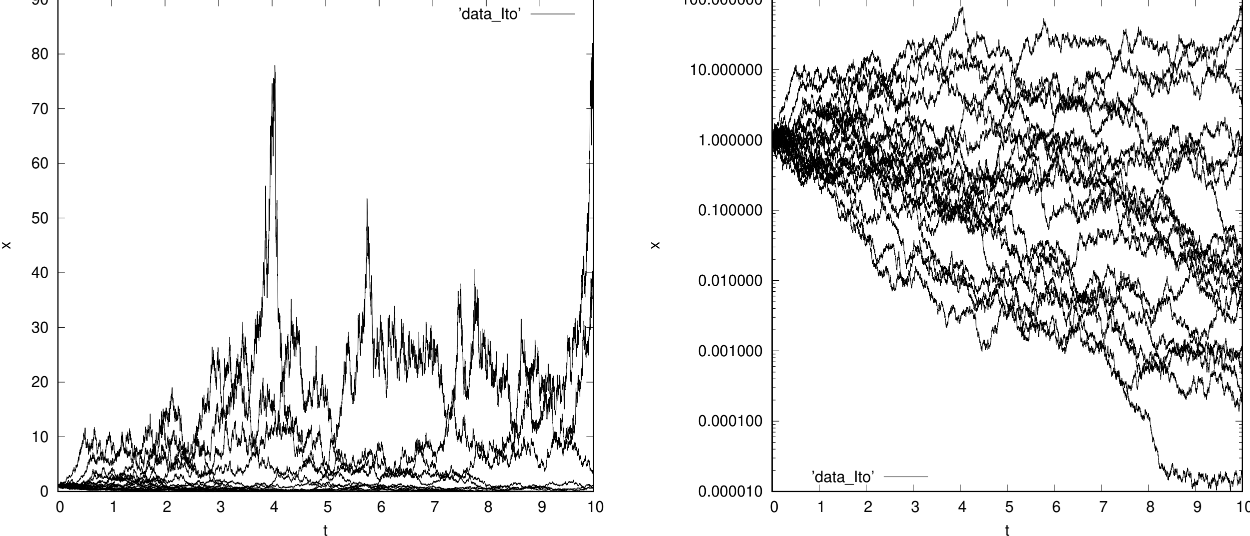
$$\text{Itô:}\quad x_{i+1} = x_i + \sqrt{2D\tau}\,x_i\,\zeta_{i+1/2}$$

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

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

Here is a Python script solving $\mathrm{d}x/\mathrm{d}t = x\xi$ using Itô dicretization ($D = 1/2$):

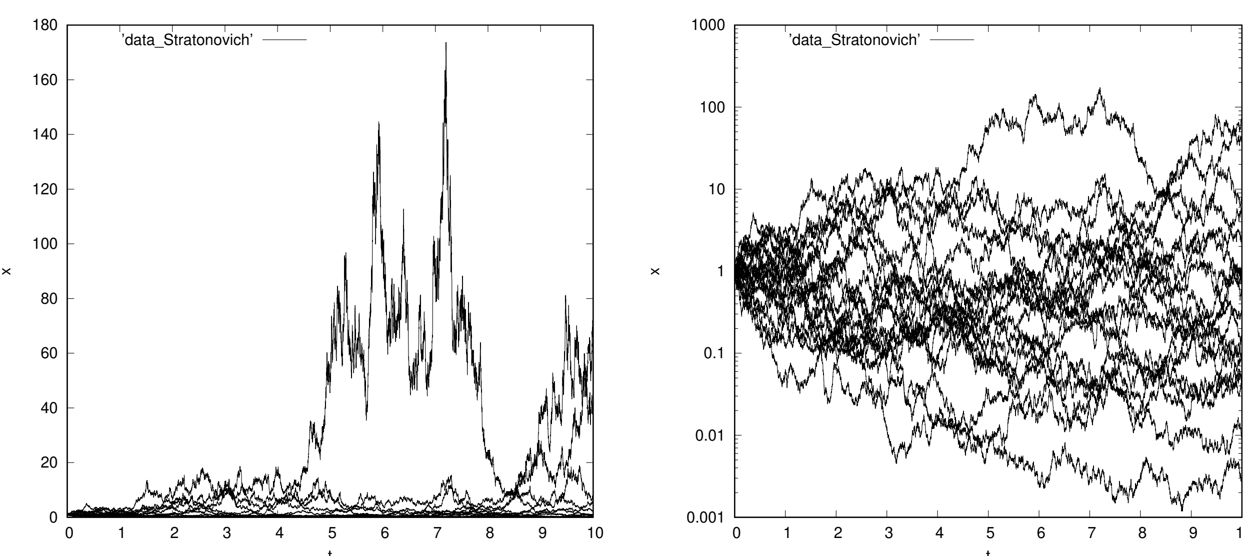
```
from math import sqrt; from random import normalvariate
dt = 0.001
for n in range(0, 20):
  t, x = 0., 1.
  print(t, x)
  for i in range(0, 10000):
    t, x = t + dt, x * (1. + sqrt(dt) * normalvariate(0., 1.))
    print(t, x)
  print()
```



Here is a Python script solving $\mathrm{d}x/\mathrm{d}t = x\xi$ using Stratonovich dicretization ($D = 1/2$):

```
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 $T \coloneqq \min_t x(t) \le 0$) 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

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

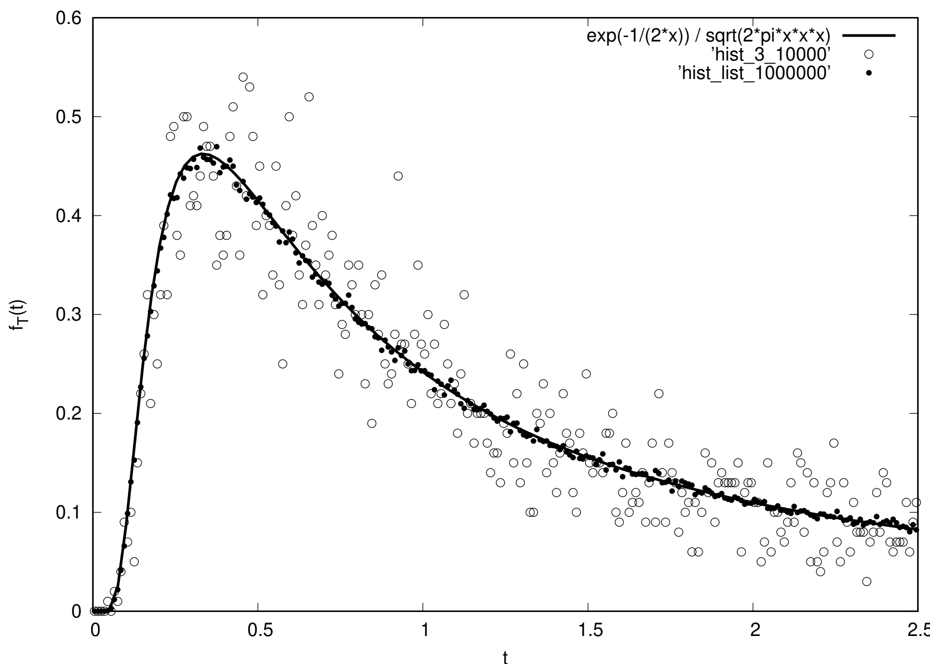
We have $F_T(t) = 1 - \int_0^\infty \mathrm{d}x\, \rho(t,x)$, and

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

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 exits. The histogram from 10^4 [and also 10^6] trials looks like

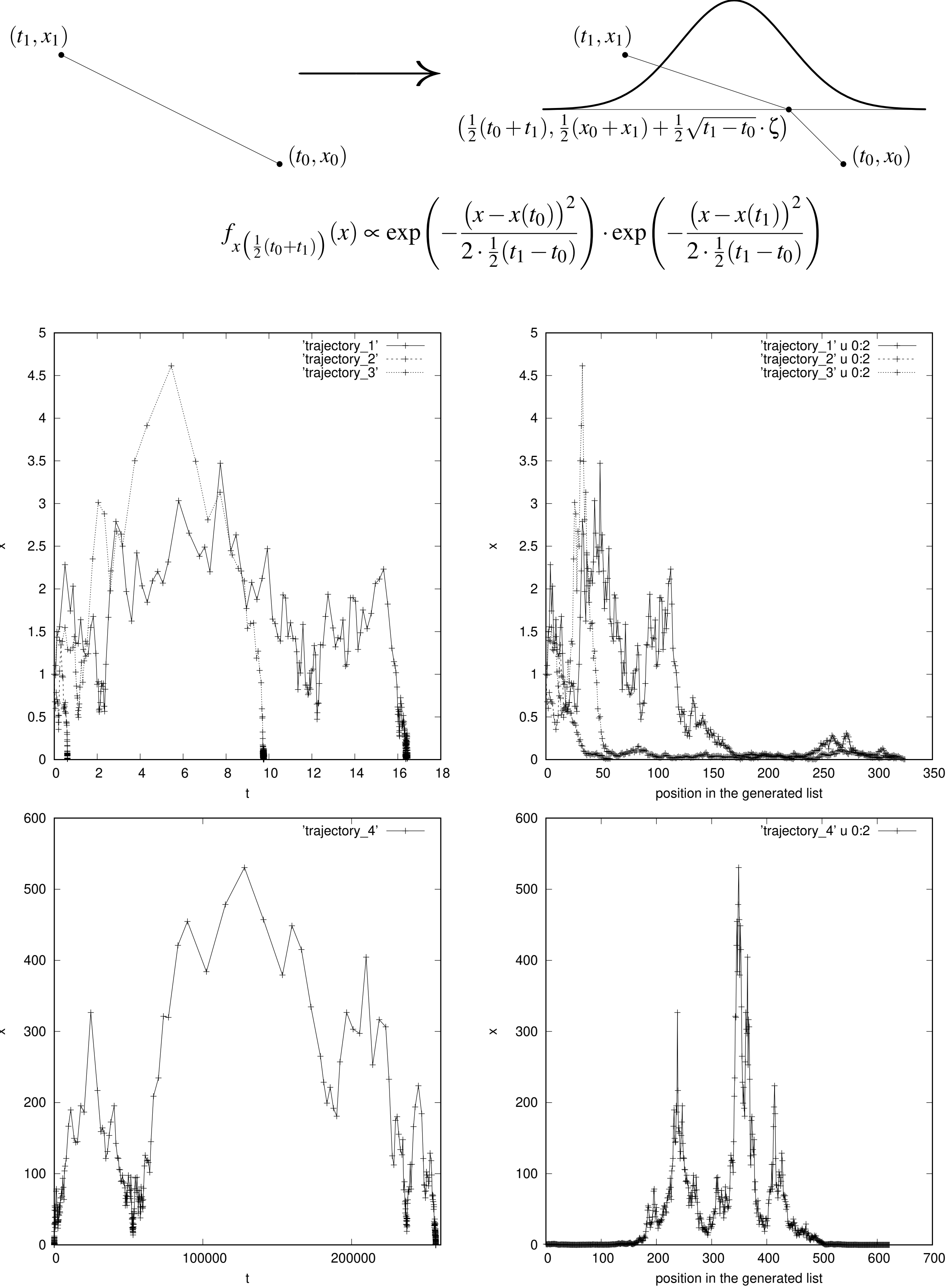


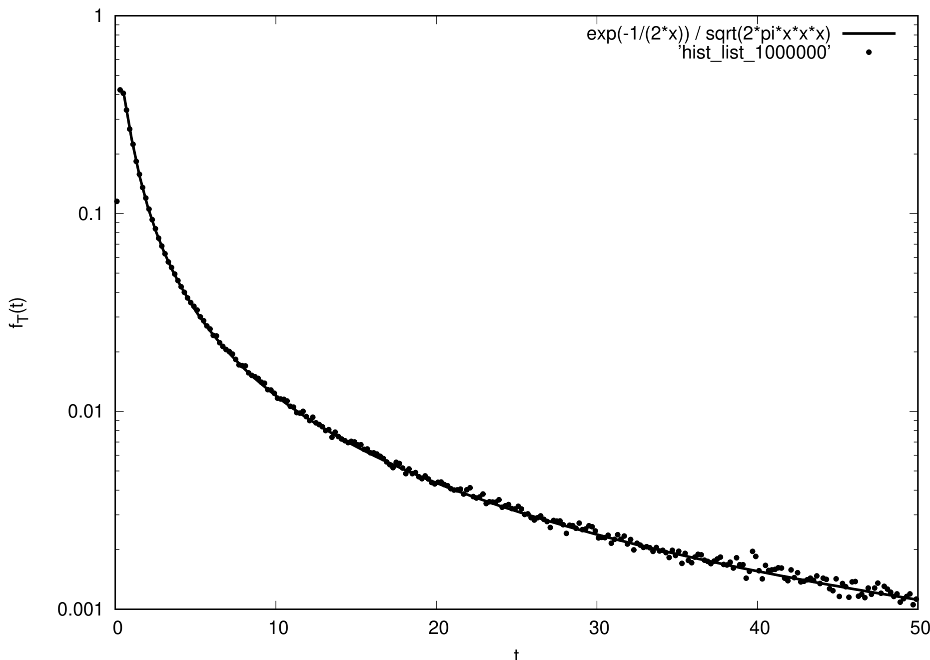
In order to work with many trials, the trials should be done in a more efficient way. In the "naive" script we spend too much time if we go far from the wall. There, to speed up the computation, we can safely do larger in time steps:

```
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 $\mathrm{P}(z > 1) \approx 0.16$.) 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:





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 $\mathsf{E}A(X)$, *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 $x_1, x_2, ..., x_N$, and then eastimate $\mathsf{E}A(X) \approx \frac{1}{N}\sum_{i=1}^{N} A(x_i)$.

