



Електротехнички факултет  
Универзитет у Београду



# Инжењерски оптимизациони алгоритми

др Драган Олћан, в. проф. ([olcan@etf.rs](mailto:olcan@etf.rs))  
Јована Петровић, асистент ([jovanap@etf.rs](mailto:jovanap@etf.rs))  
Дарко Нинковић, асистент ([darko@etf.rs](mailto:darko@etf.rs))

Изборни предмет  
Школска 2020/21. година

<http://mtt.etf.rs/si/ioa.htm>

# Који су циљеви и исходи овог курса?

- Упознавање са основним класама оптимизационих алгоритама који се користе у инжењерству и ИТ струци
- Оспособљавање за практичну примену оптимизационих алгоритама при решавању инжењерских проблема
- Пројектовање се данас ради практично искључиво коришћењем рачунара
- Практично решавање инжењерских проблема по правилу захтева употребу оптимизације
- Познавање оптимизационих алгоритама је од изузетне важности [као и спознаја када и како их (не) користити]

# Зашто курс о инжењерским оптимизационим алгоритмима?

- Познавање модерних алгоритама за оптимизације је **неопходно** за све који се баве: **инжењерством, науком и бизнисом**
- Поред алгоритама, сагледаћемо **могућности за решавање** проблема данас
- Зашто једноставно не узмем рутину из неке од готових софтверских библиотека или постојећег софтвера?
  - Познавање и разумевање алгоритама је од изузетног значаја, чак и када се користе готова решења
  - Често је случај да постојећи софтвер није оптимално решење за наш проблем

# Шта је градиво курса?

- **Увод.** Преглед појмова и представљање основне теорије решавања система нелинеарних једначина на које се своде оптимизациони алгоритми у инжењерству
- **Систематизација.** Поделе оптимизационих алгоритама
- **Оптимизациони алгоритми:**
  - систематско претраживање (енглески: systematic search)
  - случајно претраживање (енглески: random search)
  - градијентна метода (енглески: gradient method)
  - симплекс алгоритам (енглески: Nelder-Mead simplex)
  - Данцигов симплекс алгоритам (енглески: Dantzig simplex)
  - симулирано каљење (енглески: simulated annealing)
  - генетички алгоритам (енглески: genetic algorithm)
  - кретање јата (енглески: particle swarm optimization)
  - диференцијална еволуција (енглески: differential evolution)
- **Оптимизација са више критеријума.** Парето фронт и његово одређивање коришћењем оптимизационих алгоритама
- **Рад на рачунару.** Сагледавање особина и параметара оптимизационих алгоритама који су од значаја за практичну примену кроз програмирање и симулације на рачунару

# Литература

- Литература из оптимизационих алгоритама је изузетно **обимна**
- Употреба оптимизационих алгоритама је изузетно распространена
- Избрани су они алгоритми (или класе алгоритама) који се данас најчешће употребљавају у инжењерској пракси
- Нагласак је на **алгоритмима широке намене** специјализовани алгоритми само као примери (нпр: quick-sort, Dijkstra's algorithm, Chess Master...)

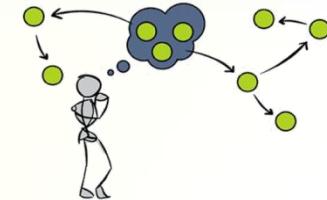


# Како је организован курс?

- Предавања:  
30 часова предавања + 30 часова вежби +15 часова лабораторије
- Литература
  - Материјал са предавања (PDF за сваку седмицу, линк најсајту предмета)
  - Z. Michalewicz, D.B. Fogel, *How to Solve It: Modern Heuristics*, Springer, 2004
  - Xin-She Yang, *Engineering Optimization An Introduction with Metaheuristic Applications*, University of Cambridge, Department of Engineering, Cambridge, United Kingdom, Wiley 2010
  - D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Professional, 1989
- Оцењивање
  - **Предиспитне обавезе** (задаци на вежбама + бонуси) – одсеца се на 70 поена
    - **Задатак са вежби** се брани на вежбама у лабораторији
    - **Бонуси** до максимално 10 поена
  - **Испит** (задатак или задаци), највише 40 поена, одсеца се на 30 поена
  - **Коначна оцена** – укупан број поена се добија сабирањем поена добијених на основу предиспитних обавеза и испита. За полагање испита неопходно је освојити бар 51 поен. Оцене 6-10 су равномерно расподељене у опсегу од 51 до 100 поена
- Распоред (термини предавања и вежби):  
предавања петак 12h-14h и вежбе 14h-16h @ MS Teams

# Како се изводе предавања и вежбе?

- Предавања:
  - слајдови
  - табла & креда
  - рачунар за илустрације и показе
- Вежбе:
  - **Ви радите на рачунару**
  - програмирање: C/C++ и Python  
битно је да програм (про)ради!
  - коришћење постојећег софтвера
- Наставници и сарадници задржавају право да, уколико посумњају или утврде да се студент не придржава правила "наставе на даљину", као и осталих норми понашања које важе током студирања на ЕТФу,
  - захтевају додатне детаље у вези са решењима задатака, у електронском облику,
  - захтевају да студент одговори на додатна питања, у електронском облику, или
  - не додељују поене на предиспитним обавезама током "наставе на даљину".



# Предиспитне обавезе током епидемије\*

- Задатак се задаје на крају предавања и важи до рока за предају (~ 6 дана)
- Током вежби решавате задатак и можете да питате асистента детаље око задатка (MS Teams канал)
- Решење задатка се састоји од
  - ASCII (TXT) фајла са јасно записаним (нумеричким) одговорима на питања из задатка
  - Кодом за решење задатка (C/C++ за Visual Studio 2017 или Python 3.7)
- Решење се шаље као **један ZIP фајл** (< 10 MB), кроз портал на сајту
- Решења се достављају најкасније до **четвртка** следеће седмице у **12:00**
- Освојени поени објављују се **четвртком до 21:00** на сајту предмета (структура поена за задатак биће изложена на одговарајућим вежбама)
- **Петком од 11:15 до 12:00** су показне лабораторијске вежбе где приказујемо званично решење задатка од прошле седмице (и евентуално решавамо примедбе на број освојених поена)

\* У случају побољшања епидемиолошке ситуације, лабораторијске вежбе се одржавају у истом термину у Лабораторији 64б

<http://mtt.etf.rs/si/IOA/up.html>

# Историја курса и примена у индустрији

## WIPL-D Optimizer

WIPL-D Optimizer is a powerful multi-algorithm optimization tool that is being used by many successful professionals around the world. The tool calculates a single solution as well as multiple solutions for complex criteria optimizations. Thanks to its simple and intuitive graphical interface, you can quickly solve the problem at hand. It enables a high level of design automation for antenna, antenna system, scatterer, or a microwave

WIPL-D Optimizer is seamlessly integrated with [WIPL-D Pro](#) and [WIPL-D Microwave](#). When you create your project, you want to get optimum performance, you just start the Optimizer from within the design environment, select the parameters to be optimized and the optimization criteria, and let the tool do the rest. You can specify cost-function based on virtually all the EM simulation results that are calculated with WIPL-D 3D EM solver, as well as the simulation results from WIPL-D Microwave.

Various optimization algorithms are available. The set covers all major methods proven to work efficiently in practice. The available optimization algorithms are:

- Particle Swarm
- Genetic
- Simplex
- Random
- Systematic Search
- Simulated Annealing
- Gradient

Some of the special options that distinguish WIPL-D Optimizer from other commercially available optimization tools:

- Hybrid optimization (involving two consecutive optimization algorithms)
- Pareto fronts (set of the best compromises in a multi-criteria optimization)
- Estimation of local minima (keeping several solutions that are in some proximity of the best found solution)
- Optimization repetitions (multiple optimizations from random starting points - decreases the probability of finding a local optimum which is not in fact the global optimum)

Optimizer Method 1: Random  
Max. Number of Iterations for Method 1: 100000  
Optimizer Method 2: None  
Max. Number of Iterations for Method 2: 300  
Current Iteration of Method 1: 21  
Last Iteration Cost = 1.17136e+002  
Current Iteration of Method 2: 0  
Total Solver Runs = 21

Symbol	Current Value	Lowest Value	Highest Value	The Best Value
reflector_din	2.00000e+000	None	None	2.00000e+000
frq_start	3.00000e+008	None	None	3.00000e+008
frq_stop	3.00000e+008	None	None	3.00000e+008
frq_step	0.00000e+000	None	None	0.00000e+000
<input checked="" type="checkbox"/> radius_1	5.64511e-002	2.00000e-002	5.00000e-001	1.64514e-001
<input checked="" type="checkbox"/> radius_2	1.13313e-001	2.00000e-002	5.00000e-001	6.59243e-002
<input checked="" type="checkbox"/> pitchang_1	5.96510e+000	5.00000e-001	1.50000e+001	3.52690e+000
<input checked="" type="checkbox"/> pitchang_2	2.06696e+000	5.00000e-001	1.50000e+001	3.42859e+000

Criterion | Weight  
helix | 0

Run | Exit | Reset | About

Simple Example: Optimized horn antenna

The open end of a rectangular waveguide is a source of electromagnetic waves, but it also represents a discontinuity creation of unwanted higher modes. A horn is added at the open end to reduce the magnetic waves at this discontinuity and to diminish the higher modes.

Let us assume that the length of the horn is specified. In that case, we can only change the surface of the open end of the horn in order to achieve greater gain. But, how can we achieve the best possible gain? On one hand, increase of the horn's aperture leads to decrease of the antenna's gain due to changes in the distribution of phase at the opening. On the other hand, by increasing the surface of the opening, the physical surface of the antenna is being increased and so is its effective surface. This leads to the augmentation of gain.

Let's say that we want to achieve the gain along the axis of the waveguide greater than 20 dB, and we wonder how large the surface of the opening needs to be. We then set the optimization criteria and the range for height and width of the horn and then we run the WIPL-D Optimizer.

When the optimization procedure finishes, we get the optimum dimensions of the horn in just a few seconds.

Automated Design of a Waveguide Filter

In which you can see the possibility of optimization-based design of a microwave circuit illustrate an effective procedure for designing waveguide filters consisting of series of coupled initial guess. The coupling is performed by using arbitrary waveguide discontinuities (e.g.,

7-pole filter were -60 dB to 7.7 GHz and -0.3 dB between 7.8 and 8.4 GHz. The standard rectangular IEC-R84 (WR-112) e 28.5 mm width (A) and the dielectric is air and the metal is copper. The model is made using analytical models of rectangular H double step components in WIPL-D-Microwave Pro environment.

The obtained filter mostly satisfies the desired characteristic. Agreement in stopband was excellent (60 dB were obtained) and insertion loss was almost (up to 0.38 dB with losses included) in the range requested. WIPL-D software enables fast and practical waveguide filter design. Even when there is only a theoretical image of solution, WIPL-D Microwave and WIPL-D Optimizer can be used to find the filter that satisfies the given criteria.

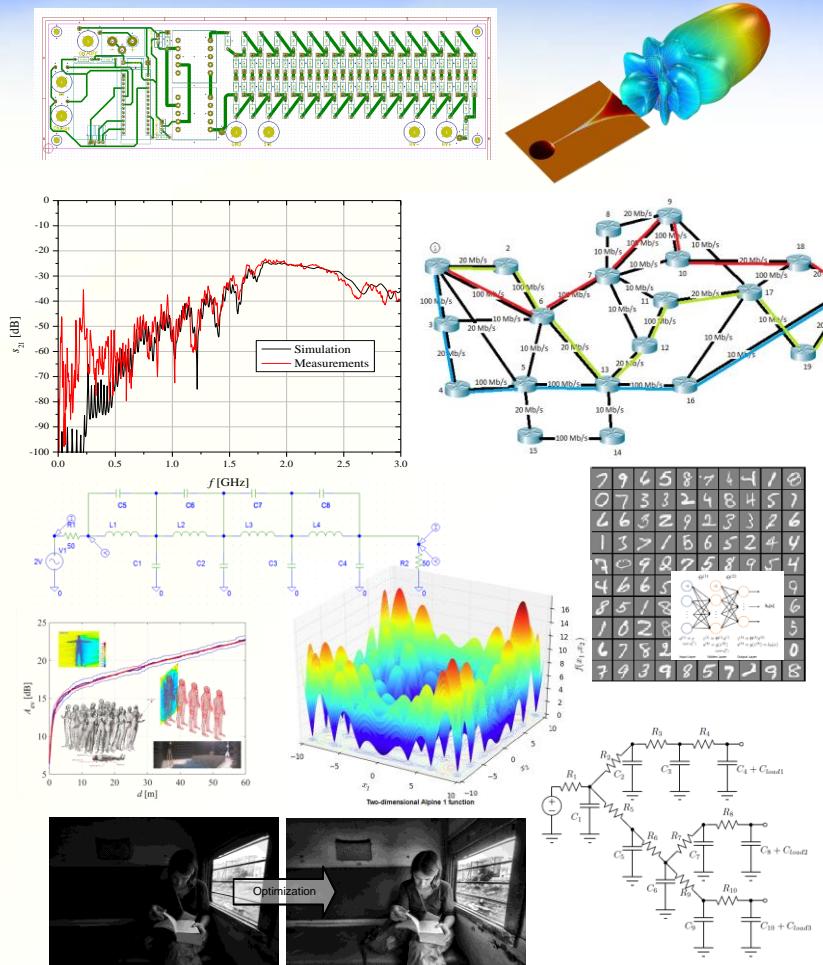
S21 parameter before and after optimization

Final S11 and S21 - optimized filter

Copyright © 2013 WIPL-D d.o.o. All rights reserved.

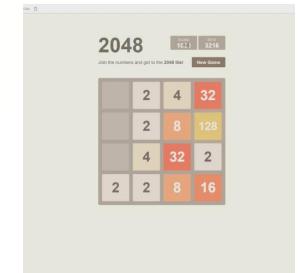
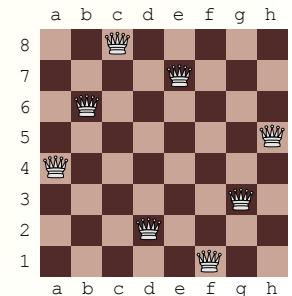
# Примери оптимизације из електротехнике и рачунарства

- Пројектовати електрично коло са одговарајућим особинама (преносне функције филтра, појачање појачавача, израда елемената кола у различитим технологијама, Q-фактор калема итд.)
- Пројектовање антене за задати дијаграм зрачења
- Проналажење оптималног рутирања у мрежама
- Проналажење оптималних кодова за пренос и запис према различитим критеријумима
- Оптимално покривање подручја радио сигналом
- Оптимално искоришћење фреквенцијског спектра
- Минимизација површине чипа и максимизације његових рачунарских перформанси
- Минимизација времена извршавања програма
- Проналажење функције која оптимално фитује задати скуп података
- Минимизација потрошње електричне енергије уређаја
- Максимизација капацитета батерије
- ...



# Други примери оптимизације

- Оптималан избор школе
- Оптималан избор изборног предмета на студијама
- Оптималан избор (будућег) радног места
- Оптимално коришћење свог и туђег времена
- Оптимална расподела економских ресурса
- Најбржа путања за задату полазну и крајњу тачку
- Избор оптималног потеза у игри
- Максимизација учинка радника
- Минимизација цене производње
- Оптималан избор тима за пројекат
- Оптимална расподела информација за постизање жељеног циља
- ...