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 $\mathsf{E}A(X)$ are quite rare (and we can think of the value of $\mathsf{E}A(X)$ as small). In this case in order to accurately estimate $\mathsf{E}A(X)$ 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 $\mathsf{E}A(X)$. The idea is to sample not $f_X(x)$, but some other distribution $g(x)$, and write

$$\mathsf{E}A(x) = \int \mathrm{d}x\, A(x)\, f_X(x) = \int \mathrm{d}x \left[A(x)\, \frac{f_X(x)}{g(x)} \right] g(x)$$

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 $\mathsf{E}A(x)$ 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 diststibution $f(x) = \mathrm{e}^{-x}$, $x \geq 0$. Let us estimate $\mathsf{P}(x > 10) = \mathrm{e}^{-10} \approx 4.54 \cdot 10^{-5}$. We have $\mathsf{P}(x > 10) = \mathsf{E}A(x)$, 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.

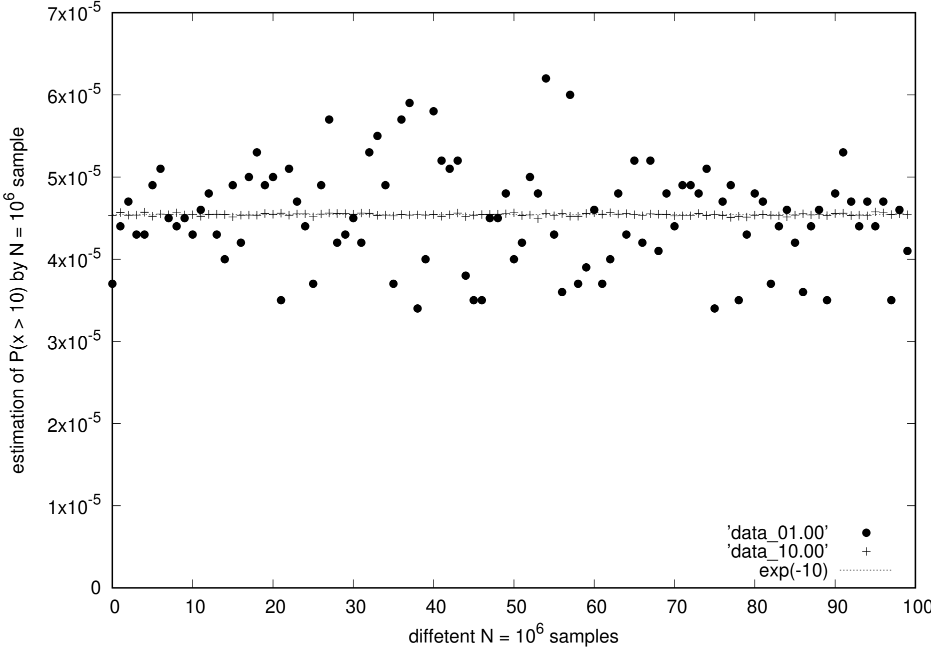
The simplest application of the Monte Carlo method with $N = 10^6$ samples would produce only about 45 events with $x > 10$ on average, with standard deviation being $\approx \sqrt{45} \approx 7$ events. Thus the estimation of $\mathsf{P}(x > 10)$ would be something like $(4.54 \pm 0.7) \cdot 10^{-5}$.

Here is a Python script which estimates $\mathsf{P}(x > 10)$, 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)
```

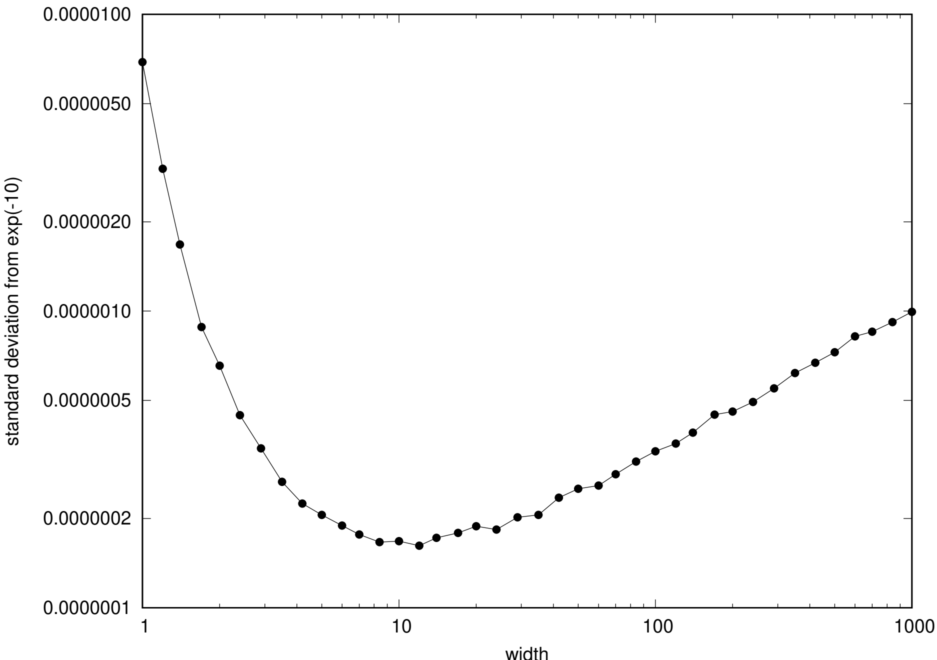
Here $g(x) := \exp\bigl(-x/\texttt{width}\bigr) / \texttt{width}$, $x \geq 0$, *i.e.*, it is an exponential distribution of width `width`. When `width` is large, we often get not small, *i.e.*, > 10 values of x, thus getting more events contributing into our estimation of $\mathsf{P}(x > 10)$. Each realization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

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



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

Here is how the standard deviation $\sqrt{\mathsf{E}\Big(\big(\text{estimation of } \mathsf{P}(x>10) \text{ from } N=10^6 \text{ trials}\big)-\exp(-10)\Big)^2}$ 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 $1/\mathrm{e}\approx 0.37$).

Another variant (and one can think of many more) of deforming $f_X(x)\longrightarrow g(x)$ is shifting the distribution: $g(x)=\exp\big(\texttt{shift}-x\big)$, $x\geq\texttt{shift}$. Increasing `shift` we produce $x>10$ events more often, and out estimation of $\mathsf{P}(x>10)$ is going to be more accurate. At `shift` $=10$ the computation of $\mathsf{P}(x>10)$ is going to be *exact*, as the chosen $g(x)$ exactly coinsides with conditional density $f_X\big(x\big|x>10\big)$. With `shift` $=10$ all the samples will fall into $x>10$ region and in the importance sampling setting will have the same weight: $A(x)f_X(x)/g(x)=1\cdot\exp(-x)/\exp\big(\texttt{shift}-x\big)=\mathrm{e}^{-10}$.[11] When `shift` >10 our estimation of $\mathsf{P}(x>10)$ 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 $\mathsf{P}(x>10)$, namely $10<x<\texttt{shift}$ ones.

Example 24.1.2: Consider a stochastic differential equation $\mathrm{d}x/\mathrm{d}t=-x+\varepsilon x^2+\xi$, where $\xi(t)$ is white noise with correlation function $\langle\xi(t_1)\xi(t_2)\rangle=\delta(t_1-t_2)$. The parameter ε is small. The white noise ξ causes $x(t)$ to diffuse. When x is not large, $x<1/\varepsilon$, there is a tendency to move towards the origin $x=0$. When $x>1/\varepsilon$, then non-random part of $\mathrm{d}x/\mathrm{d}t$ is positive, and with high chances $x(t)$ will move to the right without return (because of εx^2 term in $\mathrm{d}x/\mathrm{d}t$ 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 $x>1/\varepsilon$. 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

[11] 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 $\mathsf{E}A(x)$ itself.

somewhat optimized for making the particle to escape, any substantial deviation from the optimal shape would necessitate the increase in noise amplitute, 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 ξ, but the noise $\xi(t)$ should be such that $x(t)$ reaches $1/\varepsilon$. This can be written as the following variational problem: We need to find extrema of

$$\mathscr{L}\{x(t),\xi(t),p(t)\} = \frac{1}{2}\int \mathrm{d}t\ \xi^2(t) + \int \mathrm{d}t\ p(t)\left(\frac{\mathrm{d}x(t)}{\mathrm{d}t} + x(t) - \varepsilon x^2(t) - \xi(t)\right)$$

The noise density $f\{\xi(t)\} \propto \exp\left(-\frac{1}{2}\int \mathrm{d}t\ \xi^2(t)\right)$, 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 $x(+\infty) = 1/\varepsilon$ (we reach the point after which ξ 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. ξ, we get $\xi(t) = p(t)$. Variation w.r.t. x produces the equation for $p(t)$: $\mathrm{d}p/\mathrm{d}t = p(1-2\varepsilon x)$. The equations for x and p produce a Hamiltonian system: $\mathscr{H}(x,p) := \frac{1}{2}p^2 - p(x-\varepsilon x^2)$, and $\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{\partial\mathscr{H}}{\partial p}$, $\frac{\mathrm{d}p}{\mathrm{d}t} = -\frac{\partial\mathscr{H}}{\partial x}$. On the starting and ending points, $(x,p) = (0,0)$ and $(1/\varepsilon, 0)$, we have $\mathscr{H} = 0$, while $\mathscr{H}$ as the Hamiltonian does not depend on time. Thus $\mathscr{H} \equiv 0$, and $p = 2(x - \varepsilon x^2)$. This gives $\frac{\mathrm{d}x}{\mathrm{d}t} = x - \varepsilon x^2$ with the solution $x(t) = \frac{1}{2\varepsilon}\big(1+\tanh(t/2)\big)$ (and actually also its shifts in time). We get $\xi(t) = p(t) = 2\big(x(t) - \varepsilon x^2(t)\big) = 1/2\varepsilon\cosh^2(t/2)$. The probability per unit time to escape (with some accuracy) can be estimated as

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

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

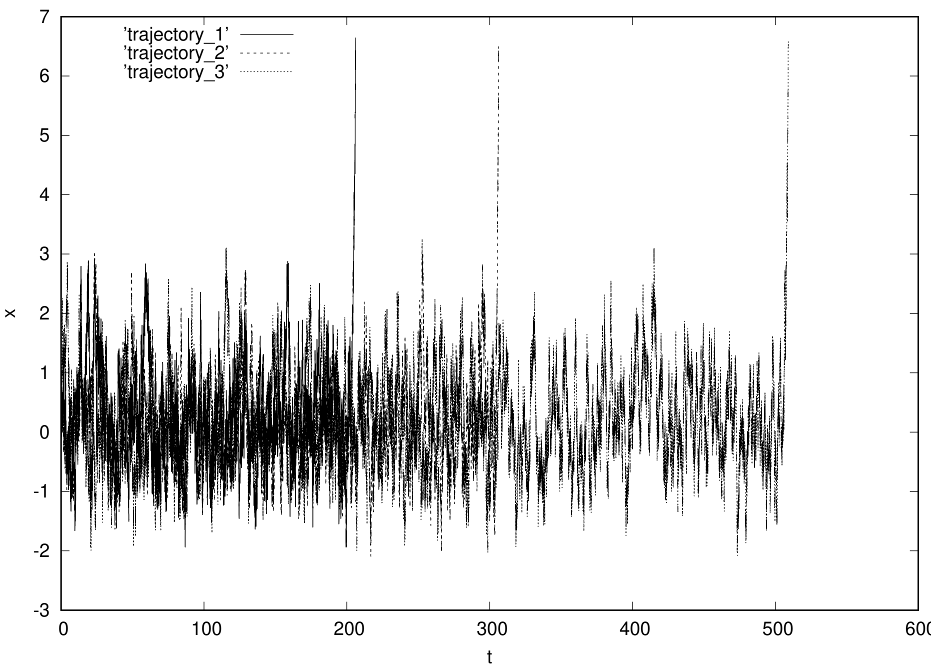
$$\frac{\partial\rho(t,x)}{\partial t} = \frac{\partial}{\partial x}\left(\frac{1}{2}\frac{\partial}{\partial x} + x - \varepsilon x^2\right)\rho(t,x)$$

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

(c) We can directly simulate the SDE $\mathrm{d}x/\mathrm{d}t = x - \varepsilon x^2 + \xi$:

```
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)
```

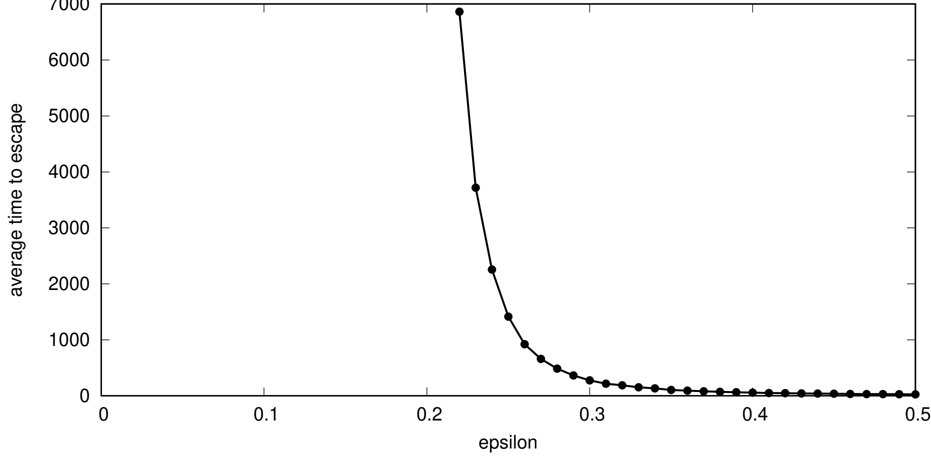
and get how $x(t)$ could depends t. Here are 3 realizations for $\varepsilon = 0.3$:

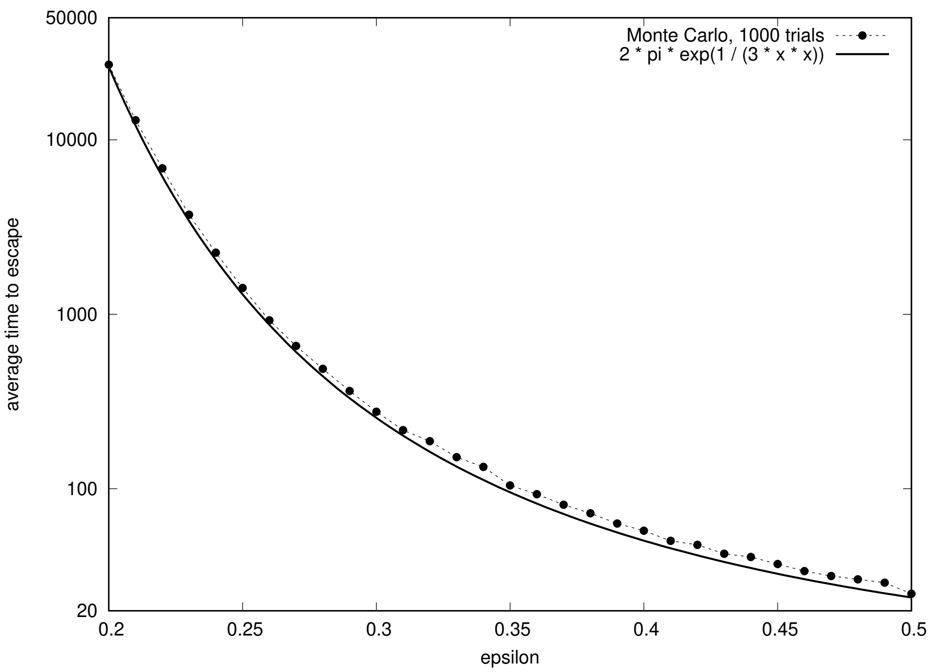


Here is a Python script that computes average time to escape by naive Monte Carlo, from 1000 trials (`for` loop over `m`), parameter ε is supplied from the command line:

```
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 ε:



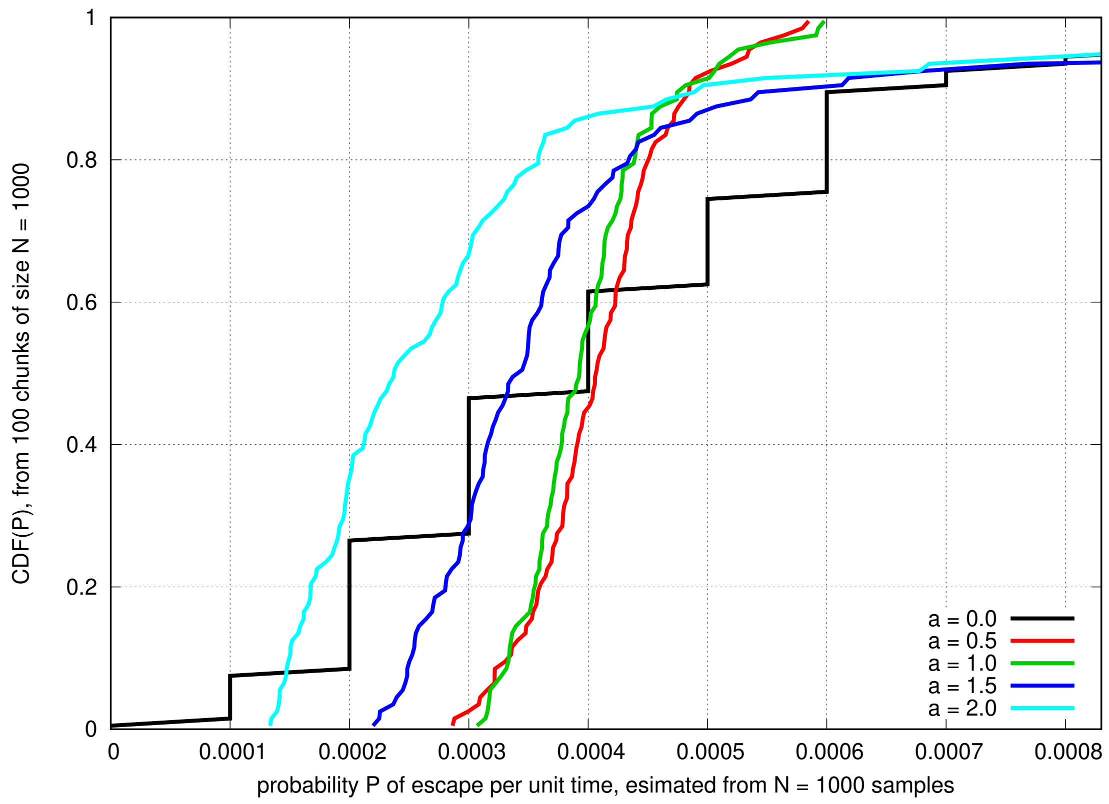


Let us apply the importance sampling technique in 2 [similar] ways:

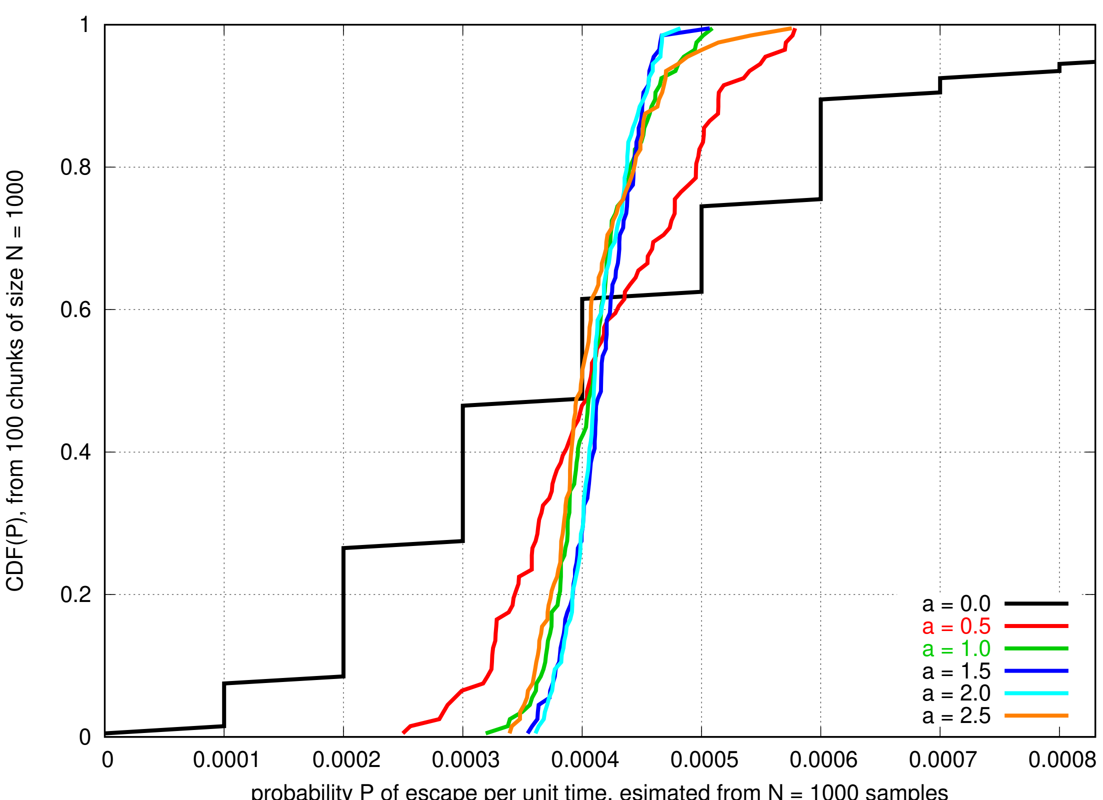
```
[...]/teaching/2020-1/math_575b/notes/Monte_Carlo/importance_sampling/ex_24.1.2$
 cat ex_24.1.2_IS1.py
from sys import argv; import numpy as np; from random import normalvariate
dt, eps, a, T = 0.01, float(argv[1]), float(argv[2]), 10.
sigma = np.sqrt(dt)
for m in range(0, 100000):
  t, x, compensation = 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 $x_{i+1} = x_i + v\tau + \sqrt{\tau}\zeta \ \ -\text{to}\rightarrow \ \ x_{i+1} = x_i + V\tau + \sqrt{\tau}\zeta$, and the compensation factor is `compensation` $= \exp\big((x_{i+1} - x_i - v\tau)^2/2\tau\big) \big/ \exp\big((x_{i+1} - x_i - V\tau)^2/2\tau\big)$.

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 $\mathrm{d}x/\mathrm{d}t$ is changed from $-x + \varepsilon x^2 \ \ -\text{to}\rightarrow \ \ -x + \varepsilon x^2 + p = x - \varepsilon x^2$.

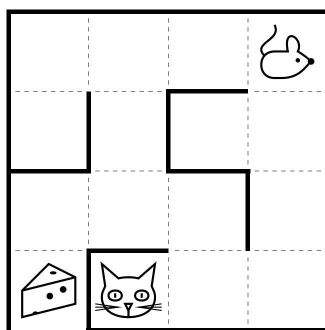


CDF(P), from 100 chunks of size N = 1000
probability P of escape per unit time, esimated from N = 1000 samples
0
0.0001
0.0002
0.0003
0.0004
0.0005
0.0006
0.0007
0.0008
0
0.2
0.4
0.6
0.8
1
a = 0.0
a = 0.5
a = 1.0
a = 1.5
a = 2.0



CDF(P), from 100 chunks of size N = 1000
probability P of escape per unit time, esimated from N = 1000 samples
0
0.0001
0.0002
0.0003
0.0004
0.0005
0.0006
0.0007
0.0008
0
0.2
0.4
0.6
0.8
1
a = 0.0
a = 0.5
a = 1.0
a = 1.5
a = 2.0
a = 2.5

Problems and exercises



1. A mouse, starting from the mouse icon, 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 the cheese icon (after that it escapes to the outside), or it is eaten by the cat at the cat icon. 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 $\mathsf{E}X^{20}$.

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 γ 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 γ 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 Mitropolis *et al.*[12] and generalized in 1970 by Hastings.[13] This is what is known as Metropolis–Hastings algorithm. The idea of the method is the following:

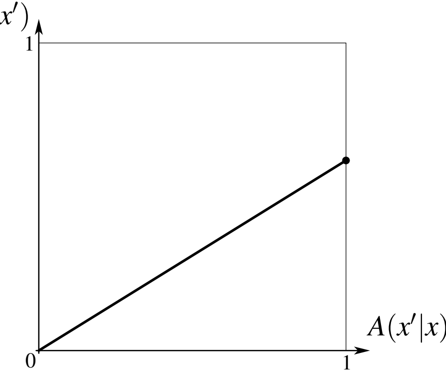
We want to sample a distribution with density $P(x)$. We would like to form a Markov chain whose stationary distribution is $P(x)$ by construction, and then simulate it. A simple way to ensure that $P(x)$ is indeed the stationary distribution of the Markov chain is making it so through detailed balance:

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

$$x \underset{A(x|x')\cdot T(x|x')}{\overset{A(x'|x)\cdot T(x'|x)}{\rightleftarrows}} x'$$

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

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



[12] 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).

[13] 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 $x \to x'$ we would accept it (otherwise we just stay at state x). Obviously, $0 \le A \le 1$, 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 $\big(A(x'|x), A(x|x')\big)$ should lie on the boundary of the square $[0,1]^2$. We set

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

Note that the acceptance probabilities A depend only on the ratio of 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 $A(x'|x) = \min\big(1, P(x')/P(x)\big)$.

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 $0 \le x < 1$, otherwise $P(x) = 0$. We choose $T(x'|x) = \exp\big(-(x'-x)^2/2\sigma^2\big)/\sqrt{2\pi}\sigma$, *i.e.*, $x' = x + \sigma \cdot \zeta$. In our MCMC simulation we always are going to have $0 \le x < 1$, so $P(x) = 1$. The acceptance factor is $A(x'|x) = 1$ if $0 \le x' < 1$, 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 $\left[\ \frac{1}{6}\ \frac{1}{6}\ \frac{1}{6}\ \frac{1}{6}\ \frac{1}{6}\ \frac{1}{6}\ \right]$.

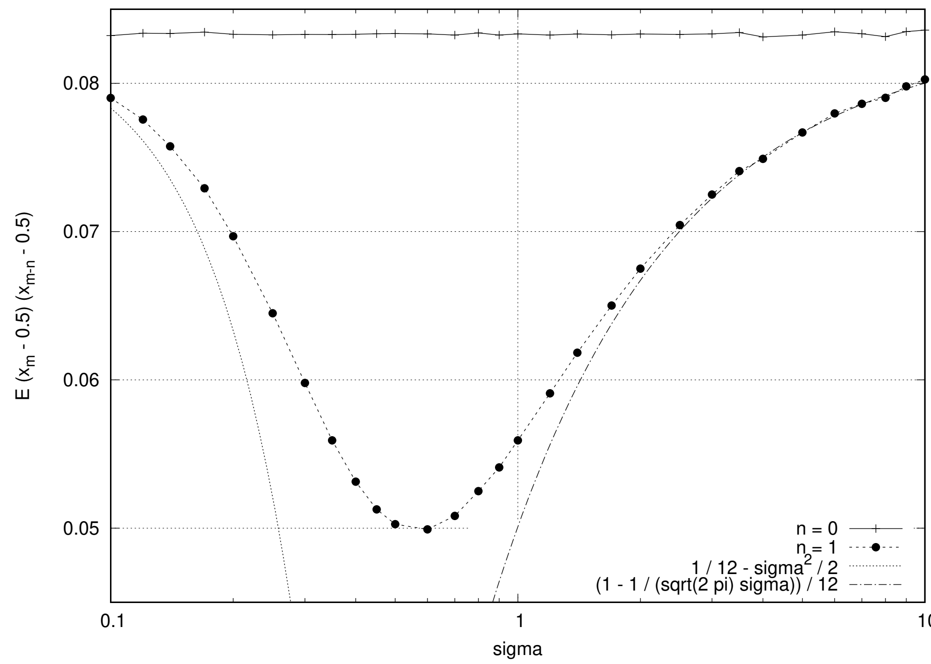
What value of σ is the best? We would like the equilibration of the MCMC to be the fastest, *i.e.*, if x_m is the Markov chain state at the mth time step, we would like x_{m-n} and x_m to be as independent as possible for even not so large n. One of the possible measures is the auto-correlation function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

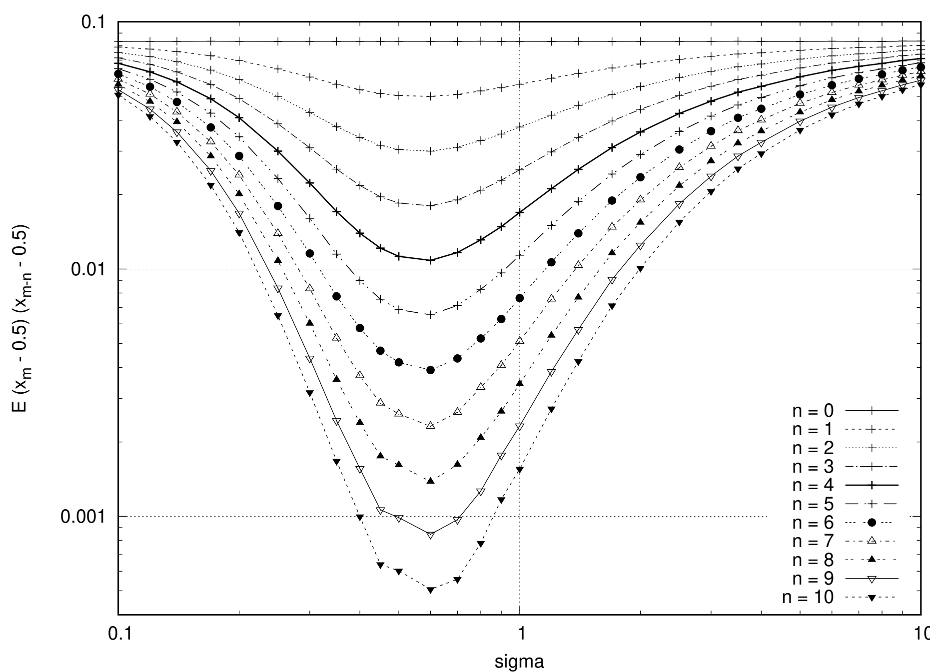
$$\mathsf{E}\big(x' - \tfrac{1}{2}\big)\big(x - \tfrac{1}{2}\big) = \int_0^1 \mathrm{d}x \left[\int_{-\infty}^{-x} \mathrm{d}\xi\, \big(x - \tfrac{1}{2}\big)^2 + \int_{-x}^{1-x} \mathrm{d}\xi\, \big(x + \xi - \tfrac{1}{2}\big)\big(x - \tfrac{1}{2}\big) + \int_{1-x}^{\infty} \mathrm{d}\xi\, \big(x - \tfrac{1}{2}\big)^2\right] \frac{\mathrm{e}^{-\xi^2/2\sigma^2}}{\sqrt{2\pi}\sigma}$$

$$= \int_0^1 \mathrm{d}x\, \big(x - \tfrac{1}{2}\big)^2 + \int_0^1 \mathrm{d}x\, \big(x - \tfrac{1}{2}\big) \int_{-x}^{1-x} \mathrm{d}\xi\, \xi\, \frac{\mathrm{e}^{-\xi^2/2\sigma^2}}{\sqrt{2\pi}\sigma} = \frac{1}{12} + \int_{-1}^{1} \mathrm{d}\xi\, \xi\, \frac{\mathrm{e}^{-\xi^2/2\sigma^2}}{\sqrt{2\pi}\sigma} \int_{\max(0,-\xi)}^{\min(1,1-\xi)} \mathrm{d}x\, \big(x - \tfrac{1}{2}\big)$$

$$= \frac{1}{12} + \int_{-1}^{1} \mathrm{d}\xi\, \xi\, \frac{\mathrm{e}^{-\xi^2/2\sigma^2}}{\sqrt{2\pi}\sigma} \left[\frac{\frac{1}{4} - (-\xi - \frac{1}{2})^2}{2} \chi_{\xi<0} + \frac{(\frac{1}{2} - \xi)^2 - \frac{1}{4}}{2} \chi_{\xi>0}\right]$$

$$= \frac{1}{12} + \int_{-1}^{1} \mathrm{d}\xi\, \xi\, \frac{\mathrm{e}^{-\xi^2/2\sigma^2}}{\sqrt{2\pi}\sigma} \left[\frac{\xi|\xi|}{2} - \frac{\xi}{2}\right] = \frac{1}{12} - \int_0^1 \mathrm{d}\xi\, \frac{\mathrm{e}^{-\xi^2/2\sigma^2}}{\sqrt{2\pi}\sigma}\, \xi^2\big(1 - \xi\big) = K(1)$$





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 is sample this distribution by Metropolis–Hastings algorithm in 2 different ways:

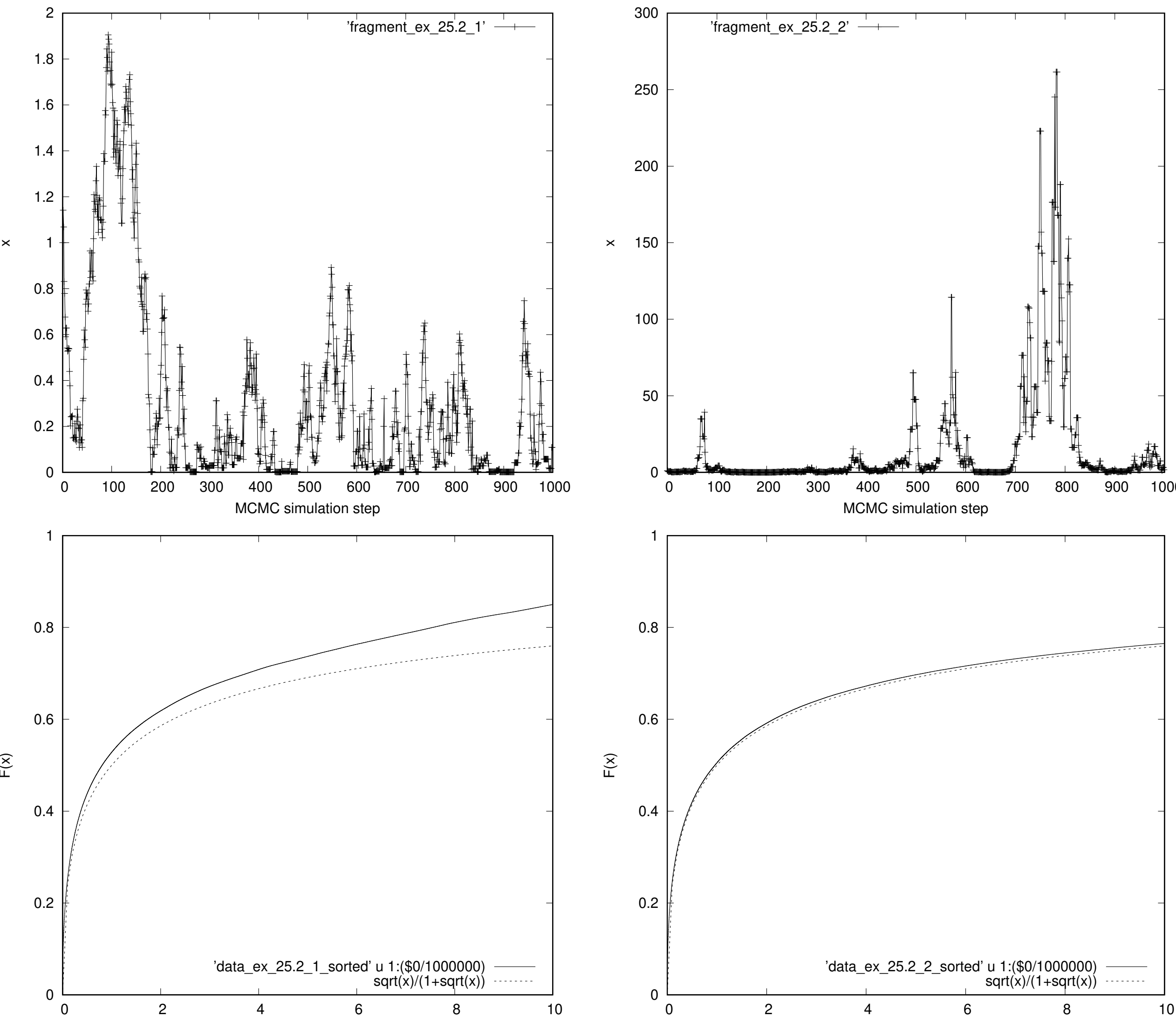
First, let us choose $T(x'|x) = \exp\bigl(-(x'-x)^2/2\sigma^2\bigr)/\sqrt{2\pi}\sigma$, with $\sigma = 0.1$:

```
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 $T(x'|x) = \exp\bigl(-(x'-x)^2/2x^2\bigr)/\sqrt{2\pi}x$. 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 $\mathsf{P}(x_1, x_2, ..., x_N)$. *Gibbs sampling* is a MCMC algorithm that starts from some initial $\boldsymbol{x}$, and then at each step we 1) randomly (or in some pre-defined order) choose a variable x_k from x_1, x_2, ..., x_N; and 2) set x_k according to the

conditional distribution $\mathsf{P}\big(x_k \big| x_1, x_2, \ldots, x_{k-1}, x_{k+1}, \ldots, x_N\big)$. Such sampling could be convenient when the conditional distribution of one (or not so many) variable(s) can be easily computed (while the whole $\mathsf{P}(\boldsymbol{x})$ 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 $\boldsymbol{n} = (n_1, n_2, \ldots, n_D)$ containing a binary variable $\sigma_{\boldsymbol{n}} = \pm 1$. The values of all the binary variables, a vector $\boldsymbol{\sigma}$, form a "state", and we introduce the following distribution over the states:

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

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 $L \to \infty$.) 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 $Z = \sum_i \mathrm{e}^{-\beta E_i} = \mathrm{e}^{-\beta F}$.

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

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

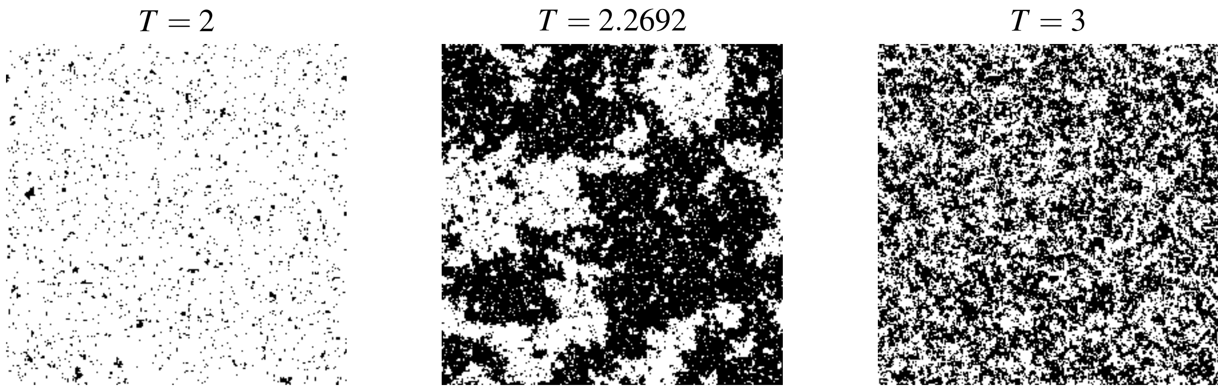
```
[...]/teaching/2020-1/math_575b/notes/Monte_Carlo/Ising_model$ ls
Ising.c
[...]/teaching/2020-1/math_575b/notes/Monte_Carlo/Ising_model$ cat Ising.c
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
#define L 256
#define FOR_ALL_SITES for (i = 0; i < L; i++) for (j = 0; j < L; j++)
int 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))); } }
```

[14] 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, attemps 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/abobe the phase transition)



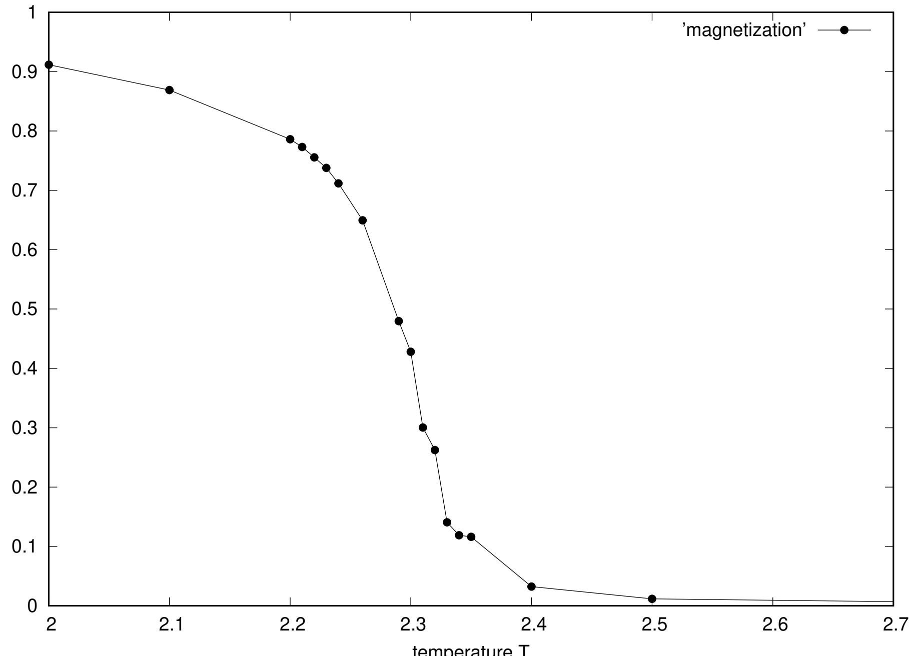
Of course, we would be interested in how fast the mixing within the state space is happening. We do not have any chance to visit a notable share of states, as the number of states is the $2^{\text{number of spin sites}}$. We visit a large number of states and hope that it is representative enough for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

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

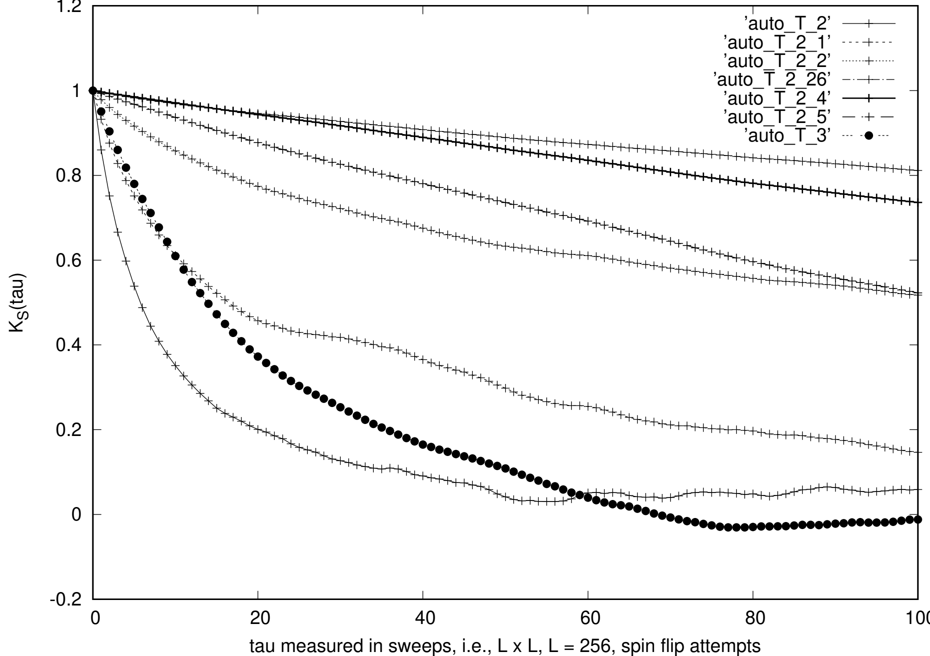
How fast $K_A(\tau)$ decays with τ is our estimation about how uncorrelated/independent are our samples of $\mathsf{P}(\boldsymbol{\sigma})$.

Let us choose the quantity $S = \sum_{\boldsymbol{n}} \sigma_{\boldsymbol{n}}$ — 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:

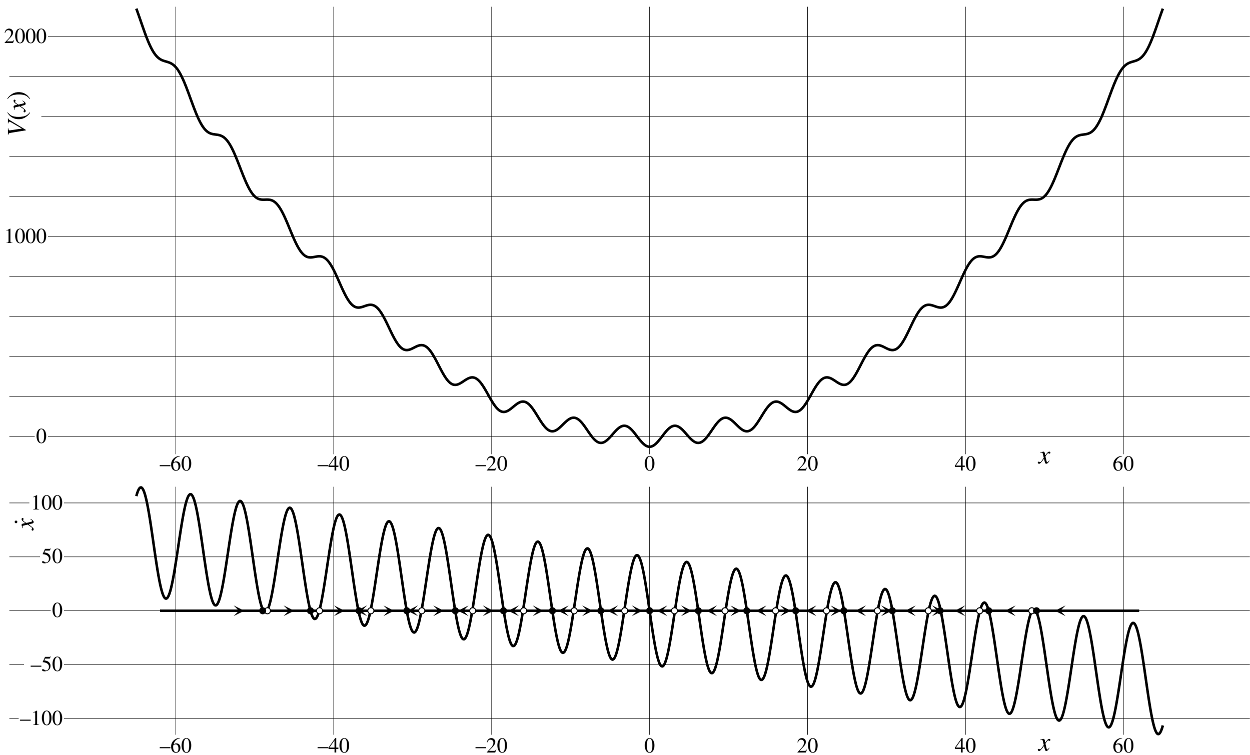


Here is how $K_S(\tau)$ falls down with τ:



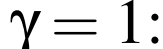
25.2 Stochastic optimization

Example 25.2.1: Consider $V(x) = x^2/2 - 50\cos x$. To minimize it we can try the gradient descent method, $\mathrm{d}x/\mathrm{d}t = -\mathrm{d}V(x)/\mathrm{d}x = -x - 50\sin x$.