# Како је оптимизација повезана са другим научним дисциплинама?

- Инжењери користе знања математике, логике, економије, искуство и интуицију да пронађу решење проблема
- Математика: решавање система нелинеарних једначина на континуалном или дискретном домену
- Pattern recognition / machine learning / data mining / knowledge discovery су уско повезани са оптимизацијом
- Оптимизација се врши алгоритмима који се данас по правилу реализују програмираним
- Дарвин: еволуција ствара и чува особине које су боље за опстанак у условима у којима се одиграва природна селекција
- Економија је наука која се бави проучавањем како друштво управља ограниченим ресурсима



# Шта је “проблем” или задатак?

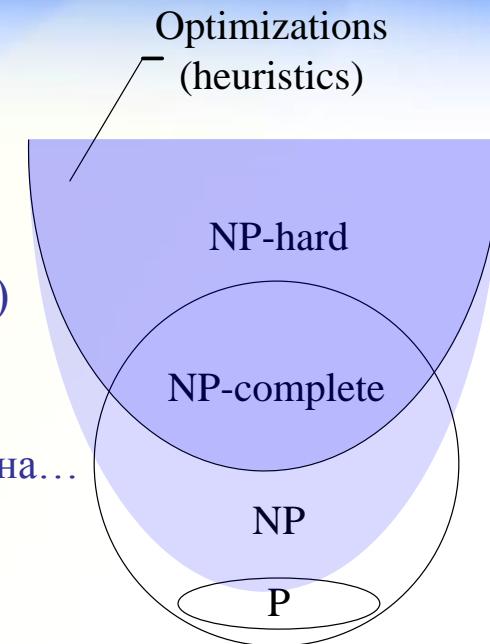
- Проблем (задатак) постоји онда када желимо нешто да решимо (постоји несклад/разлика између текућег и жељеног стања)
  - Задатак на испиту: имам поставку, а желим решење
  - Електротехника: направи електрични уређај или систем задатих карактеристика
  - Рачунарство: направи софтвер задатих карактеристика
- Реални проблеми се увек решавају ван јасно дефинисане области (на факултету, по правилу се решавају у оквиру области)
  - Велики и први корак ка решавању је знати одакле почети
- Решење је постизање жељеног циља у оквиру ограничења
- У пракси најчешће мора да се решава више проблема истовремено, а ти проблеми често могу имати опречне захтеве
- Постоје разни проблеми – ми ћемо се бавити рачунским

# Рачунски проблем (енг: computational problem)

- Проблем који се решава помоћу рачуна (и рачунара)
- Проблем = скуп свих вредности улазних података + одговарајући резултат за сваку вредност улазних података
  - Скуп улазних података може бити коначан или бесконачан
- Подела:
  - Проблеми одлучивања: имају бинаран резултат {да, не} или {1, 0} или {true, false}
  - Проблеми оптимизације: имају генералан нумерички резултат {реалан број, низ бита, скуп бројева, пермутација елемената скупа итд. }
- Проблеми оптимизације могу се трансформисати у проблеме одлучивања формулацијом питања
  - Оптимизација: пронаћи најкраћу путању између задатих тачака
  - Одлука: да ли постоји путања краћа од  $x$  [m] између задатих тачака?
  - Последица: **теорија рачунске сложености** (енг: computational complexity theory) **важи и за оптимизационе проблеме!**

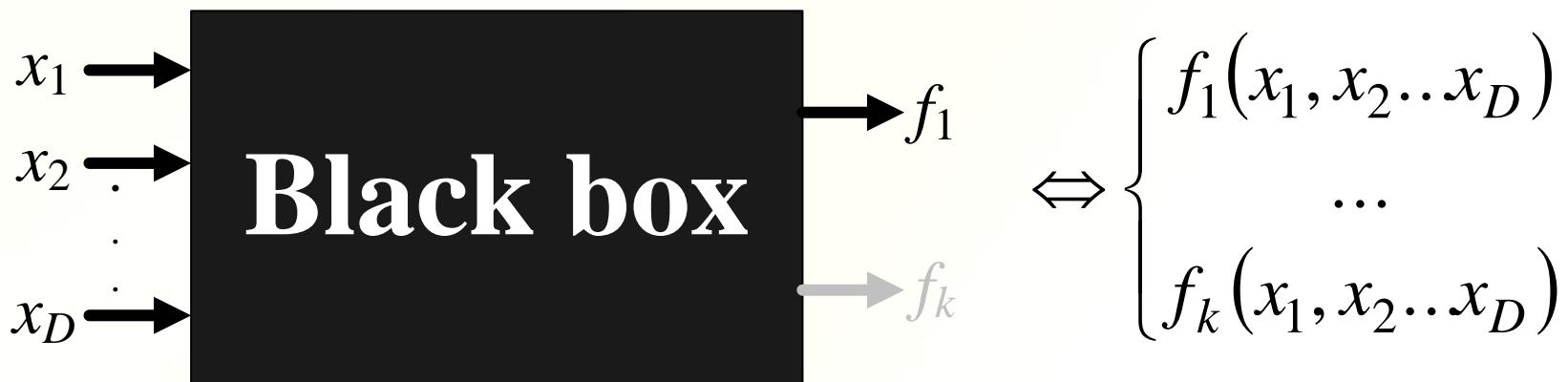
# Сложеност алгоритама и поређење перформанси

- Тјурингова машина
- Детерминистички и недетерминистички приступи
- Временска сложеност и просторна сложеност:  
рачунарски ресурси потребни за решавање проблема  
= време (рачунске операције) + простор (меморијске локације)
- Претпоставимо да алгоритам обрађује  $N$  улазних елемената
- Сложеност алгоритма
  - линеарна, квадратна, логаритамска, полиномска, експоненцијална...
- Означавање
  - $O(N)$ ,  $O(N^2)$ ,  $O(\log N)$ ,  $O(N^{\text{const.}})$ ,  $O(2^N)$  итд.
- Теорија: алгоритми се пореде према сложености
- Питање од милион долара:  $P \neq NP$  или  $P = NP$ ?
- Пракса:  $t = k_t \cdot O(f_t(N))$  и  $m = k_m \cdot O(f_m(N))$   
да ли проблем може да се реши на задатом хардверу у задатом времену?
- За стохастичке оптимизационе алгоритме тражићемо и  
најбрже проналажње (средњег) најбољег решења



# Black-box оптимизације (Модел црне кутије)

- У пракси решавање се своди на проблеме за које не постоји предзнање
  - све што зnamо о проблему потребно је искористити за решавање
- За задате улазне податке (побуду)  $\mathbf{x} = (x_1, x_2, \dots, x_D)$  добијају се резултати (одзиви)  $\mathbf{f} = (f_1, f_2, \dots, f_k)$
- Одзиви нису познати унапред за све могуће побуде (ако су сви одзиви познати, најбољи одзив је решење проблема)
- Унутрашња организација непозната (због природе, ограничења...)



# Резултати црне кутије:

# “The Blind Men and the Elephant”

## - John Godfrey Saxe (1816-1887) -

It was six men of Indostan  
To learning much inclined,  
Who went to see the Elephant  
**(Though all of them were blind),**  
**That each by observation**  
Might satisfy his mind.

The *First* approached the Elephant,  
And happening to fall  
Against his broad and sturdy side,  
At once began to bawl:  
"God bless me! but the **Elephant**  
**Is very like a WALL!**"

The *Second*, feeling of the tusk,  
Cried, "Ho, what have we here,  
So very round and smooth and sharp?  
To me 'tis mighty clear  
This wonder of an **Elephant**  
**Is very like a SPEAR!**"

The *Third* approached the animal,  
And happening to take  
The squirming trunk within his hands,  
Thus boldly up and spake:  
"I see," quoth he, "**the Elephant**  
**Is very like a SNAKE!**"

The *Fourth* reached out an eager hand,  
And felt about the knee  
"What most this wondrous beast is like  
Is mighty plain," quoth he:  
""Tis clear enough the **Elephant**  
**Is very like a TREE!**"

The *Fifth*, who chanced to touch the ear,  
Said: "E'en the blindest man  
Can tell what this resembles most;  
Deny the fact who can,  
This marvel of an **Elephant**  
**Is very like a FAN!**"

The *Sixth* no sooner had begun  
About the beast to grope,  
Than seizing on the swinging tail  
That fell within his scope,  
"I see," quoth he, "**the Elephant**  
**Is very like a ROPE!**"

And so these men of Indostan  
Disputed loud and long,  
Each in his own opinion  
Exceeding stiff and strong,  
**Though each was partly in the right,**  
**And all were in the wrong!**

# Формализација

- Формално, сваки одзив је функција више променљивих (побуда)
- Одзив, или нека функција одзива, је мера колико је решење добро или лоше
- При оптимизацији (решавању проблема), за сваки скуп улазних података које желимо да разматрамо, потребно је израчунавати одзив(е)
- Подела модела (поставки) према броју одзива:
  - са једним одзивом (један критеријум оптимизације)
  - са више одзива (више критеријума оптимизације)
- Први део курса: оптимизације са једним критеријумом
- Последња трећина курса: вишекритеријумске оптимизације

# Оптимизација и функције више променљивих

- Побуде  $\leftrightarrow$  оптимизационе променљиве
- Одзив  $\leftrightarrow$  оптимизациона функција  
(нумеричка мера квалитета решења)
  - Оптимизациона функција је функција са једном или више променљивих  $f(x_1, x_2..x_D)$
  - Променљиве и резултат оптимизационе функције могу бити из скупа дискретних или континуалних бројева

# Терминологија

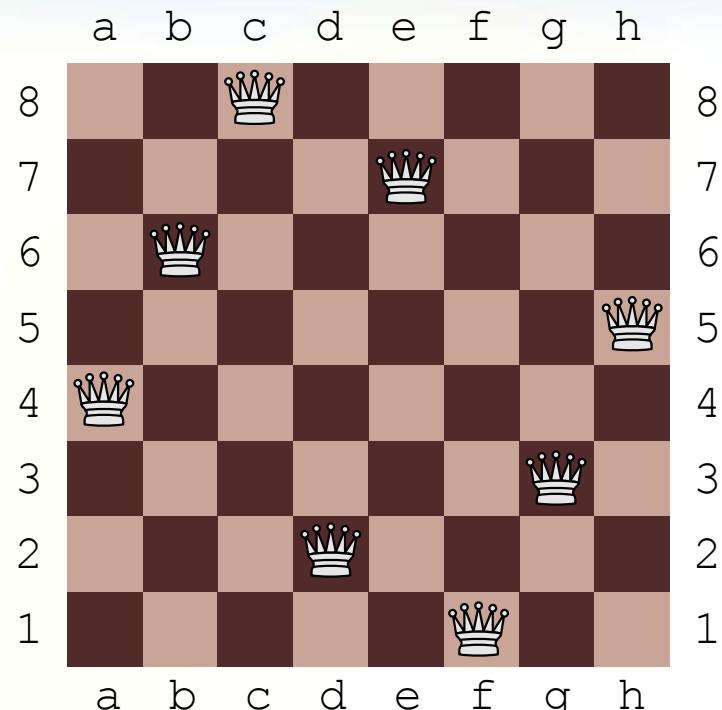
- **Оптимизациона функција** ≡  
функција грешке ≡ cost function ≡ evaluation function ≡  
нумеричка мера квалитета разматраног решења
- Једно израчунавање оптимизационе функције ≡ **итерација**
- **Оптимизационе променљиве** ≡ улазни подаци за проблем
- **Оптимизациони простор ( $S$ )** је  
скуп свих могућих вредности оптимизационих променљивих  
(енг: optimization space, search space, set of candidate solutions...)
- **Простор (смислених) решења ( $F$ )** је  
скуп свих решења које има смисла разматрати  
(енг: feasible solutions,...) 
$$F \subseteq S$$
- **Број димензија оптимизационог простора ( $D$ )**  
је број оптимизационих променљивих
- Свака тачка у оптимизационом простору  
представља један избор вредности улазних података

# Колико је велики оптимизациони простор?

- Величина оптимизационог простора је од изузетне важности
- Избор оптимизационог алгоритма зависи од:
  - величине оптимизационог простора
  - домена оптимизационих променљивих
- Величина простора зависи од усвојеног записа!
  - Бијективни записи имају исту величину оптимизационог простора
  - Најбољи запис је онај за који важи  $F = S$
- Оптимизациони простор може бити ограничен или неограничен  
(constrained vs. unconstrained)

# Пример записа и величине опт. простора: $N$ -краљица

- Проблем: поставити  $N$ -краљица на “шаховску таблу” димензија  $N \times N$  тако да се краљице не нападају
- Нека су  $r_k$  и  $c_k$  позиције краљице  $k$ :  
 $1 \leq k \leq N$  (редни број)  
 $1 \leq r_k \leq 8$  (ред)  
 $1 \leq c_k \leq 8$  (колона)
- Краљица:  
 $f1 \rightarrow (r_k, c_k) = (1, 6)$
- Колико је велики оптимизациони простор?



# Величина оптимизационог простора зависи од записа!

- Запис #1: 64 могућа места  $(1,1), (1,2), \dots (8,8)$  на која можемо да ставимо 8 краљица  
 $\mathbf{x} = \{K_1, K_2, \dots, K_8\}, 1 \leq K_n \leq 64$
- Запис #2: свака краљица у свом реду  
 $\mathbf{x} = \{K_1, K_2, \dots, K_8\}, 1 \leq K_n \leq 8$   
K#1:  $(1, K_1), \dots, K#8: (8, K_8)$
- Запис #3: краљице не могу бити у истом реду и у истој колони  
 $\mathbf{x} = \{3, 6, 2, 4, 1, 5, 7, 8\}$   
K#1:  $(1, 3), K#2: (2, 6), \dots, K#8: (8, 8)$

$$|F| = \binom{64}{8} = 4\,426\,165\,368$$

$$|F| = 8^8 = 16\,777\,216$$

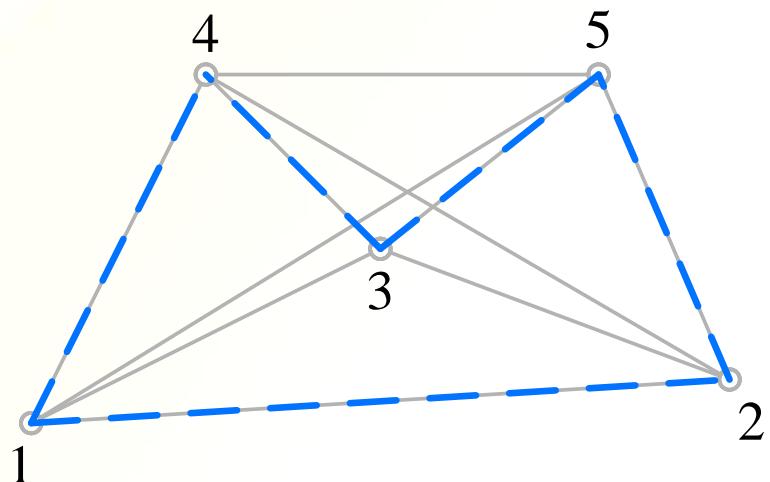
$$|F| = 8! = 40\,320$$

# Основни типови оптимизационих проблема (TSP, SAT, NLP)

- Проблем трговачког путника  
(енглески: traveling salesman problem, TSP)
- Булова алгебра (дискретна стања)  
(енглески: Boolean satisfiability, SAT)
- Нелинеарни проблеми  
(енглески: Nonlinear programming, NLP)
- Домени основних типова проблема  
(дискретан или континуалан)

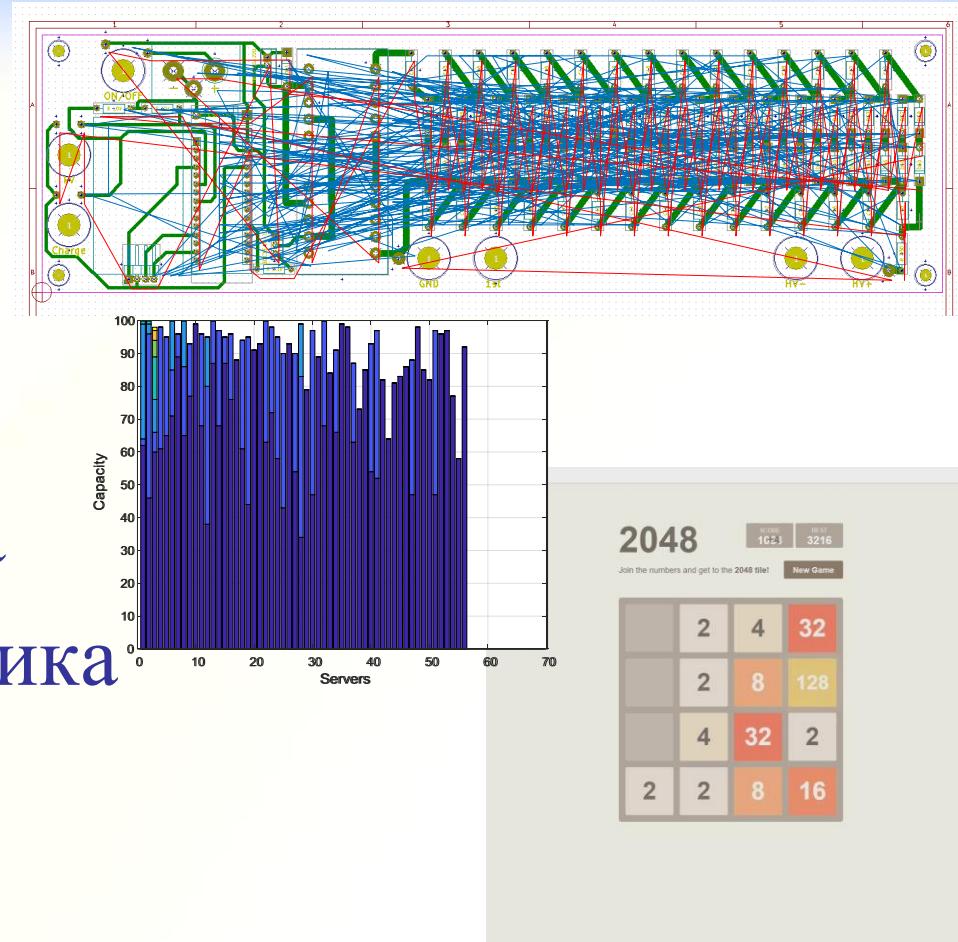
# TSP: поставка

- За задати скуп  $D$  градова и позната растојања између сваког паре градова, пронаћи најкраћи пут који пролази кроз сваки град и завршава се у полазном граду
- Број могућих путања  $D$ !
  - $D = 15$ , број путања  $\approx 1,3 \cdot 10^{12}$
  - $D = 150$ , број путања  $\approx 57,1 \cdot 10^{261}$
- Приказана путања  
5-3-4-1-2(-5)
- Подела:
  - Симетричан:  $dist(p,q) = dist(q,p)$ 
    - Асиметричан:  $dist(p,q) \neq dist(q,p)$
  - Метрички ( $d_{AB} \leq d_{AC} + d_{CB}$ ) и они који то нису
  - Еуклидски: симетричан + метрички



# TSP: инжењерски проблеми

- Минимална путања алата за бушење на штампаним плочама
- Рутирање возила
- Путање у графовима
- Планирање и логистика
- ...



# TSP класа проблема и поделе

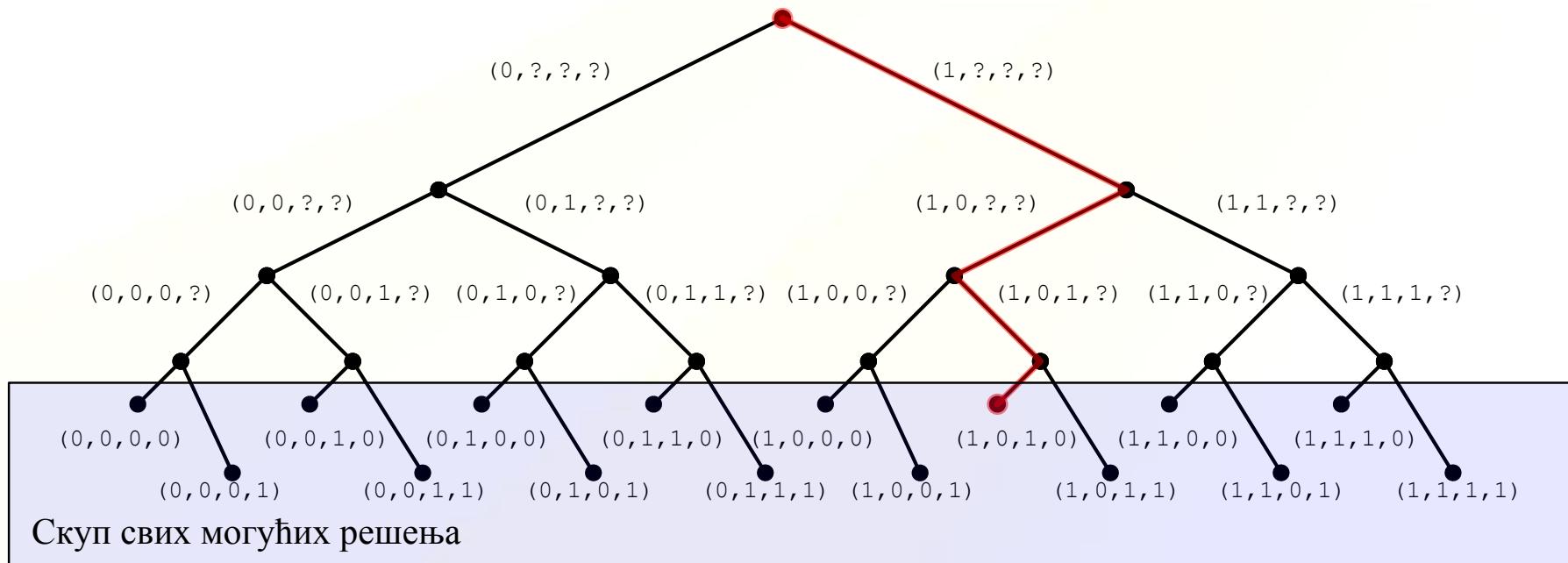
- Сваки проблем који може да се сведе на проверу **свих пермутација** називаћемо **проблем TSP класе**
- Енглески: TSP-encoded problems
- Поделе TSP:
  - Симетричан (енг: symmetric TSP, STSP)
  - Асиметричан (енг: asymmetric TSP, ATSP)
  - Најкраћи Хамилтонов пут (енг: shortest Hamiltonian path, SHP)
    - Хамилтонова контура (енг: Hamiltonian cycle)
  - Обилазак са условима (енг: Sequential ordering problem, SOP)
  - Рутирање возила ограниченог капацитета (енг: Capacitated vehicle routing problem, CVRP)
    - ...
- Посебне поткласе могу имати једноставнија решења
- Генерализација: job shop problem

# SAT: поставка

- За задати логички израз  $F(x_1, x_2, \dots, x_D)$  пронаћи вредности променљивих  $(x_1, x_2, \dots, x_D)$ ,  $x_k \in \{\text{true (1), false (0)}\}$ ,  $k = 1, 2, \dots, D$  тако да је  $F(x_1, x_2, \dots, x_D) = \text{true}$   
$$F(\mathbf{x}) = x_1 \wedge ((x_2 \wedge x_1) \vee (x_2 \wedge x_3 \wedge \neg x_4))$$
- Пример  $\mathbf{x}_{\text{opt}} = (x_1, x_2, x_3, x_4) = (1, 0, 1, 0)$   
$$F(\mathbf{x}_{\text{opt}}) = \text{true}$$
- Број различитих  $\mathbf{x}$  је  $2^D$ 
  - $D = 15$ , број могућих решења 32768
  - $D = 150$ , број могућих решења  $\approx 1,4 \cdot 10^{45}$
- Улазни подаци могу да се запишу као низ бита 100110...

# SAT: приказ помоћу графова (бинарно стабло графа)

$$F(\mathbf{x}) = x_1 \wedge ((x_2 \wedge \bar{x}_1) \vee (\bar{x}_2 \wedge x_3 \wedge \bar{x}_4))$$



# SAT класа проблема и поделе

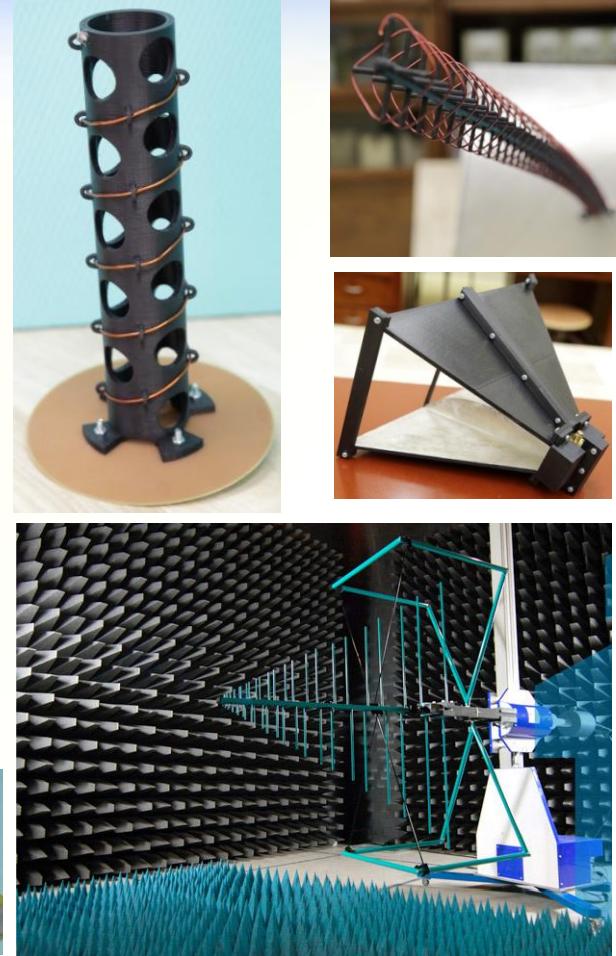
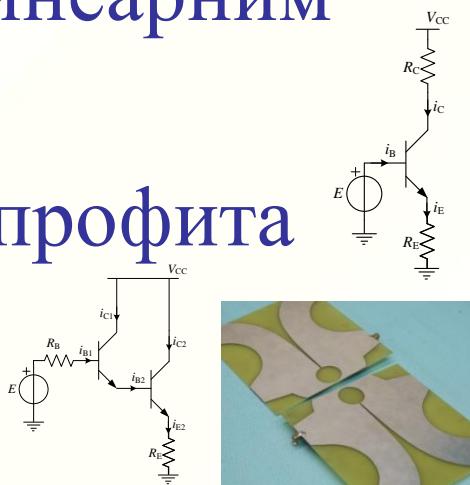
- Сваки проблем који се своди на проверу свих могућих вредности секвенце бита називаћемо проблем SAT класе
- Поделе (Karp's 21 NP-complete problems)
  - проблем ранца (knapsack)
  - подела посла (job-sequencing)
  - directed/undirected Hamiltonian cycle
  - ...
- Примери: свака манипулација битима може да претвори у проблем SAT класе
  - Откључавање ZIP архиве
  - Пробијање RSA кодова
  - ...

# NLP проблеми

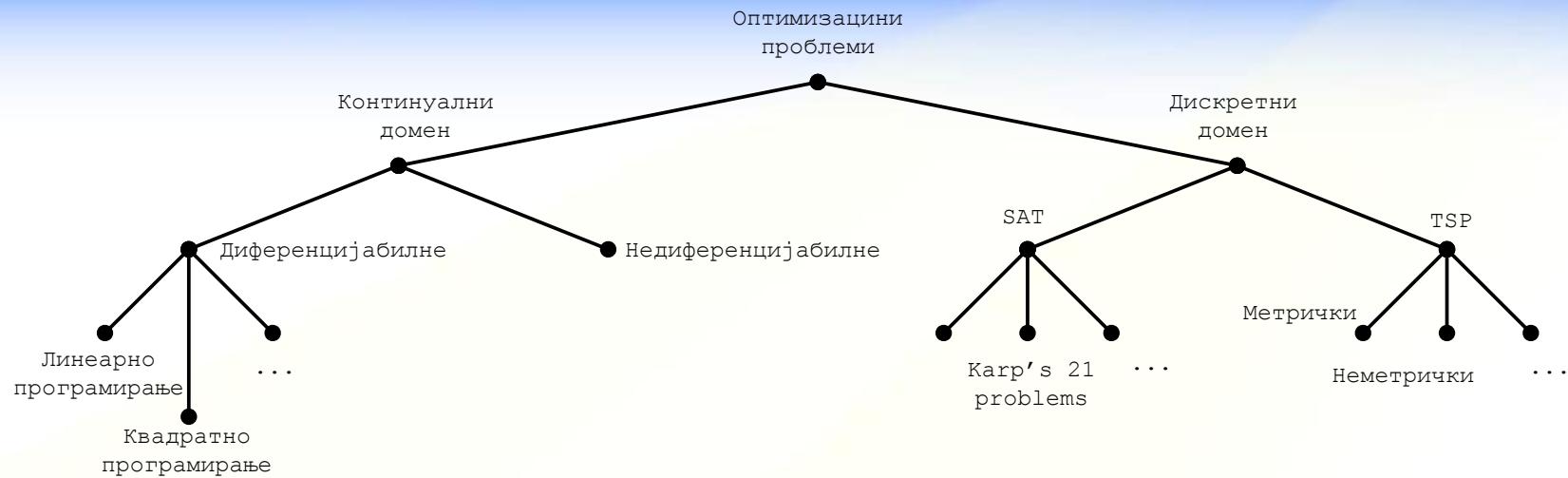
- Сваки оптимизациони проблем са **континуалним променљивима** спада у NLP класу проблема
- Број могућих решења је (теоријски) **бесконачан!**
- Посебни случајеви
  - диференцијабилне и недиференцијабилне  $f$
  - линеарно програмирање
  - квадратно програмирање
  - конвексни/конкавни проблеми
  - са ограниченим или неограниченом доменом
  - ...

# NLP инжењерски примери

- Пројектовање ЕМ уређаја
- Решавање нелинеарних кола
- Фитовање резултата мерења
- Управљање нелинеарним системима
- Максимизација профита
- ...



# Класификација оптимизационих проблема



- Подела није јединствена
- Пермутације су специјалан случај низова бита:  $TSP \subset SAT$  ?
- Све што радимо на рачунару је са коначном тачношћу (64-бита):  $NLP \subset SAT$  ?
- Класификација је према **запису улазних података (побуда)** и природи оптимизационог проблема

# Колико је брз мој рачунар?

- Колико времена могу да потрошим за оптимизацију?
- Колико пута могу да израчунам опт. функцију  $f(\mathbf{x})$ ?
- Желим процену брзине: максималан број позива (оптимизационе) функције
- Једноставан програм: прототип оптимизације

```
#include <stdio.h>
#include <time.h>

double F(double a, double b)
{
    return a + b;
}

int main(void)
{
    double x,eval;
    double max = 7e9;

    time_t t1, t2;
    time(&t1);

    for (x=0; x < max; x=x+1.0 )
        eval = F(x,x);

    time(&t2);

    printf("%5.5e\n", (t2 - t1)/max);
    return 0;
}
```

# Колико времена је потребно да решимо проблеме?

- Претпоставимо да је свака провера 1ns
- TSP 100 градова:  
 $100! \cdot 1\text{ns} \approx 3 \cdot 10^{141}$  година!
- SAT 100 бита:  
 $2^{100} \cdot 1\text{ns} \approx 4 \cdot 10^{13}$  година!
- NLP: **x** има  $\infty$  могућности  
→ потребно је теоријски бесконачно много времена!



# Решавање реалних оптимизационих проблема

- Број могућих решења је изузетно велики те је потребно изузетно много рачунарских ресурса
- Постоји много **ограничења** тако да је тешко наћи било какво решење (а не оптимално)
  - Посебно ограничење: човек који решава проблем није адекватно припремљен
- Често је доволно наћи **задовољавајуће решење**
  - Трагање за најбољим (оптималним) решењем је можда занимљиво али најчешће неприхватљиво
- Тада се примењују  
**оптимизациони алгоритми и хеуристике**

# Шта су хеуристике?

- ХЕУРИСТИКА (енг: heuristic)  
грчки корен речи “Εύρισκω“ – пронаћи или открити
- IEEE: All engineering is heuristic
- Технике решавања проблема засноване на искуству, учењу и откривању које доводе до решења (не мора нужно бити оптимално али је довољно добро)
- Када год је немогуће или непрактично потпуно претраживање простора, користе се хеуристике (интуиција, стереотипи, здрав разум, енг: rule-of-thumb, educated guess, ...)
- **State-of-the-art** проблеми се увек решавају хеуристички

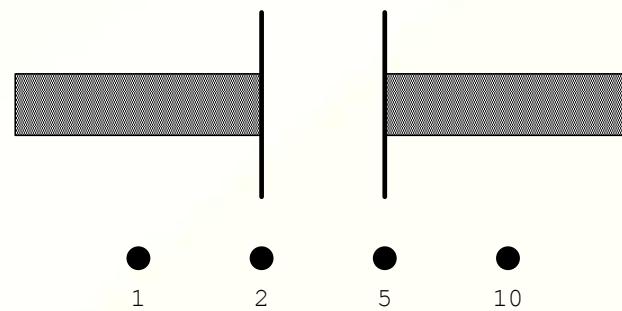
# Шта је заједничко за све приступе решавању “проблема” (задатака)?

- Структура сваког алгоритма (приступа) за решавање “проблема” има три основна дела
  1. запис могућих решења,
  2. циљ који је потребно постићи и
  3. оптимизациону функцију (нумеричку меру појединачних решења)
- Запис (енг: representation)
  - TSP: једна пермутација, нпр.  $\mathbf{x} = (2,1,3,6,4,5,7)$
  - SAT: један низ бита, нпр.  $\mathbf{x} = 1001100101$
  - NLP: један вектор реалних бројева, нпр.  $\mathbf{x} = (1.5, -3.2, 4.76, 17.2)$
- Запис не мора да буде “природан”: TSP као бинарни низ, реални бројеви као низ бита, итд.
  - Генерално на запис се може применити било која трансформација (пресликавање) и добити други запис.
  - Бијективни записи проблема имају исту сложеност решавања!
- Циљ и оптимизациона функција НИСУ исто!
- Запис и опт. функцију формулише онај ко решава проблем

# Пример:

## 4 човека и трошни мост

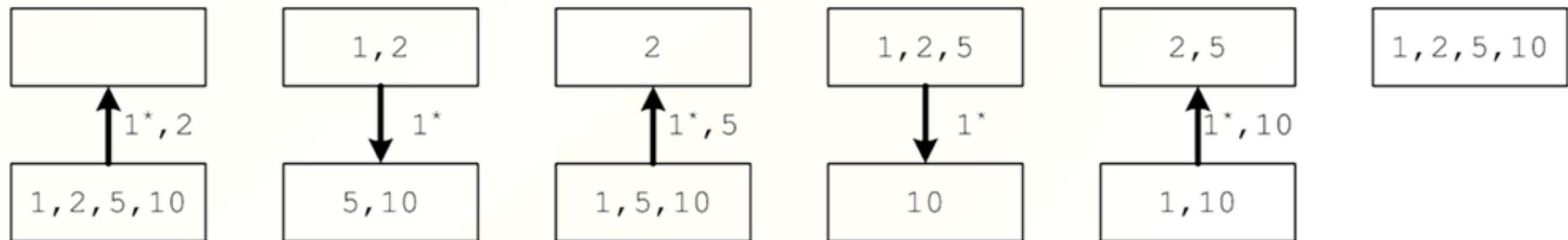
Четири особе морају да пређу са једне стране трошног моста на другу, током ноћи. Мост највише може да издржи две особе истовремено. За прелазак је неопходна лампа да би се осветлио сваки корак и избегле рупе (тј. при преласку у паровима иду брзином споријег). Постоји само једна лампа, коју није могуће пребацити са једне стране моста на другу (тј. увек неко мора да је носи). Први човек може да пређе мост за 1 min, други за 2 min, трећи за 5 min и четврти за 10 min. Пронађи минимално време потребно да сви пређу на другу страну моста.



# Пример:

## Основне идеје и једно решење

- Домен проблема је дискретан, а број могућности је пребројив и коначан.
- Прелази се у паровима, а лампу враћа једна особа.
- Да би четири особе прешли, потребно је 3 преласка и 2 повратка.
- Лампу са друге стране враћа увек најбржи који је на тој страни.
- Интуитивно решење, најбржи носи (враћа) лампу и преводи једног по једног  $\{1,2,1,1,5,1,1,10\}$ ,  $T = 19 \text{ min}$ .