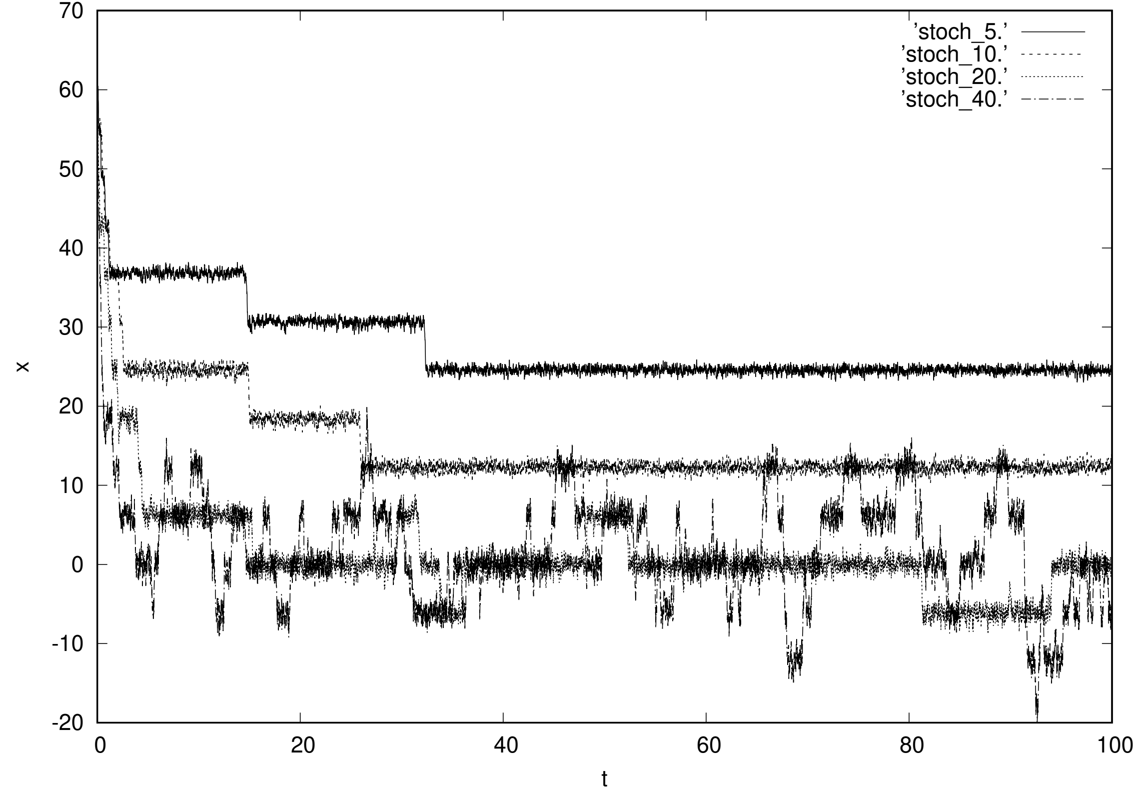


Here is a Python script that updates x according to the rule $x(t+\mathrm{d}t) := x(t) - \mathrm{d}t \cdot V'(x) + \sqrt{2D\mathrm{d}t}\zeta$, where $\zeta \sim N(0,1)$. The case $D = 0$ would correspond to standard gradient descent (its ODE formulation with forward Euler method). When $D > 0$, but $\gamma = 1$, 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), $D(t+\mathrm{d}t) := \gamma \cdot D(t)$.

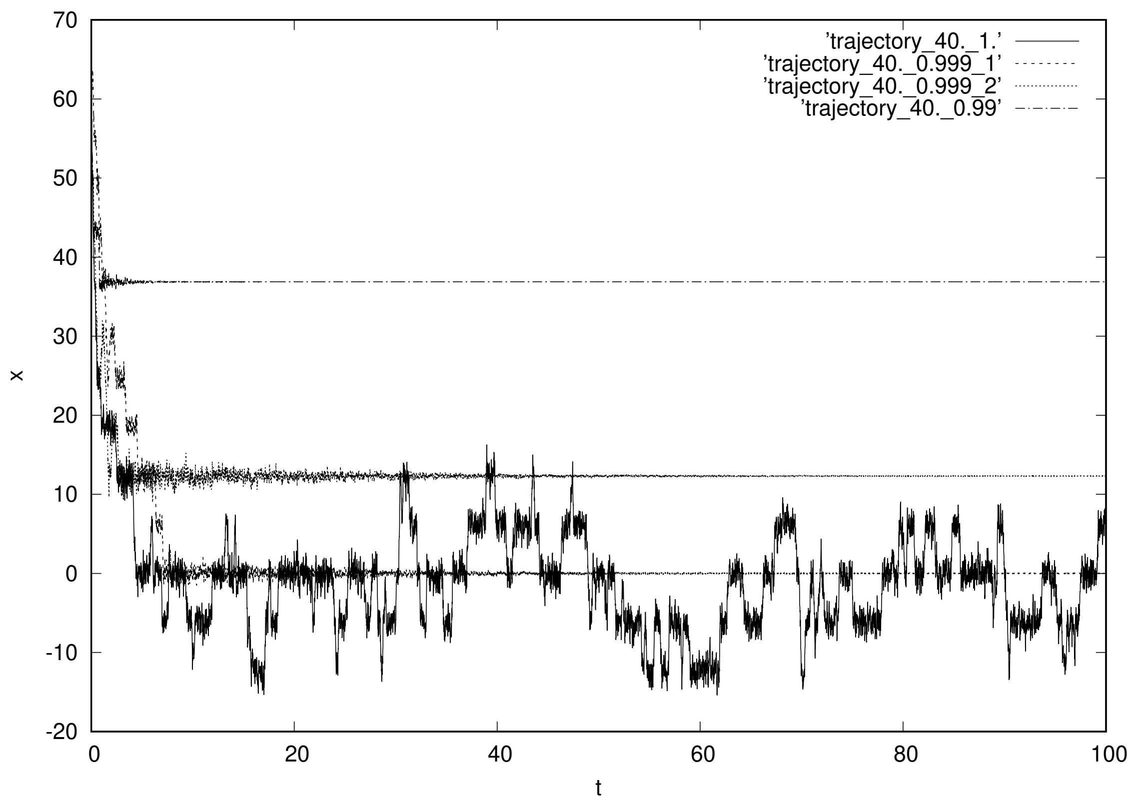
```
from sys import argv
from math import cos, sin, sqrt
from random import normalvariate
def f(x):
  return 0.5 * x**2 - 50 * cos(x)
def df(x):
  return x + 50 * sin(x)

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)
```



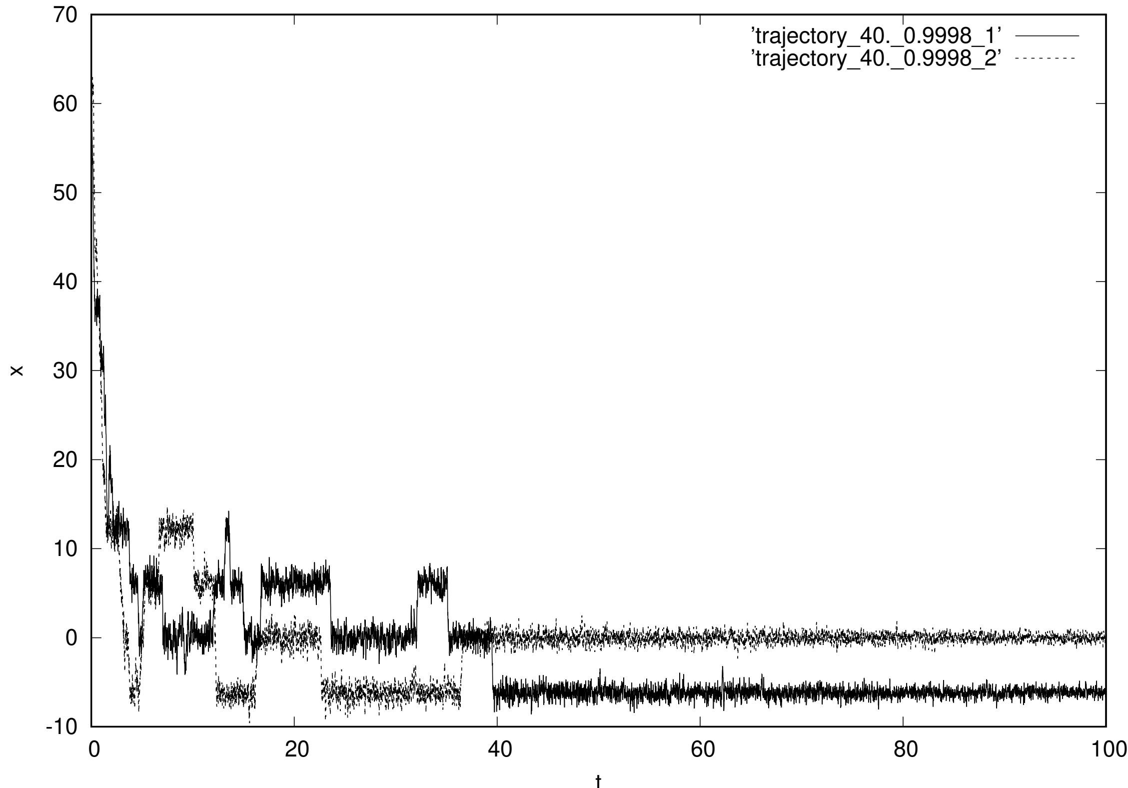
$\gamma = 1$:



various γ's:



$\gamma = 0.9998$:



Problems and exercises

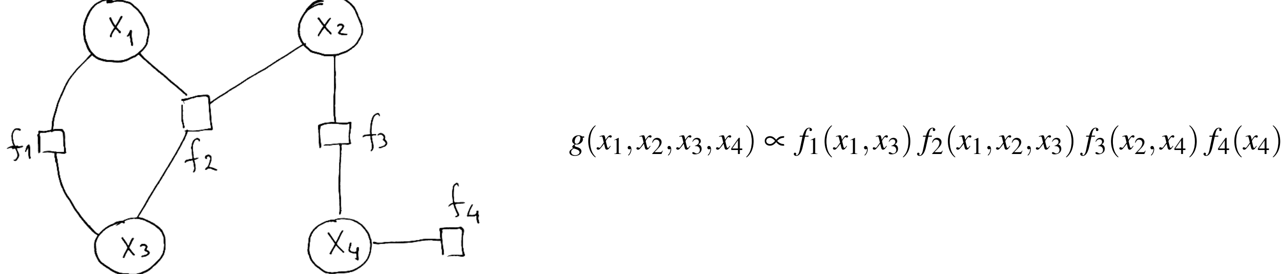
1. Using Metropolis algorithm, simulate a Markov chain with $P(x) = \exp(-x)$, $x \geq 0$ density and transition function $T(x'|x) = \exp\left(-(x'-x)^2/2\sigma^2\right)/\sqrt{2\pi}\sigma$ (with appropriate acceptance probability $A(x'|x)$). Find σ that minimizes $K_{xx}(1) = \mathsf{E}(x_m - 1)(x_{m-1} - 1)$, 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 $m(T) = \left|\sum_{\boldsymbol{n}} \sigma_{\boldsymbol{n}}\right|/L^2$ for $L = 32$, 64, and 128. Is the transition from non-zero to [almost] zero value of $m(T)$ becomes sharper for larger L?