# Пример:

## Пребројавање решења и запис

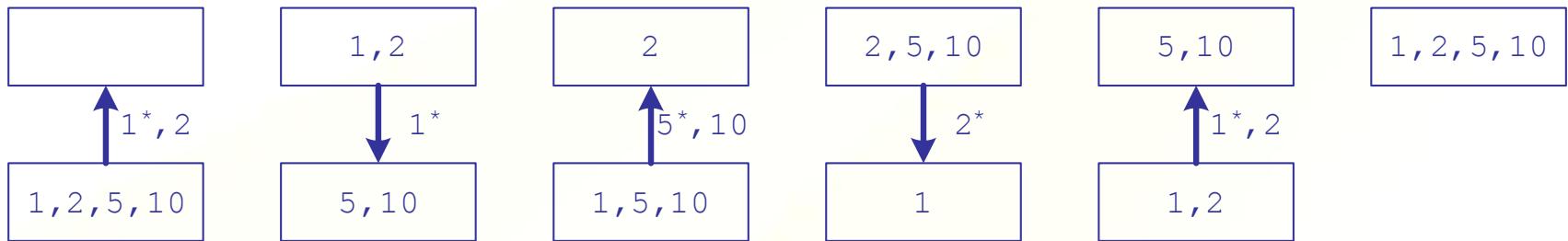
- При преласку ( $\uparrow$ ) за избор првог паре постоји  $\binom{4}{2}$  могућности, за избор другог паре  $\binom{3}{2}$  могућности и за избор трећег паре  $\binom{2}{2} = 1$  могућност.
- При повратку ( $\downarrow$ ) имамо само једну могућност.
- Укупан број могућих комбинација прелазака је  $N = \binom{4}{2} \cdot 1 \cdot \binom{3}{2} \cdot 1 \cdot \binom{2}{2} = 18$ .
- Све комбинације можемо да запишемо у облику вектора  $\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$  где су елементи вектора брзине одговарајућих људи, тј.  $x_k \in \{1, 2, 5, 10\}$ ,  $k = 1, 2, \dots, 8$ . Први пар при преласку једнозначно одређују  $x_1, x_2$ , други пар  $x_4, x_5$ , трећи пар  $x_7, x_8$ , а повратак једнозначно одређују  $x_3$  и  $x_6$ .
- Током прелазака потребно је водити евиденцију о томе ко је на којој страни (брзине једнозначно одређују људе). На почетку, на првој страни су сви  $S_1 = \{1, 2, 5, 10\}$ , а на другој нема људи  $S_2 = \{\}$ . После преласка првог паре и повратка једне особе са лампом  $|S_1| = 3$  и  $|S_2| = 1$ .
- Укупно време преласка је  $T = \max(x_1, x_2) + x_3 + \max(x_4, x_5) + x_6 + \max(x_7, x_8)$ .

# Пример: све могућности

$\uparrow$ #1	$\downarrow$ #1	$\uparrow$ #2	$\downarrow$ #2	$\uparrow$ #3	$T$			
$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	
1	2	1	1	5	1	1	10	19
1	2	1	1	10	1	1	5	19
<b>1</b>	<b>2</b>	<b>1</b>	<b>5</b>	<b>10</b>	<b>2</b>	<b>1</b>	<b>2</b>	<b>17</b>
1	5	1	1	2	1	1	10	19
1	5	1	1	10	1	1	2	19
1	5	1	2	10	2	1	2	20
1	10	1	1	2	1	1	5	19
1	10	1	1	5	1	1	2	19
1	10	1	2	5	2	1	2	20
2	5	2	1	2	1	1	10	20
2	5	2	1	10	1	1	2	19
2	5	2	2	10	2	1	2	21
2	10	2	1	2	1	1	5	20
2	10	2	1	5	1	1	2	20
2	10	2	2	5	2	1	2	21
5	10	5	1	2	1	1	5	23
5	10	5	1	5	1	1	2	23
5	10	5	2	5	2	1	2	24

# Пример: решење

- Минимално време преласка је 17 min



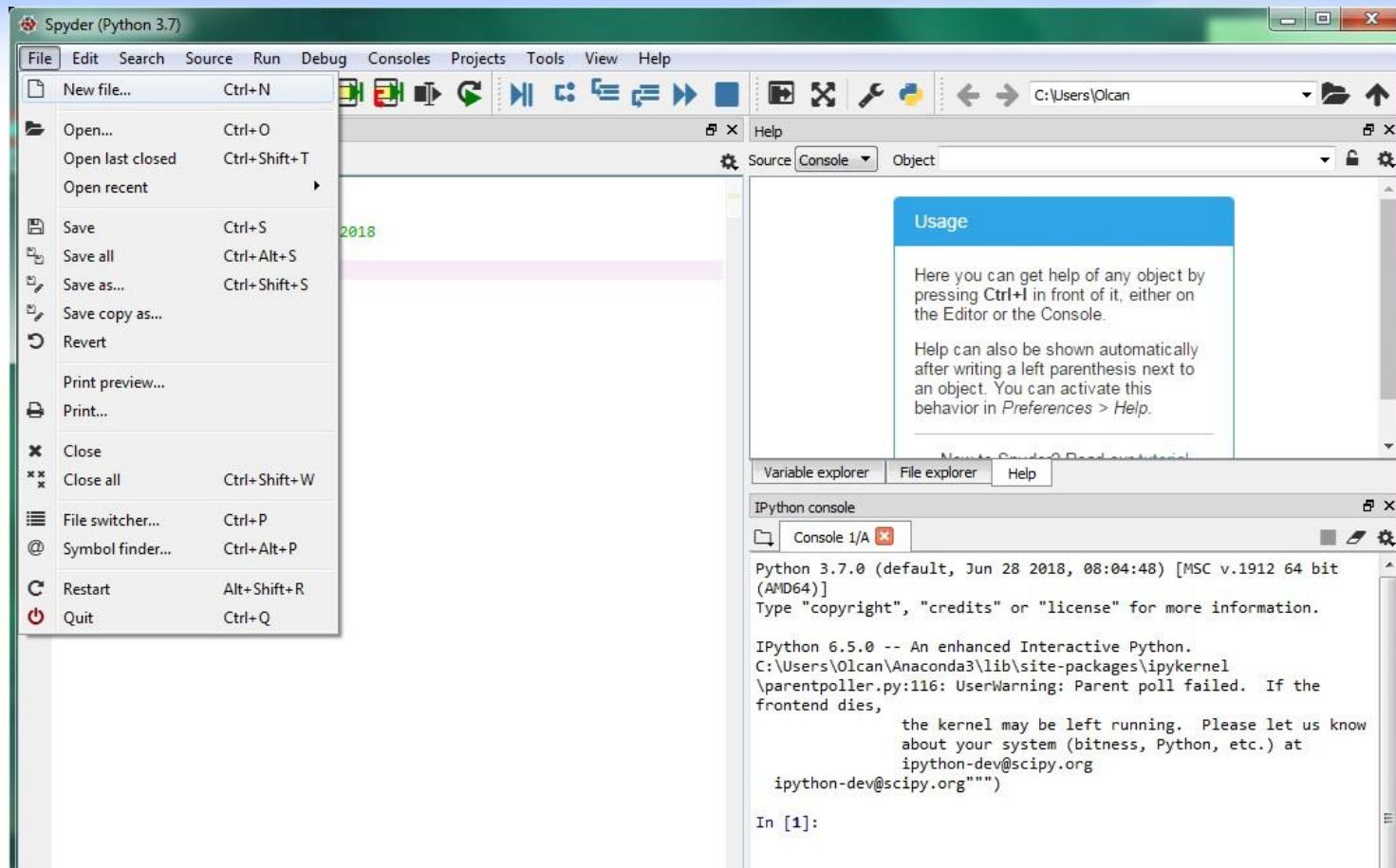
# Задатак за вежбе

- Купац је купио четири производа. Ако се цене изразе у доларима, онда су збир и производ свих цена (бројчано) исти и износе 7,11. Цене се заокружују на \$0,01 (1 cent).  
(а) Написати програм који извршава потпуну претрагу по све четири цене и помоћу њега одредити цене. Израчунати максималан број позива оптимизационе функције.  
(б) Изразити једну цену преко осталих и написати програм који извршава потпуну претрагу по (преостале) три цене. Израчунати максималан број позива опт. функције и упоредити брзину програма у односу на програм из (а).  
(в) Који од ова два програма је бржи?

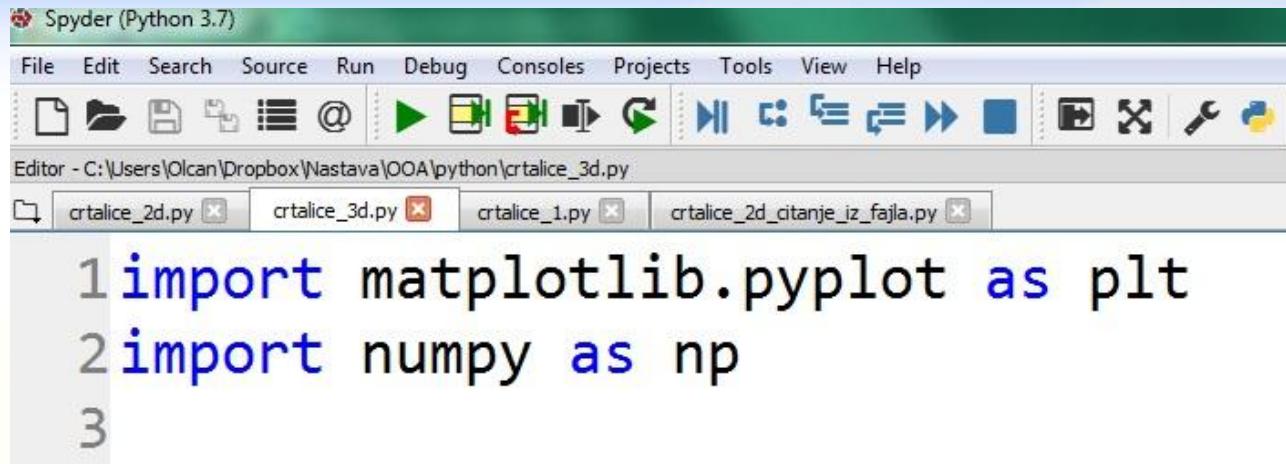
# Вежбе на рачунару: Мотивација

- Упознавање са окружењима која ће бити коришћена на вежбама
  - **Python 3.7** (Spyder Anaconda 3):  
приказ и обрада резултата
  - **C/C++** (Visual Studio 2017):  
рачунарски захтевни делови програма
- Студенти се охрабрују да користе и своје рачунаре.

# Python 3.7 (Spyder окружење са Anaconda 3)



# Библиотеке



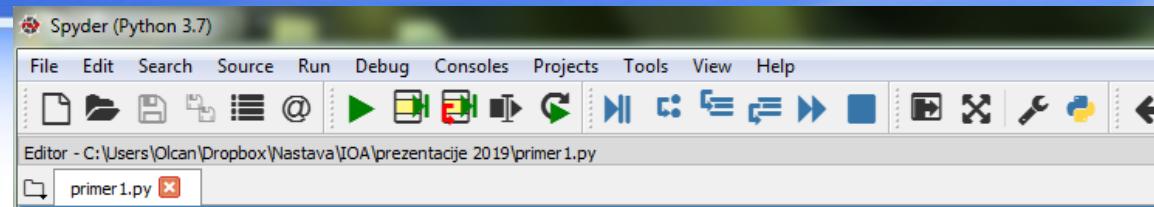
The screenshot shows the Spyder Python IDE interface. The title bar reads "Spyder (Python 3.7)". The menu bar includes File, Edit, Search, Source, Run, Debug, Consoles, Projects, Tools, View, and Help. Below the menu is a toolbar with various icons. The central area is an "Editor" window titled "C:\Users\Olcan\Dropbox\Nastava\OOA\python\crtalice\_3d.py". The code in the editor is:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
```

The status bar at the bottom shows the file path "C:\Users\Olcan\Dropbox\Nastava\OOA\python\crtalice\_3d.py".

- **plt**: figure(), plot(), plot\_surface(),  
set\_xlabel(), plt.legend()
- **np**: arange(), meshgrid()
- Користан сайт: <https://matplotlib.org/>

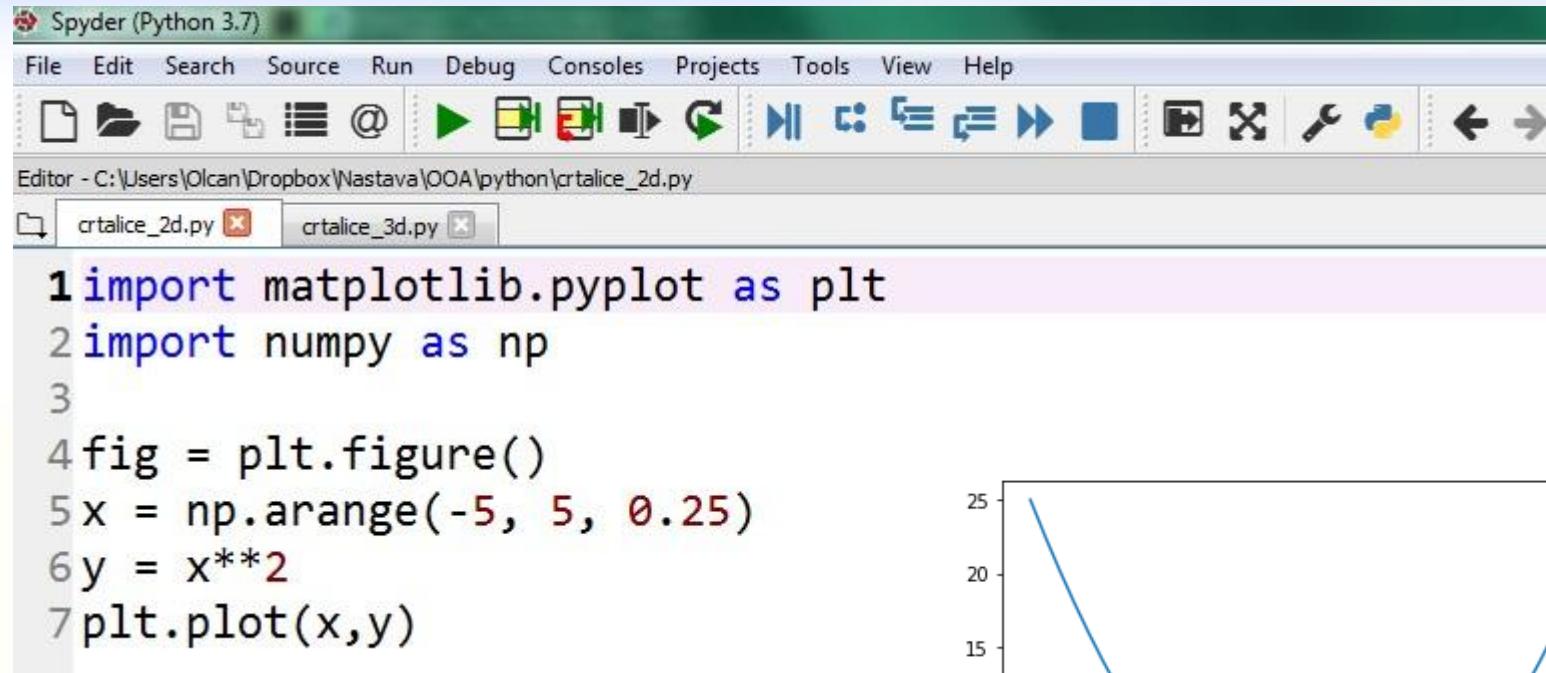
# Дефинисање функција



```
Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - C:\Users\Olcان\Dropbox\Nastava\IOA\prezentacije 2019\primer1.py
primer1.py

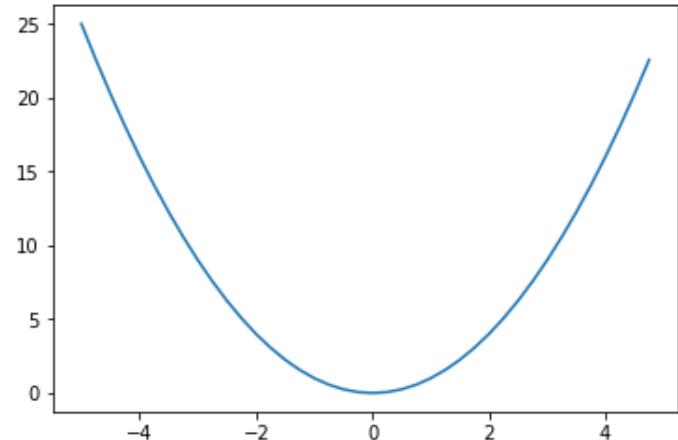
1 import numpy as np
2
3 def napon_otpornika(R, I):
4     return R*I
5
6 def prosto_kolo(R1, R2, E):
7     I = E/(R1+R2)
8     U1 = I*R1
9     U2 = I*R2
10    return np.array([I, U1, U2])
11
12 R = 2e3
13 I = 1e-3
14 U = napon_otpornika(R, I)
15 print(U)
16
17 R1 = 1e3
18 R2 = 3e3
19 E = 1
20 resenje = prosto_kolo(R1, R2, E)
21 print(resenje[1], resenje[2])
```

# 1D функција: *plot*



The screenshot shows the Spyder Python IDE interface. The title bar says "Spyder (Python 3.7)". The menu bar includes File, Edit, Search, Source, Run, Debug, Consoles, Projects, Tools, View, and Help. The toolbar has various icons for file operations and debugging. The status bar indicates "Editor - C:\Users\Olcan\Dropbox\Nastava\OOA\python\crtalice\_2d.py". Below the toolbar, there are two tabs: "crtalice\_2d.py" and "crtalice\_3d.py", with "crtalice\_2d.py" being active. The code editor contains the following Python script:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig = plt.figure()
5 x = np.arange(-5, 5, 0.25)
6 y = x**2
7 plt.plot(x,y)
```



# Легенда

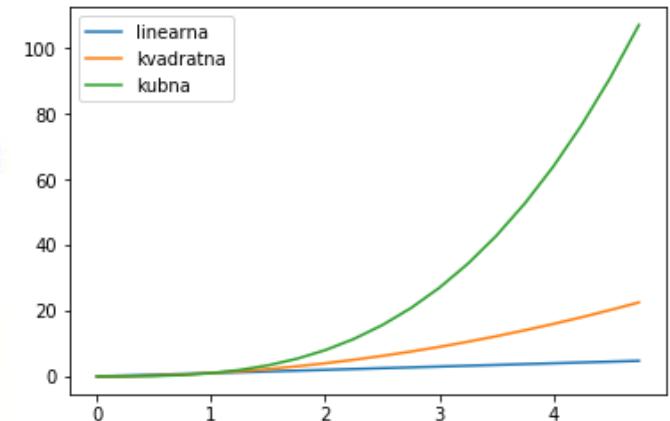
Spyder (Python 3.7)

File Edit Search Source Run Debug Consoles Projects Tools View Help

Editor - C:\Users\Olcan\Dropbox\Nastava\OOA\python\crtalice\_2d.py

crtalice\_2d.py\* crtalice\_3d.py crtalice\_1.py crtalice\_2d\_citanje\_iz\_fajla.py

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig = plt.figure()
5 x = np.arange(0, 5, 0.25)
6
7 plt.plot(x, x, label = 'linearna')
8 plt.plot(x, x**2, label = 'kvadratna')
9 plt.plot(x, x**3, label = 'kubna')
10 plt.legend()
```



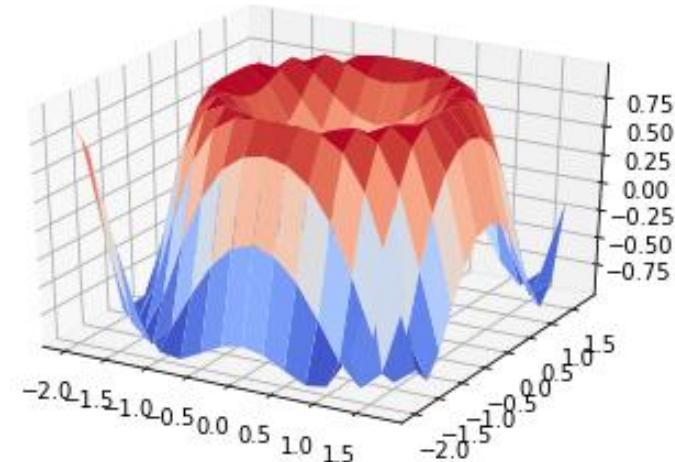
# 2D функција: *plot\_surface* и *meshgrid*

Spyder (Python 3.7)

```
File Edit Search Source Run Debug Consoles Projects Tools View Help  
Editor - C:\Users\Olcان\Dropbox\Nastava\OOA\python\crtalice_3d.py  
crtalice_s_par.py crtalice_3d.py crtalice_2d.py
```

1 from mpl\_toolkits.mplot3d import Axes3D  
2  
3 import matplotlib.pyplot as plt  
4 import numpy as np  
5  
6 fig = plt.figure()  
7 ax = fig.gca(projection = '3d')  
8  
9 x = np.arange(-2, 2, 0.25)  
10 y = np.arange(-2, 2, 0.25)  
11 x,y = np.meshgrid(x,y)  
12 z = np.sin(x\*\*2+y\*\*2)  
13 h = ax.plot\_surface(x,y,z, cmap=plt.cm.coolwarm)

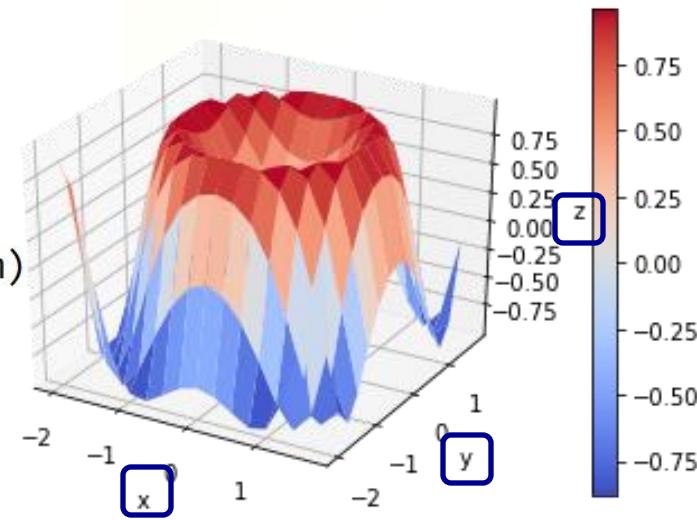
14



# Означавање оса

```
Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - C:\Users\Olcان\Dropbox\Nastava\OOA\python\crtalice_3d.py
crtanje_s_par.py crtalice_3d.py crtalice_2d.py

1 from mpl_toolkits.mplot3d import Axes3D
2
3 import matplotlib.pyplot as plt
4 import numpy as np
5
6 fig = plt.figure()
7 ax = fig.gca(projection = '3d')
8
9 x = np.arange(-2, 2, 0.25)
10 y = np.arange(-2, 2, 0.25)
11 x,y = np.meshgrid(x,y)
12 z = np.sin(x**2+y**2)
13 h = ax.plot_surface(x,y,z, cmap=plt.cm.coolwarm)
14
15 ax.set_xlabel('x')
16 ax.set_ylabel('y')
17 ax.set_zlabel('z')
18
19 fig.colorbar(h)
```



# Учитавање података из датотеке

Spyder (Python 3.7)

File Edit Search Source Run Debug Consoles Projects Tools View Help

Editor - C:\Users\Olcان\Dropbox\Nastava\OOA\python\crtalice\_2d\_citanje\_iz\_fajla.py

crtanje\_s\_par.py crtalice\_3d.py crtalice\_2d.py crtalice\_2d\_citanje\_iz\_fajla.py

```
1 import matplotlib.pyplot as plt
2
3 file = open('funkcija.txt','r')
4 x = []
5 y = []
6 for line in file:
7     s = line.split()
8     if (len(s)!=0):
9         x.append(float(s[0]))
10        y.append(float(s[1]))
11 file.close()
12
13 plt.figure()
14 plt.plot(x,y)
15 plt.xlabel('x')
16 plt.ylabel('y')
17
```

funkcija.txt - No...

File Edit Format View Help

x	y
-5.0	25.0
-4.75	22.5625
-4.5	20.25
-4.25	18.0625
-4.0	16.0
-3.75	14.0625
-3.5	12.25
-3.25	10.5625
-3.0	9.0
-2.75	7.5625
-2.5	6.25
-2.25	5.0625
-2.0	4.0
-1.75	3.0625
-1.5	2.25
-1.25	1.5625
-1.0	1.0
-0.75	0.5625
-0.5	0.25
-0.25	0.0625
0.0	0.0
0.25	0.0625
0.5	0.25
0.75	0.5625
1.0	1.0
1.25	1.5625
1.5	2.25
1.75	3.0625
2.0	4.0
2.25	5.0625
2.5	6.25
2.75	7.5625

# Пример минимизације оптимизационе функције

Spyder (Python 3.7)

File Edit Search Source Run Debug Consoles Projects Tools View Help

Editor - C:\Users\Olcان\Dropbox\Nastava\IOA\prezentacije 2019\primer2.py

primer2.py

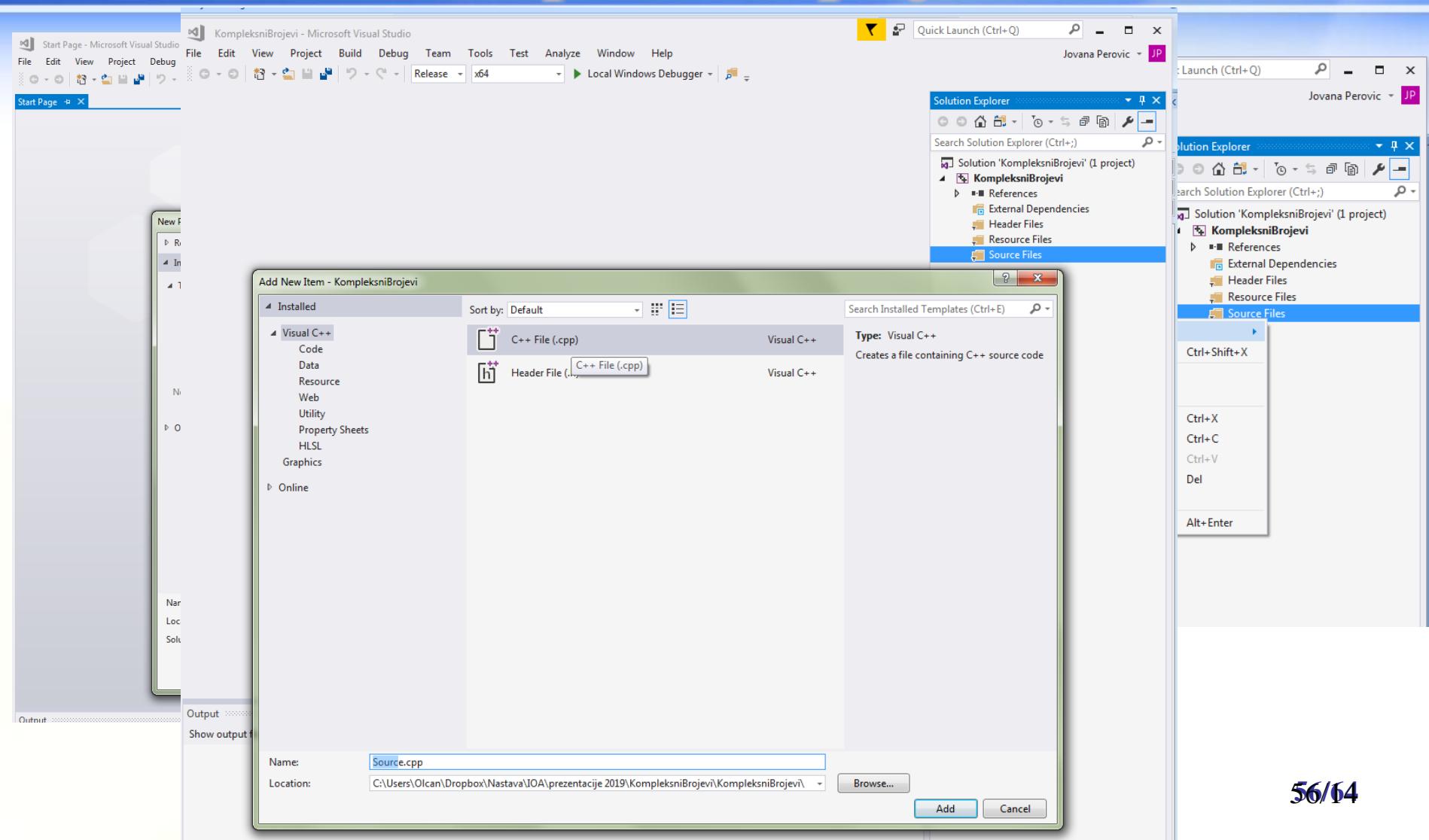
```
1 from scipy.optimize import minimize
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 "Rosenbrock function"
6 def fun(x):
7     return (1-x[0])**2.0 + 100*(x[1]-x[0]**2)**2
8
9
10 "crtanje"
11 x = np.arange(-2, 2, 0.11)
12 y = np.arange(-1, 3, 0.11)
13 x,y = np.meshgrid(x, y)
14 z = np.log10(fun([x,y]))
15
16 h = plt.contourf(x,y,z)
17 plt.scatter(1.0, 1.0, facecolor='red')
18 plt.xlabel('x')
19 plt.ylabel('y')
20 plt.colorbar(h)
21 print(fun([1,1]))
22
23 "trazenje minimuma"
24 x0 = np.array([0,0])
25 res = minimize(fun, x0, method='Nelder-Mead', options = {'ftol':1e-8,'disp': True, 'maxfev':1e4})
26 xmin = np.array(res.x)
27 print("xmin = (",xmin[0], ", ", xmin[1],")")
```

Optimization terminated successfully.  
Current function value: 0.000000  
Iterations: 79  
Function evaluations: 146  
xmin = ( 1.0000043858986165 , 1.0000106409916478 )

55/64

# Visual Studio 2017

## Отварање пројекта



# Рад са комплексним бројевима

The screenshot shows the Microsoft Visual Studio interface with a project named "KompleksniBrojevi". The left pane displays the "Source.cpp" file containing C++ code for performing arithmetic operations on complex numbers. The right pane shows the terminal window displaying the program's output.

```
#include <stdio.h>
#include <iostream>
#include <complex>

using namespace std;

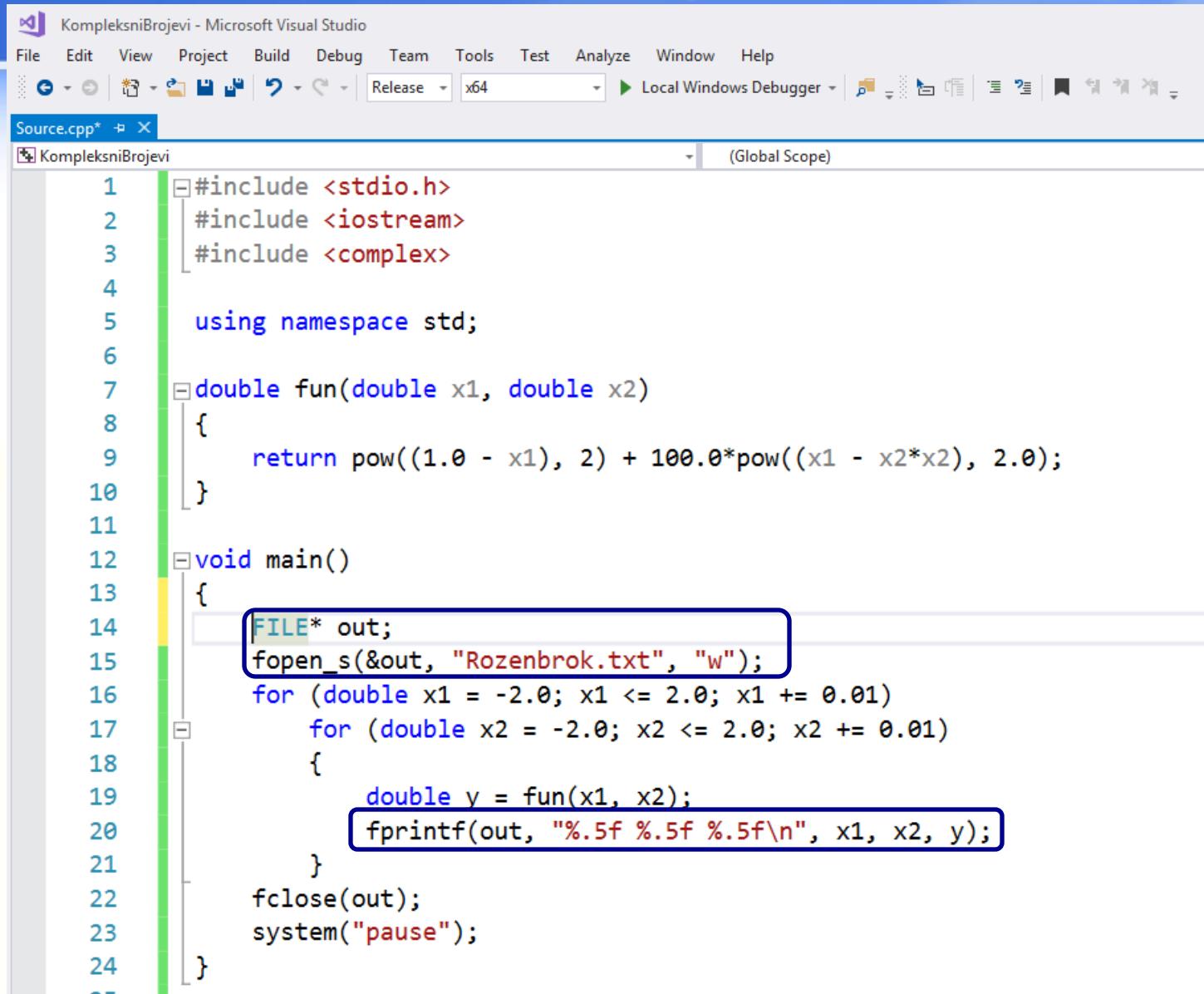
void main()
{
    //imaginarna jedinica
    complex<double> i;
    i.real(0.0);
    i.imag(1.0);

    complex<double> z1, z2, add, prod, div;
    double re, im;
    scanf_s("%lf %lf", &re, &im);
    z1.real(re);
    z1.imag(im);
    scanf_s("%lf %lf", &re, &im);
    z2.real(re);
    z2.imag(im);
    add = z1 + z2;
    prod = z1*z2;
    div = z1 / z2;
    printf("zbir: (%.2f, %.2f)\n", real(add), imag(add));
    printf("proizvod: (%.2f, %.2f)\n", real(prod), imag(prod));
    printf("kolicnik: (%.2f, %.2f)\n", real(div), imag(div));
    system("pause");
}
```

The terminal window output is:

```
1
2
3
4
5
zbir: <-1.00, 7.00>
proizvod: <-12.00, 1.00>
kolicnik: <0.28, -0.31>
Press any key to continue . . .
```

# Уписивање у датотеку



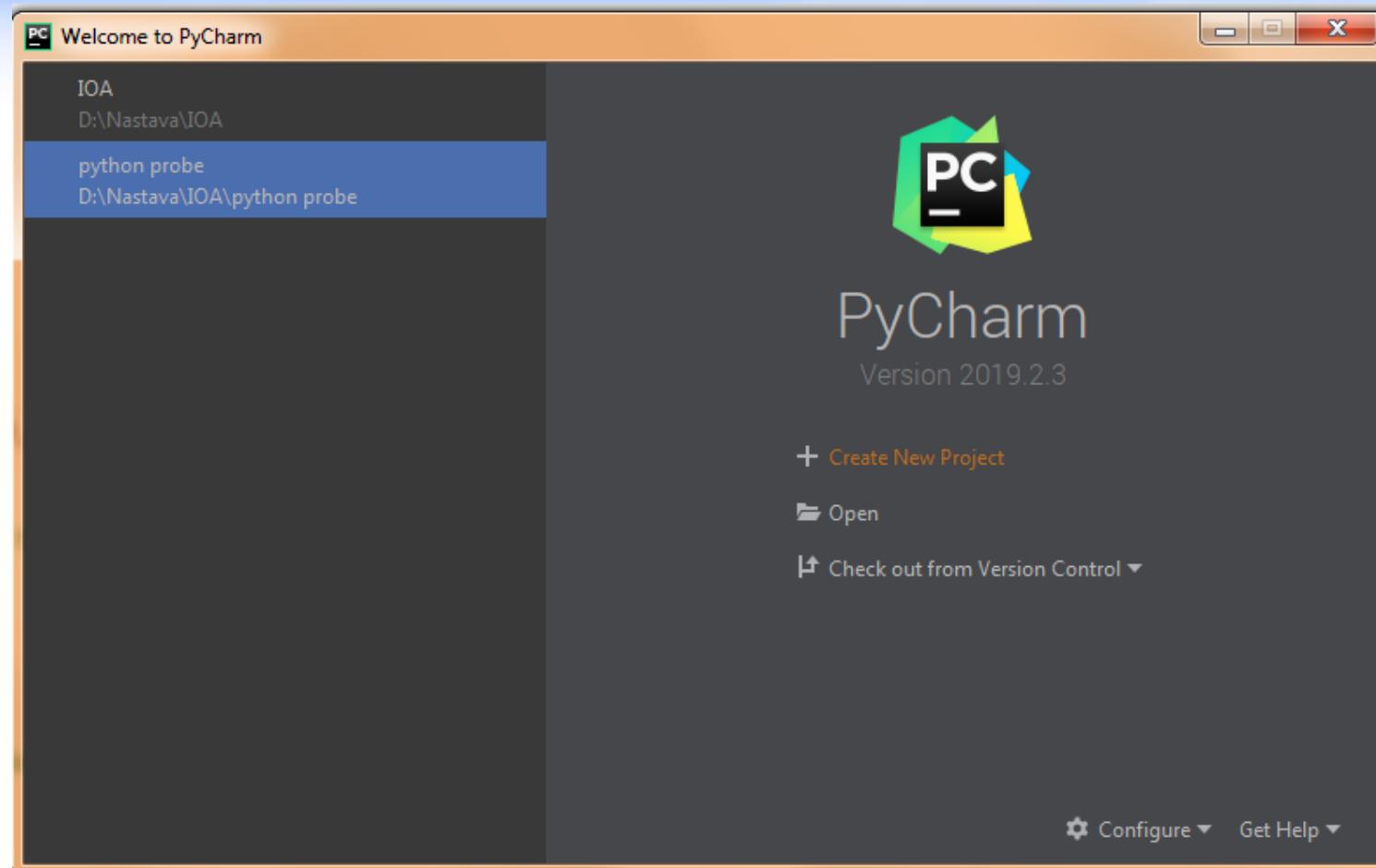
The screenshot shows the Microsoft Visual Studio interface with the title bar "KompleksniBrojevi - Microsoft Visual Studio". The menu bar includes File, Edit, View, Project, Build, Debug, Team, Tools, Test, Analyze, Window, and Help. The toolbar below has icons for file operations like Open, Save, and Build. The status bar indicates "Release x64 Local Windows Debugger". The code editor window displays "Source.cpp\*" under the project "KompleksniBrojevi". The code itself is as follows:

```
1 #include <stdio.h>
2 #include <iostream>
3 #include <complex>
4
5 using namespace std;
6
7 double fun(double x1, double x2)
8 {
9     return pow((1.0 - x1), 2) + 100.0*pow((x1 - x2*x2), 2.0);
10 }
11
12 void main()
13 {
14     FILE* out;
15     fopen_s(&out, "Rozenbrok.txt", "w");
16     for (double x1 = -2.0; x1 <= 2.0; x1 += 0.01)
17         for (double x2 = -2.0; x2 <= 2.0; x2 += 0.01)
18         {
19             double y = fun(x1, x2);
20             fprintf(out, "%.5f %.5f %.5f\n", x1, x2, y);
21         }
22     fclose(out);
23     system("pause");
24 }
```

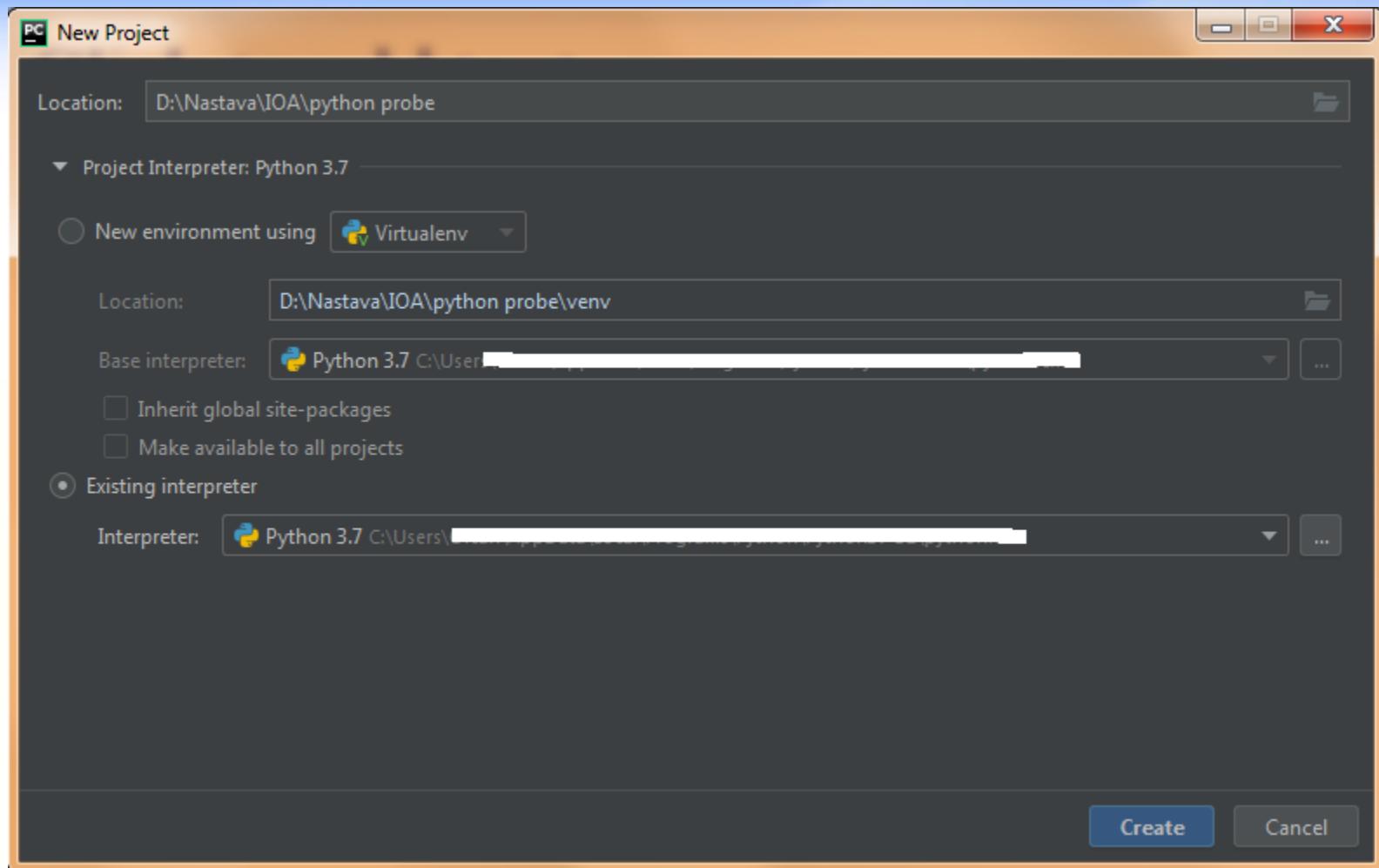
The lines of code from 14 to 20 are highlighted with a red rectangle, and the line from 20 to 21 is highlighted with a blue rectangle, indicating specific sections of the code being discussed.

# Python 3.7

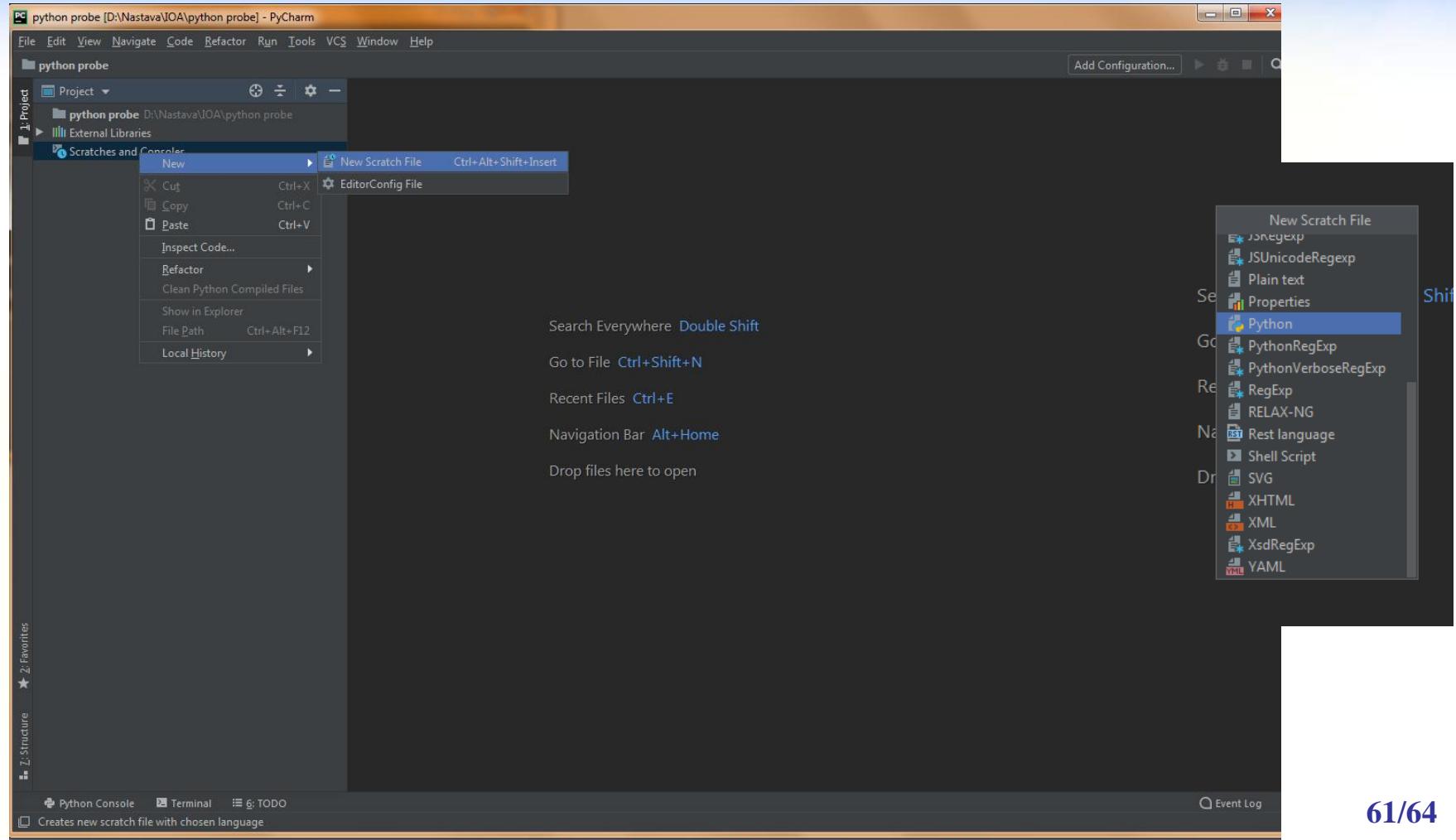
# (PyCharm окружење)



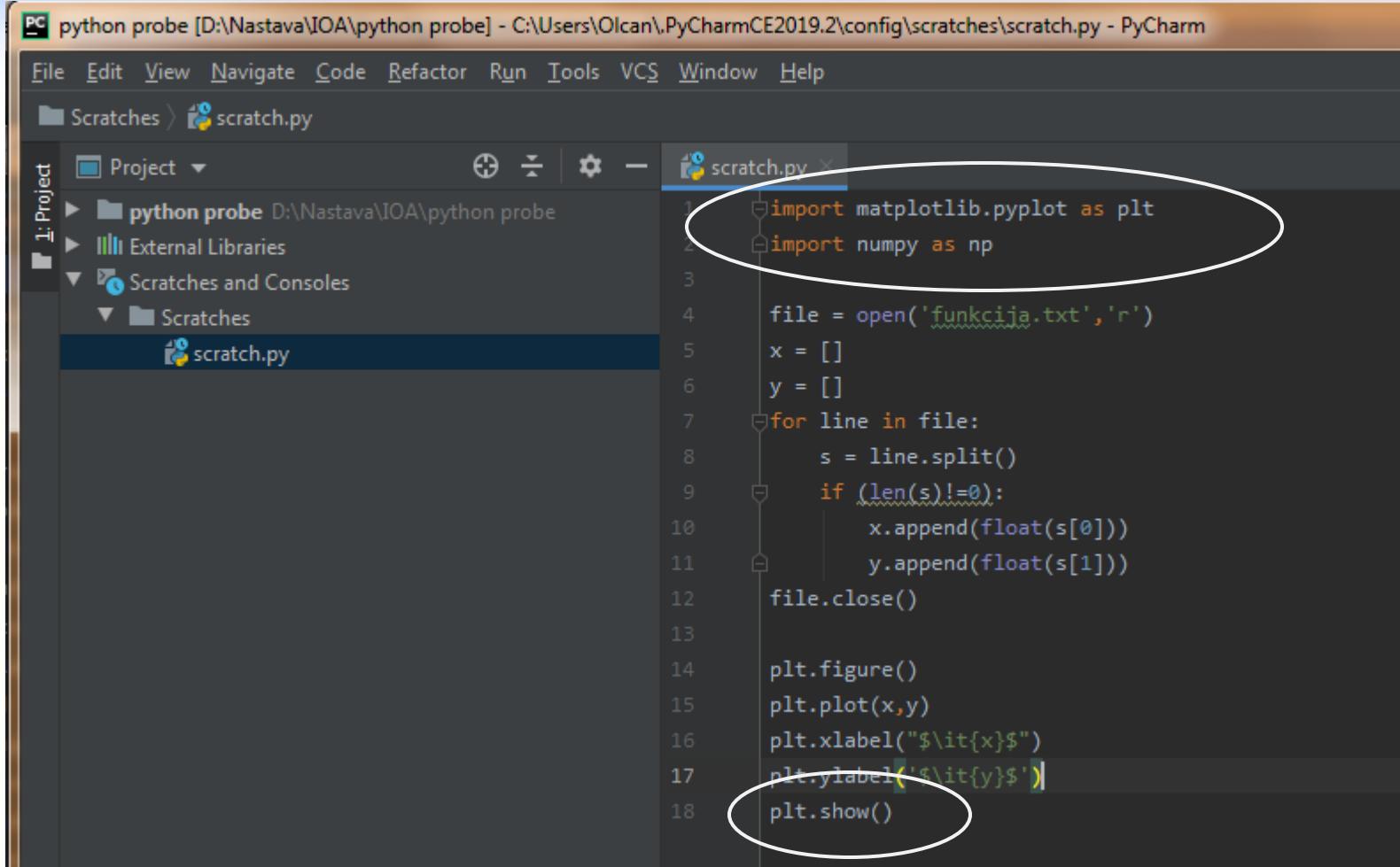
# Нови пројекат



# New scratch file



# Додавање библиотека

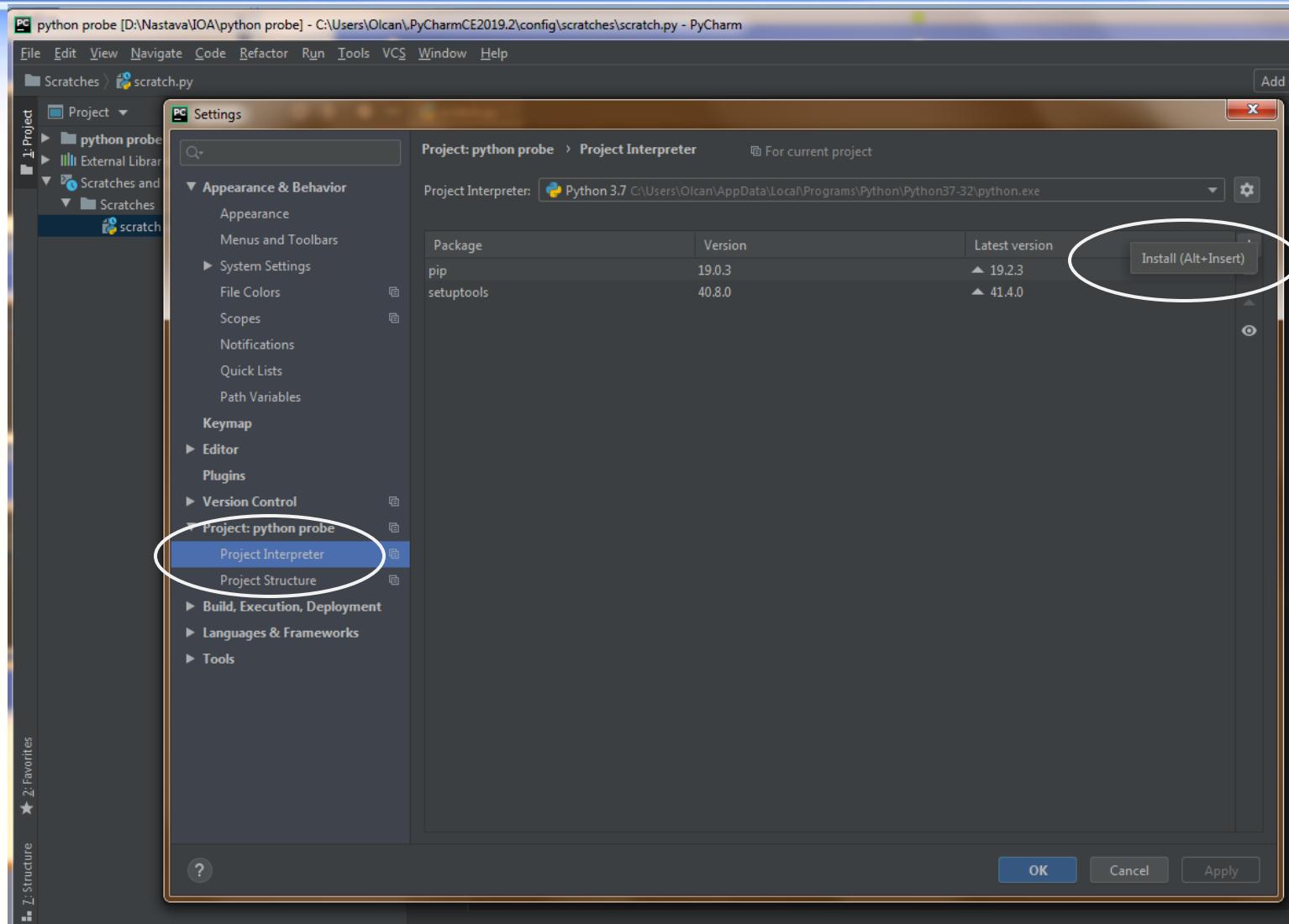


The screenshot shows the PyCharm IDE interface with the following details:

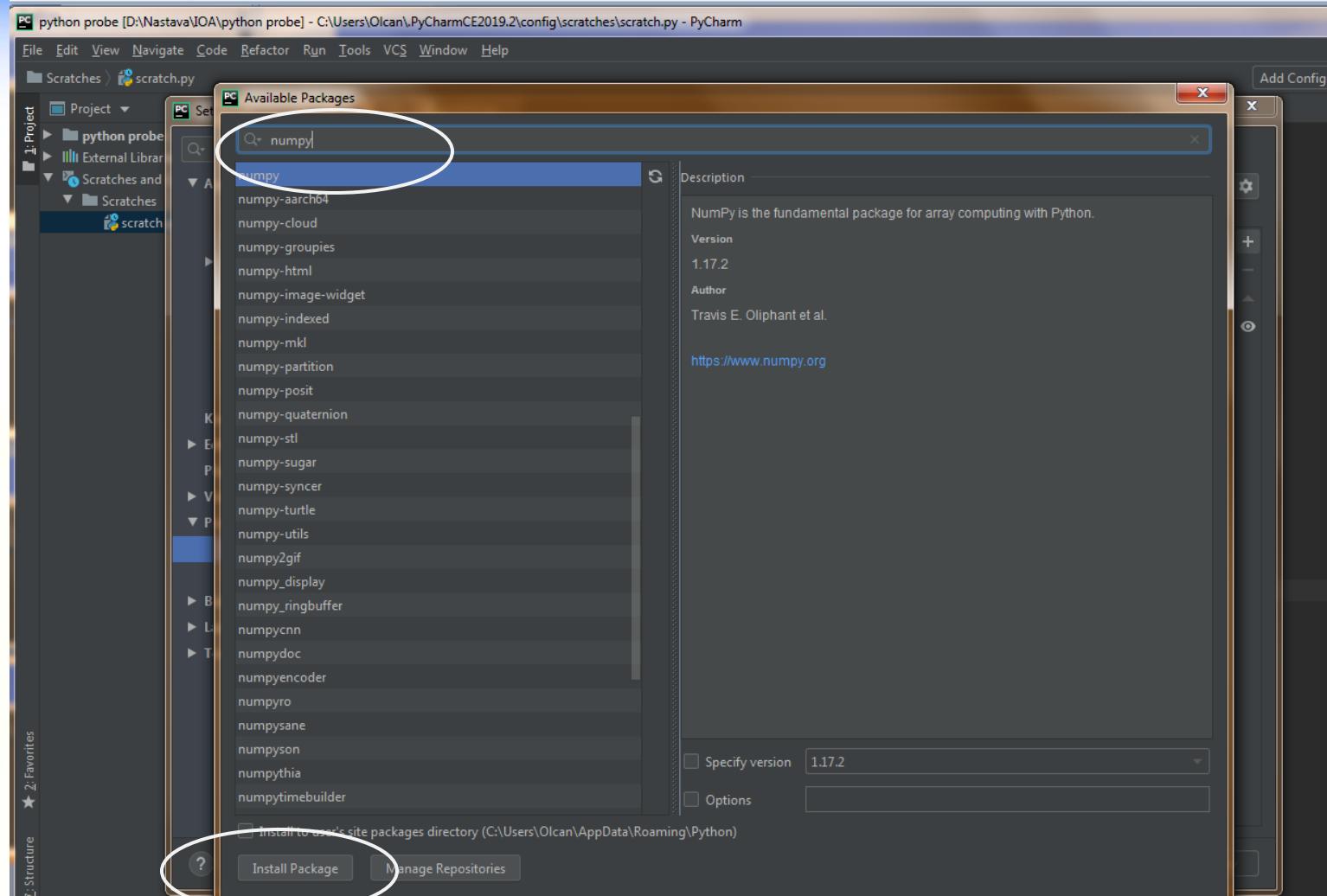
- Title Bar:** python probe [D:\Nastava\IOA\python probe] - C:\Users\Olcan\PyCharmCE2019.2\config\scratches\scratch.py - PyCharm
- Menu Bar:** File Edit View Navigate Code Refactor Run Tools VCS Window Help
- Toolbars:** Standard and Project
- Project Explorer:** Shows a project named "python probe" at D:\Nastava\IOA\python probe, an External Libraries section, and a Scratches and Consoles section containing a "Scratches" folder with a file named "scratch.py".
- Code Editor:** Displays the content of "scratch.py":

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 file = open('funkcija.txt','r')
5 x = []
6 y = []
7 for line in file:
8     s = line.split()
9     if (len(s)!=0):
10         x.append(float(s[0]))
11         y.append(float(s[1]))
12 file.close()
13
14 plt.figure()
15 plt.plot(x,y)
16 plt.xlabel("$\it{x}$")
17 plt.ylabel("$\it{y}$")
18 plt.show()
```
- Annotations:** Two white ovals highlight specific parts of the code:
  - The first oval encloses the two import statements: `import matplotlib.pyplot as plt` and `import numpy as np`.
  - The second oval encloses the line `plt.ylabel("$\it{y}$")`.