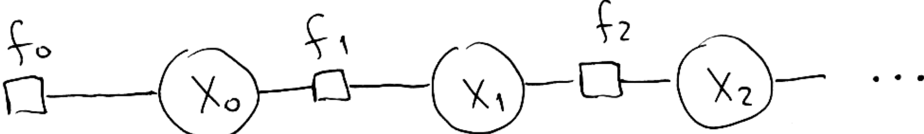
26 Message passing/belief propagation

A factor graph (X, F, E) is a bipartite graph, whose vertices are separated into two groups: variable vertices $X = (x_1, x_2, ..., x_N)$ and facror vertices $F = (f_1, f_2, ..., f_m)$. For each factor vertex $1 \leq \alpha \leq M$ let us consider a subset of varible vertices $X_\alpha \subseteq X$ that are connected to f_α by an edge. Similarly, for each $1 \leq i \leq N$ let F_i be the subset of factor vertices that are connedted to i. [15] 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 $\sum_X g(X) = 1$.

[15] A similar structure would be a hypergraph, where [hyper]edges could connect an arbitrary subset of vertices.



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 $\int \mathrm{d}x_t \;\, f_t(x_{t-1},x_t) = 1$ (or a constant not depending on x_{t-1}). Inegrating over $x_{t+1}x_{t+2}x_{t+3}\ldots$ we would get the marginal distribution $g(x_0,x_1,\ldots,x_t) \propto f_0(x_0)f_1(x_0,x_1)f_2(x_1,x_2)\ldots f_t(x_{t-1},x_t)$.

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[16] 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 ($\mu^{(-1)}_{i\to\alpha} := 1$)

$$\mu^{(t)}_{\alpha\to i}(x_i) := \sum_{x_{X_\alpha\setminus i}} f_\alpha(X_\alpha) \prod_{j\in X_\alpha\setminus i} \mu^{(t-1)}_{j\to\alpha}(x_j), \qquad \mu^{(t)}_{i\to\alpha}(x_i) := \prod_{\beta\in F_i\setminus\alpha} \mu^{(t)}_{\beta\to i}(x_i)$$

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

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

$$g(X) \propto \prod_\alpha f_\alpha(x_{X_\alpha}) = \prod_\alpha \frac{g_{X_\alpha}(x_{X_\alpha})}{\prod\limits_{i\in X_\alpha} \mu_{i\to\alpha}(x_i)} = \frac{\prod\limits_\alpha g_{X_\alpha}(x_{X_\alpha})}{\prod\limits_{i-\alpha} \dfrac{g_i(x_i)}{\mu_{\alpha\to i}(x_i)}} = \frac{\prod\limits_\alpha g_{X_\alpha}(x_{X_\alpha})}{\prod\limits_i \Big(g_i(x_i)\Big)^{|F_i|-1}}$$

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.

[16] 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 $p \to 3p^2$ 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 mttaer 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

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

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

In the case of x_i, $1 \le i \le N$, 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 $m_i = (1/2)\ln\big(g_i(0)/g_i(1)\big)$. The message passing becomes (here $\eta := (1/2)\ln\big(\mu(0)/\mu(1)\big)$)[17]

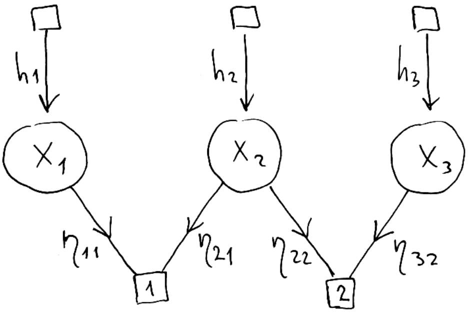
$$\eta^{(t)}_{\alpha\to i} = \frac{1}{2}\ln\left(\frac{\displaystyle\sum_{X_\alpha,\text{parity},x_i=0}\ \prod_{j\in X_\alpha\setminus i}\mu^{(t-1)}_{j\to\alpha}(x_j)}{\displaystyle\sum_{X_\alpha,\text{parity},x_i=1}\ \prod_{j\in X_\alpha\setminus i}\mu^{(t-1)}_{j\to\alpha}(x_j)}\right), \qquad \eta^{(t)}_{i\to\alpha}(x_i) := h_i + \sum_{\beta\in F_i\setminus\alpha}\eta^{(t)}_{\beta\to i}, \qquad m^{(t)}_i := h_i + \sum_{\alpha\in F_i}\eta^{(t)}_{\alpha\to i}$$

$$\eta^{(t)}_{\alpha\to i} = \operatorname{arctanh}\big((\Sigma_0-\Sigma_1)/(\Sigma_0+\Sigma_1)\big) = \operatorname{arctanh}\left(\prod_{j\in X_\alpha\setminus i}\tanh\eta^{(t)}_{j\to\alpha}\right)$$

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

Example 26.1: Consider a repetition code

[17] $\operatorname{arctanh}(x) = (1/2)\ln\big((1+x)/(1-x)\big)$



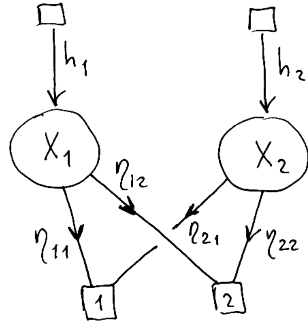
The numbers h_1, h_2, and h_3 are logarithmic likelihoods from the output of the communication channel. Positive/negative value of h means that 0/1 is more probable. The magnitude of h is corresponding to how much more probable.

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 arctanh(·) functions because the product of tanh(·)'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 $\eta_{11}^{(t)} = h_1 + \eta_{22}^{(t-1)}$, $\eta_{12} \hookleftarrow h_1 + \eta_{21}$, $\eta_{21} \hookleftarrow h_2 + \eta_{12}$, and $\eta_{22} \hookleftarrow h_2 + \eta_{11}$, 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, $\eta_{11} + \eta_{22} \hookleftarrow h_1 + h_2 + \eta_{11} + \eta_{22}$, 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 $(\boldsymbol{x}_i, y_i)$, and we want to come up with a function $f(\boldsymbol{x})$ such that $f(\boldsymbol{x}_i) \approx y_i$.

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

Let us say, we decide to choose our function approximating the data from a family of functions $f(\boldsymbol{x}, \boldsymbol{\theta})$, 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

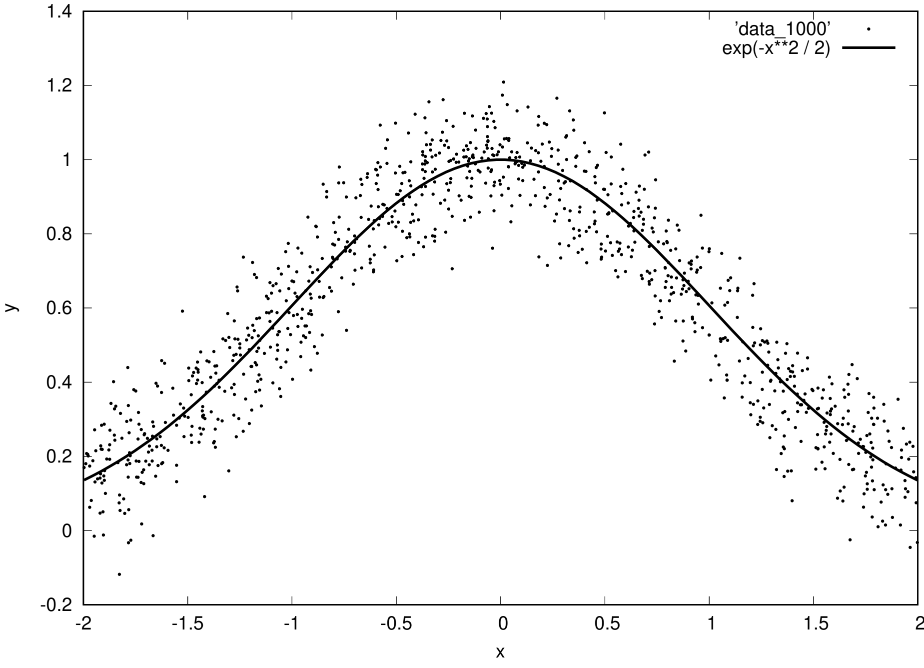
$$\boldsymbol{\theta}_{\text{ML}} := \arg\max_{\boldsymbol{\theta}} \underbrace{\prod_{i=1}^{N} P\big(y_i \,\big|\, f(\boldsymbol{x}_i, \boldsymbol{\theta})\big)}_{\text{likelihood}}, \qquad \boldsymbol{\theta}_{\text{ML}} = \arg\min_{\boldsymbol{\theta}} \underbrace{\sum_{i=1}^{N} \big(y_i - f(\boldsymbol{x}_i,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

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

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

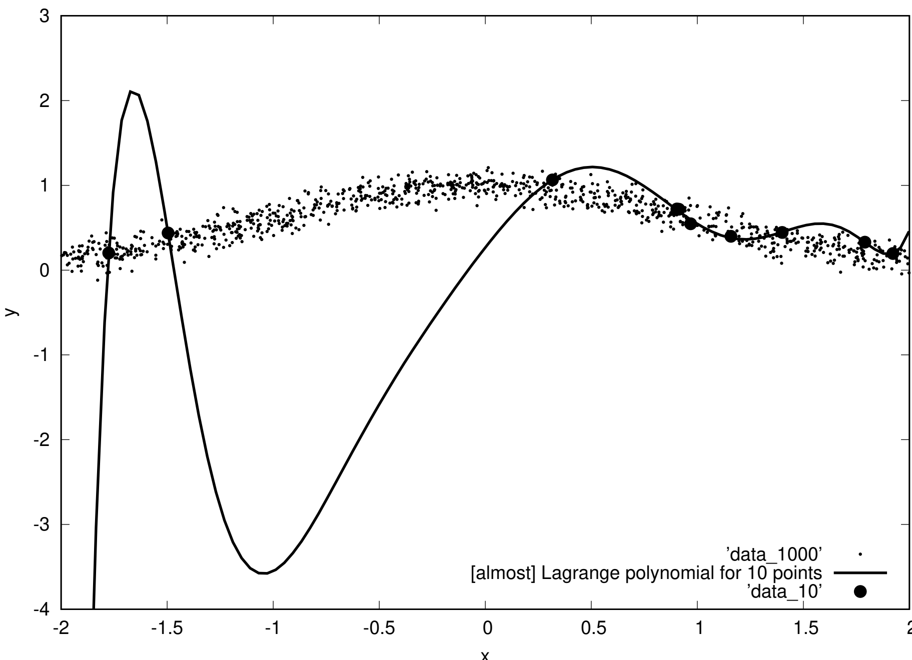
Example 27.1: Consider data being generated by a Python script

```
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 9th degree polynomial $f(x,\boldsymbol{\theta}) = \sum_{m=0}^{9} \theta_m x^m$:

```
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 $f(x, \boldsymbol{\theta})$ is going well through the points that were used in estimation of $\boldsymbol{\theta}$ (*training data*), but does not *generalize* well to new data.