# Добавање библиотека



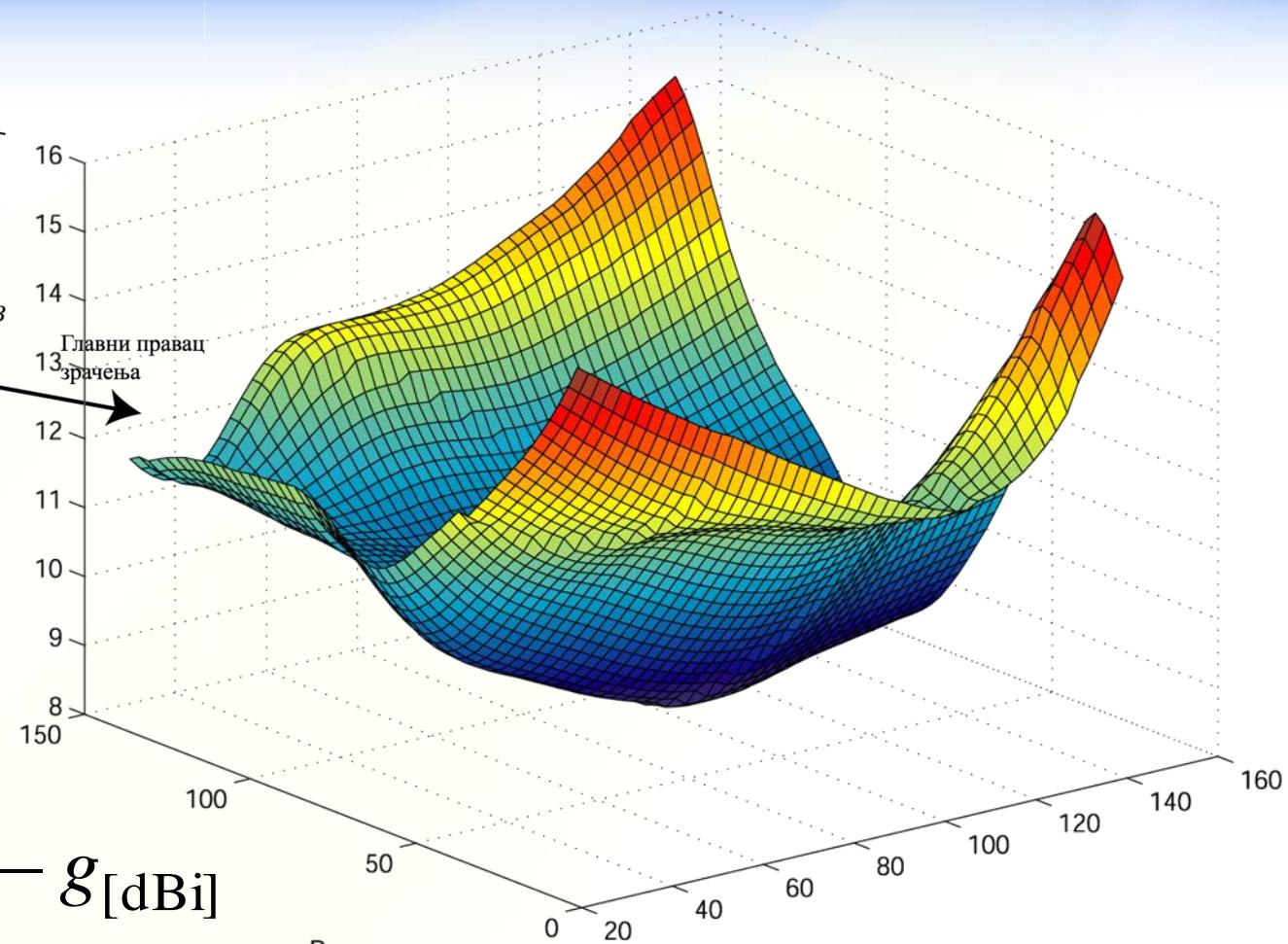
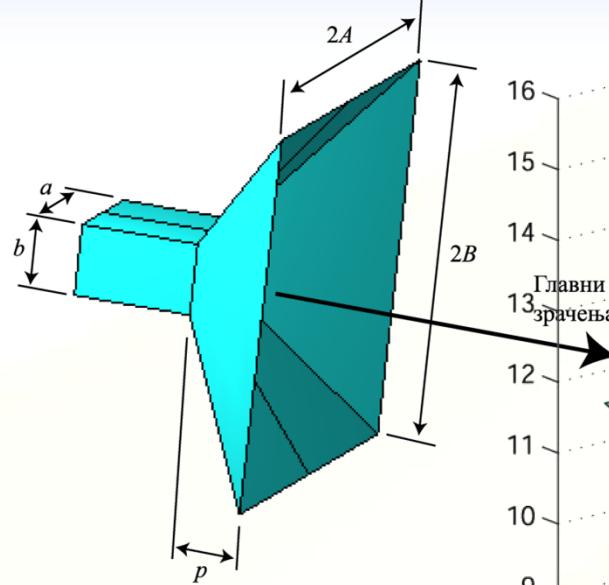
# Инсталација библиотека



# Оптимизациона функција

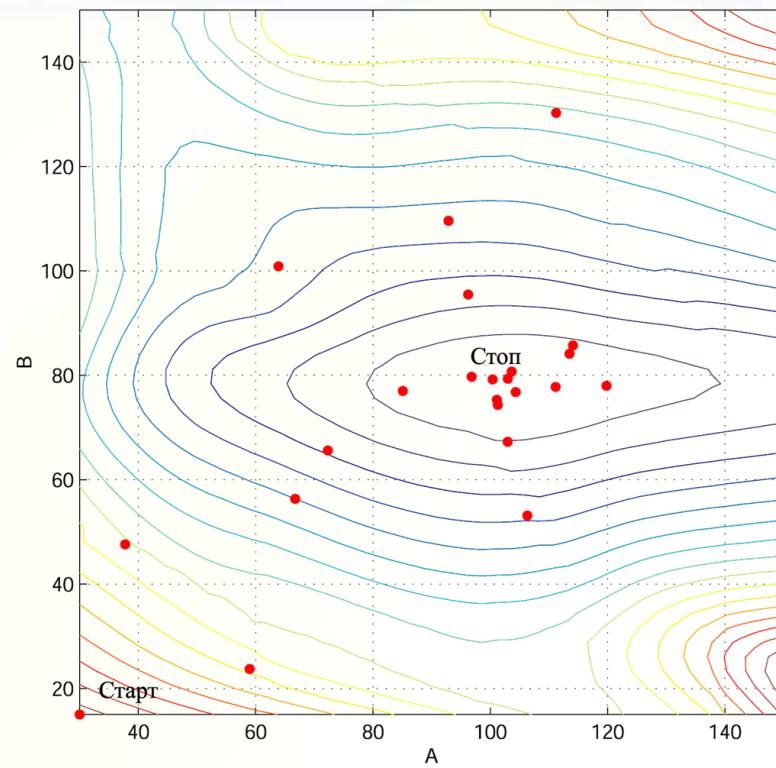
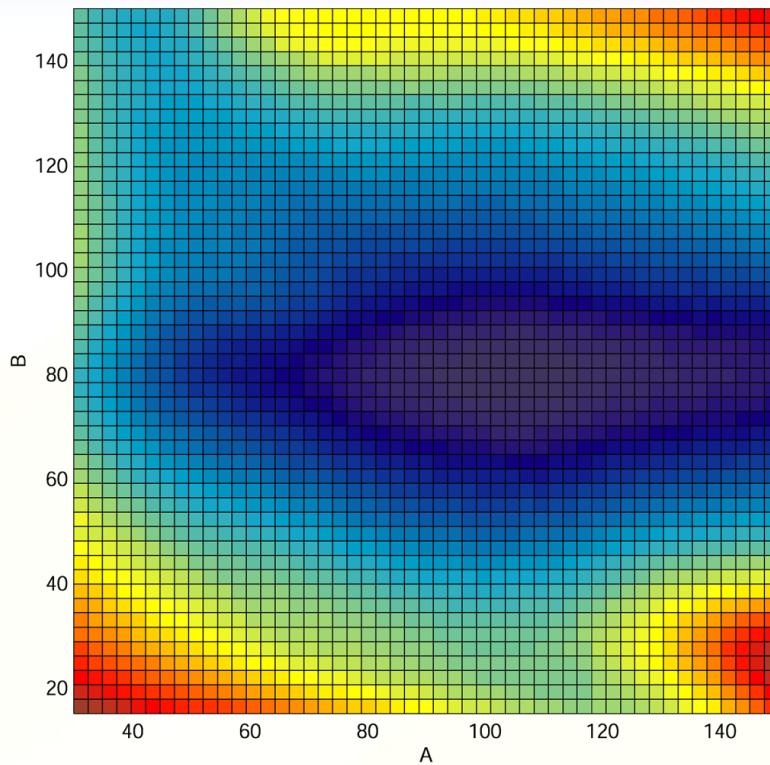
- Оптимизациону функцију,  $f(\mathbf{x})$ , дефинише онај ко решава проблем!
- Избор  $f(\mathbf{x})$  може олакшати или отежати решавање
- Увек је потребно уградити СВЕ што зnamо о проблему
- У неким ситуацијама постоји и шум
  - Нумеричка (ЕМ) анализа и грешка заокруживања
  - TSP и гужва у саобраћају
  - SAT и промењени бити услед грешака у преносу бита