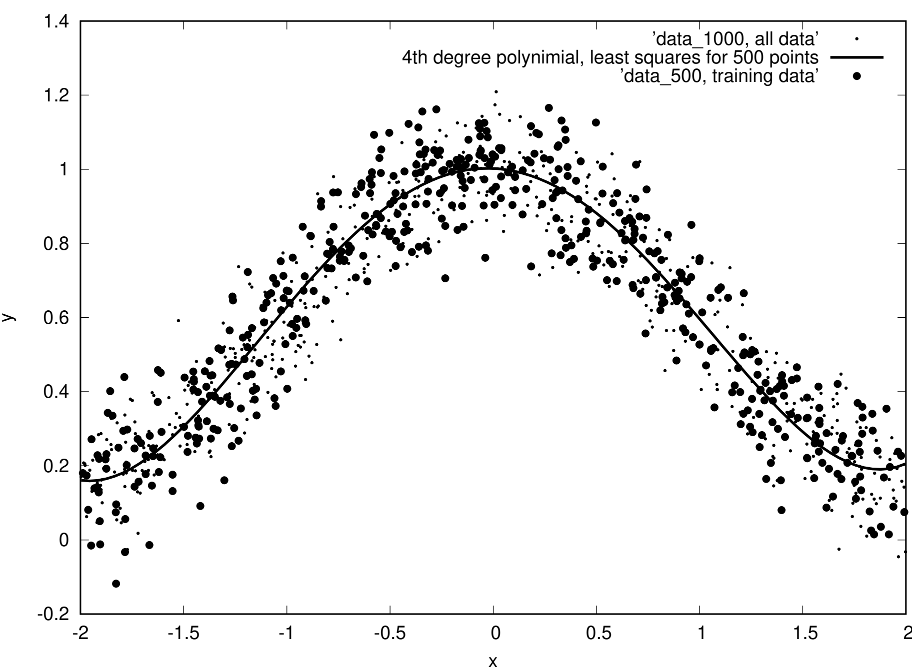
Here is an example of an application of another model to the data, $f(x, \boldsymbol{\theta}) = \theta_0 \exp\big(-(x-\theta_1)^2/2\theta_2^2\big)$. This family of functions contains $f_{\text{true}}(x) = \exp(-x^2/2)$ for $\theta_0 = 1$, $\theta_1 = 0$, and $\theta_2 = 1$. Here [non-linear] regression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

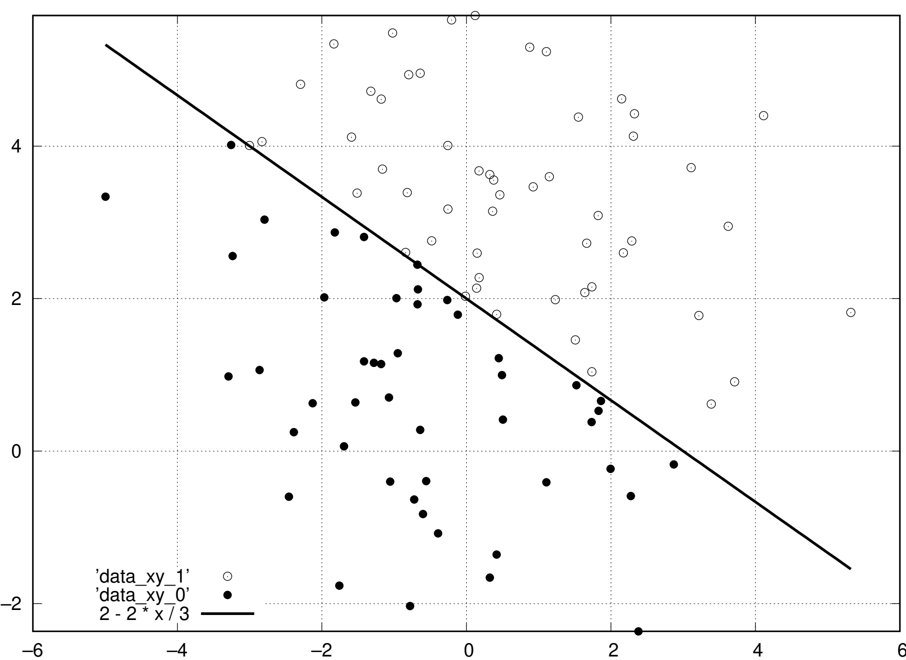
```
from math import sin, pi; from random import seed, random, normalvariate
seed(0)
for i in range(0, 1000):
  x = 2. * pi * random() - pi
  print(x, sin(x) + normalvariate(0., 0.1))
```

We would like to fit the data by the Mth degree polynomial $f(x,\boldsymbol{\theta}) = \sum_{m=0}^{M} \theta_m x^m$. Let the loss function per data point be $\frac{1}{\text{number of data points}} \sum_{i \text{ in data}} \big(y_i - f(x_i,\boldsymbol{\theta})\big)^2$. 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 $\boldsymbol{x} \in \mathbf{R}^D$. 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 $f : \mathbf{R}^D \to L$.

Example 28: Consider the points on (x,y)-plane. The points differ in whether they are above (label "1") of 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.[18] 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$
```

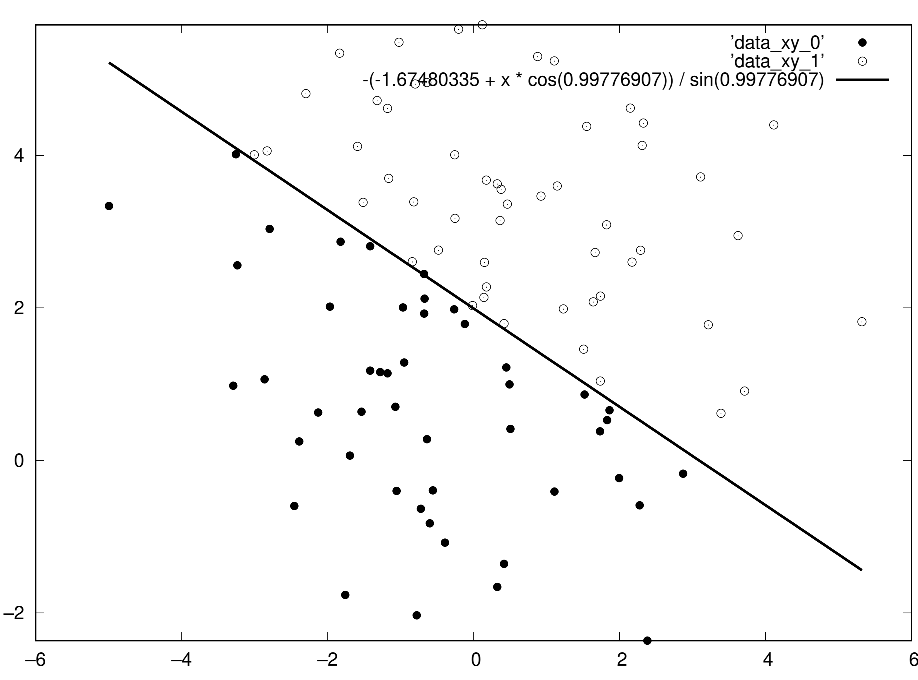
Here we try the following change $\theta_0 + \theta_1 x + \theta_2 y \longrightarrow \theta_0 + \cos(\theta_1)x + \sin(\theta_1)y$:

```
[...]/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$
```

[18] 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 $f_{\mathrm{true}}(x,y) \to \{0,\}$, 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 $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, ..., $\boldsymbol{x}_N \in \mathbf{R}^D$ (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 algoritgm 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:

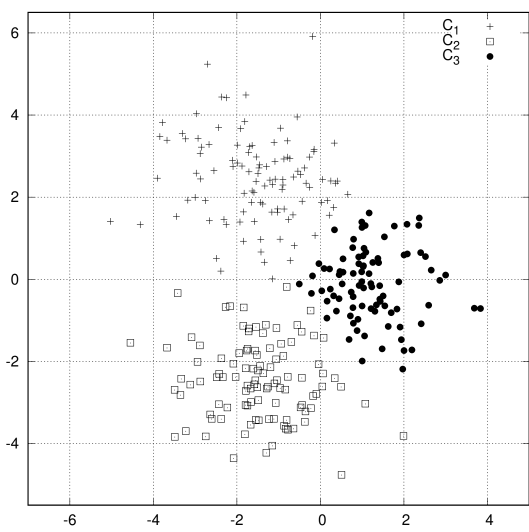
- hierarchical clustering
  - agglomerative: Initially clusters are data points. Two "closest" points/clusters are found and then merged, with clusters closeness measure redefined/updated.
  - divisive: The whole data is divided/cut into two [large] groups, which are further processed.
- setting clustering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

$$k\text{-means}: \qquad \underset{C_1, C_2, \ldots, C_k}{\arg\min} \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in C_i} \|\boldsymbol{x} - \boldsymbol{\mu}_i\|^2, \qquad \boldsymbol{\mu}_i := \frac{1}{|C_i|} \sum_{\boldsymbol{x} \in C_i} \boldsymbol{x}$$

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

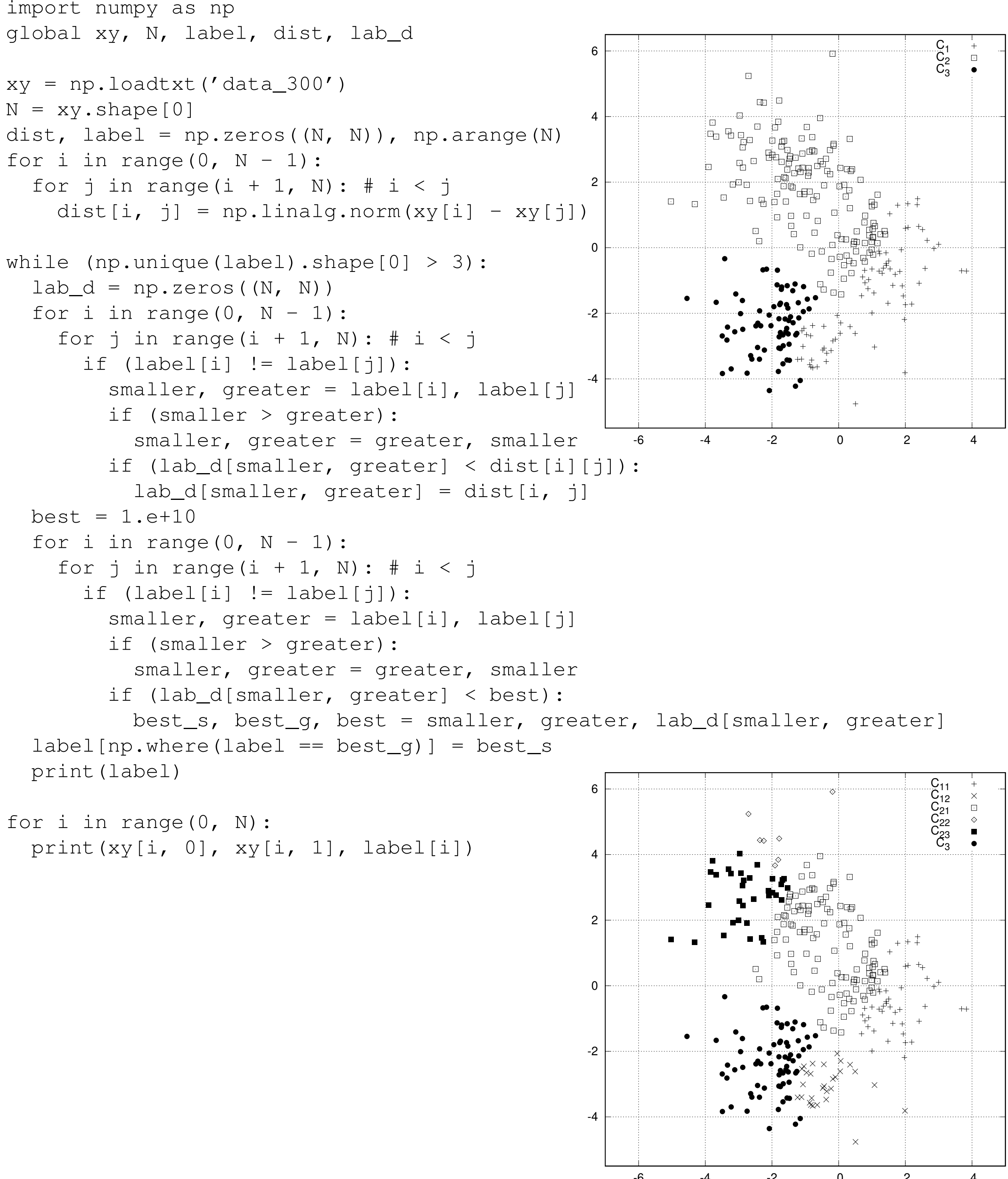
```
import numpy as np
global xy, mu, label

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*, 7th ed. (Cambridge U. Press, 2009).

[Cal20] Jeff Calder, *The Calculus of Variations*.

[TrBa97] L. N. Trefethen, D. Bau III, *Numerical linear algebra* (SIAM, 1997).