# Пример из праксе једне оптимизационе функције (NLP)

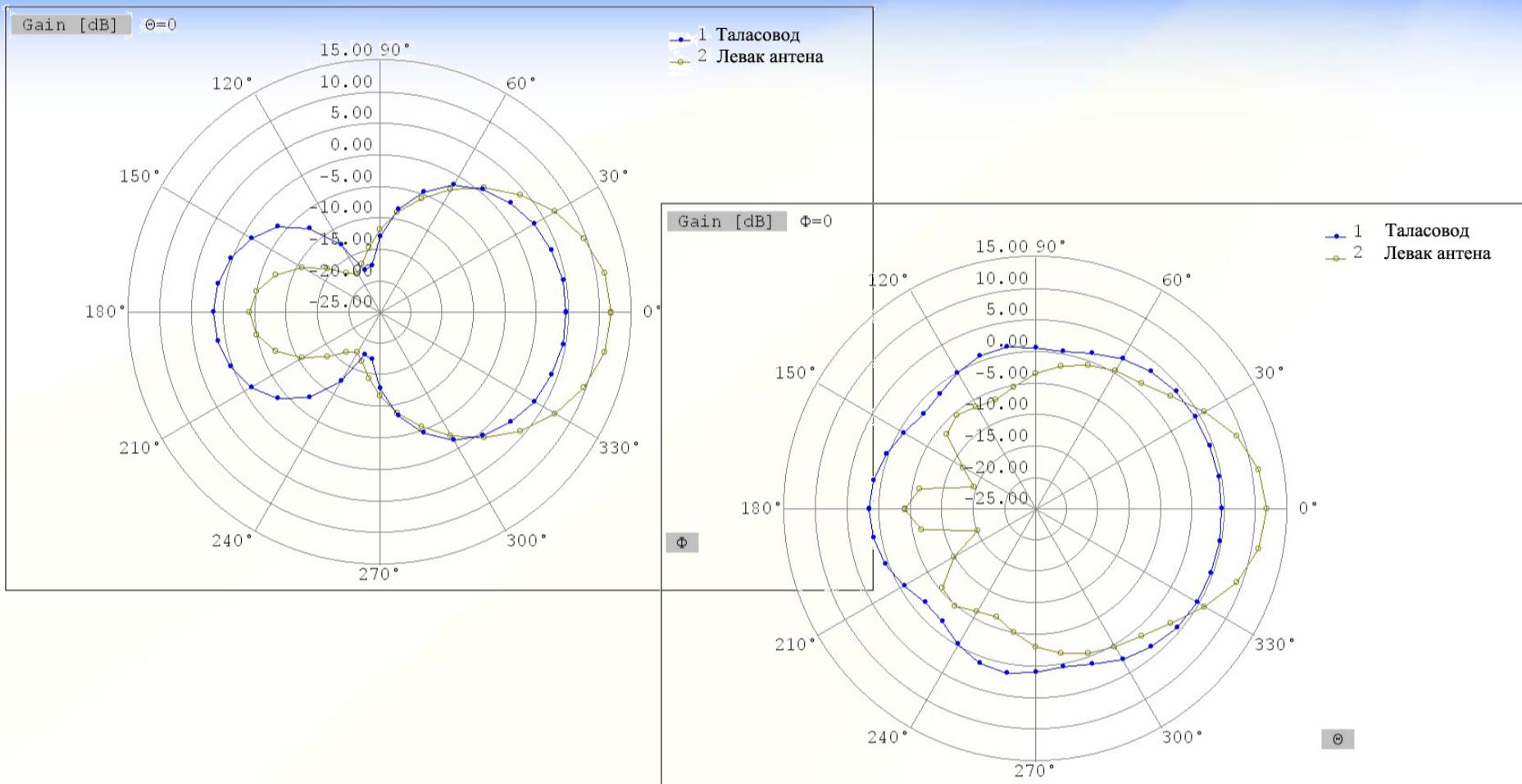


$$f_{\text{opt}}(A, B) = 20 - g[\text{dBi}]$$

# Описна функција и оптимални отвор левак антене



# Оптимална левак антена



# Како проћенити особине и перформансе опт. алгоритма?

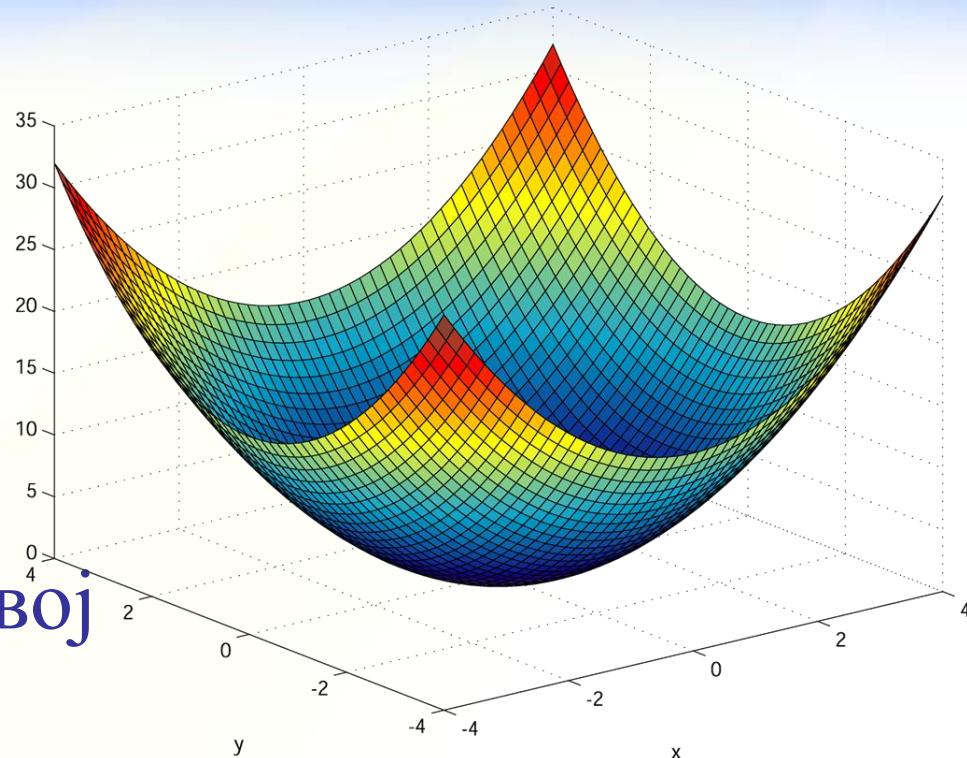
- Ситуација #1 (инжењерска пракса):  
постављен је оптимизациони проблем и потребно је решити га
  - бирали смо алгоритам на основу претходног искуства и информација које имамо о датом проблему
  - пробали сме све доступне оптимизационе алгоритме и користили оне који имају најбоље перформансе
- Ситуација #2 (истраживање / наука):  
потребно је сагледати особине оптимизационог алгоритма
  - тестирали смо алгоритме на познатим оптимизационим функцијама (проблемима) и поредили перформансе оптимизационих алгоритма
- Потребне су нам познате оптимизационе функције (познати оптимизациони проблеми) са различитим особинама, када разматрамо алгоритме

# Аналитичке (NLP) тест функције: сума квадрата

- Аналитички израз за  $D$  димензија

$$f(\mathbf{x}) = \sum_{i=1}^D x_i^2$$

- Једноставна: мин. по свакој променљивој
- Минимум познат: координатни почетак (или нуле полинома)

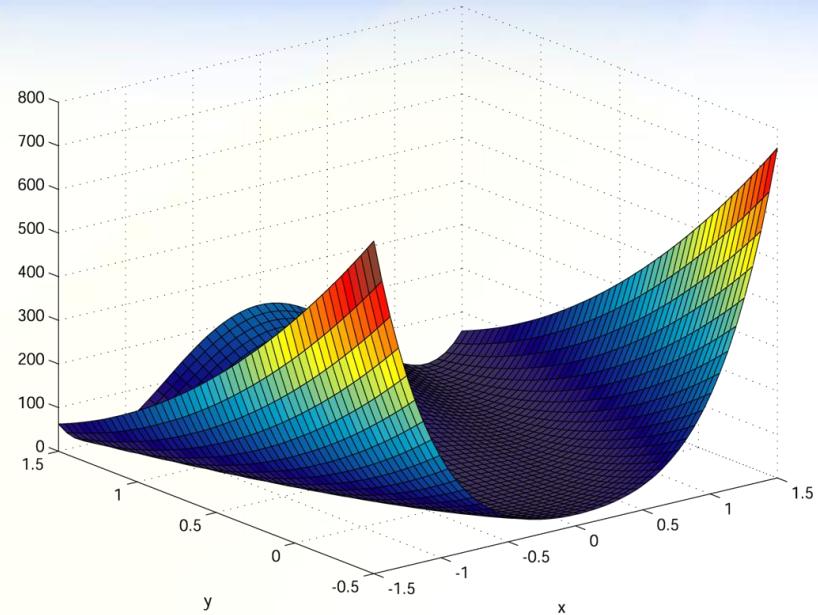


# Розенброкова функција

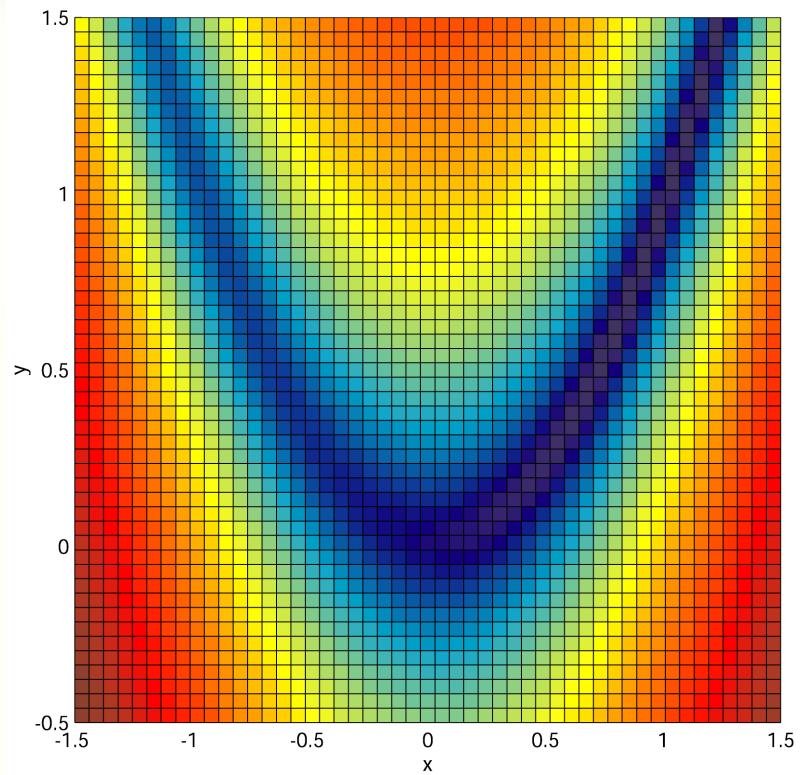
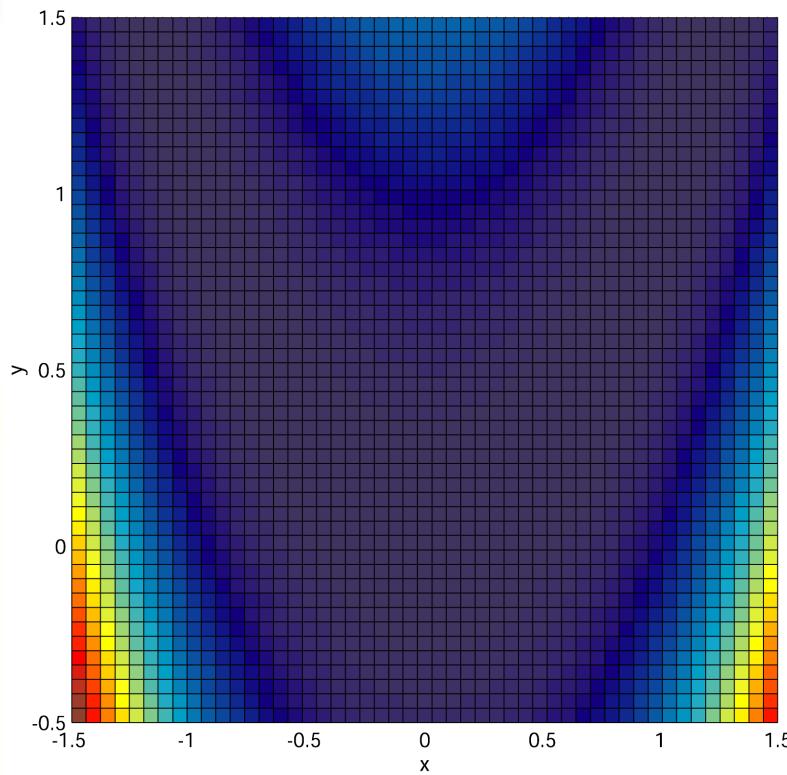
- Аналитички израз за  $D$  димензија

$$f(\mathbf{x}) = \sum_{i=1}^{D-1} \left( 100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2 \right)$$

- Глобални минимум у тачки  $(1, 1, \dots, 1)$
- Број локалних минимума за функцију са више од две димензије није познат
- Спрега између променљивих

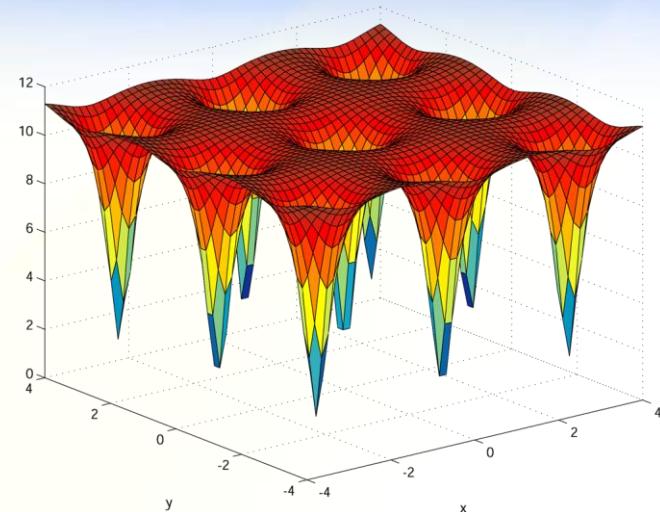


# Розенброкова функција: поглед одозго и лог-размера

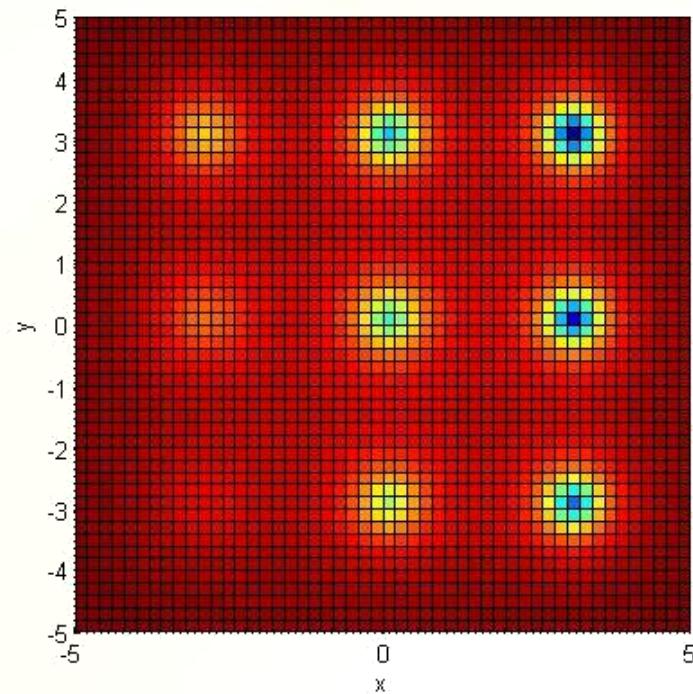
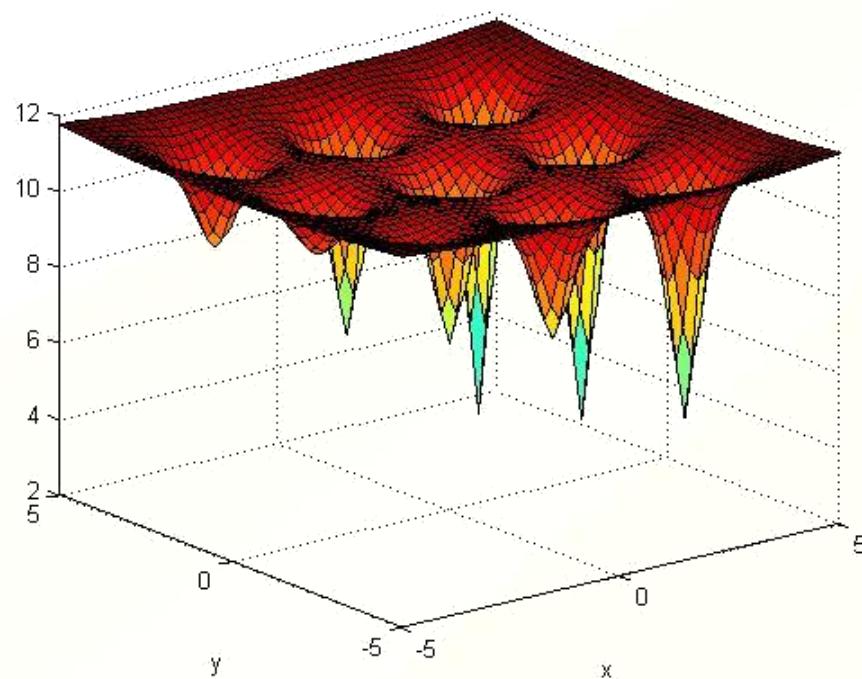


# Шекелова функција (Fox-hole function)

- Аналитички израз
$$f(\mathbf{x}) = K - \sum_{i=1}^M \left( (\mathbf{x} - \mathbf{a}_i)^T (\mathbf{x} - \mathbf{a}_i) + c_i \right)^{-1}$$
- $\mathbf{x}$  је  $D$ -димензиони вектор
- $\mathbf{a}_i$  је вектор са координатама  $i$ -тог локалног минимума
- $c_i$  је константа обрнуто пропорционална дубини  $i$ -тог локалног минимума
- $M$  је укупан број минимума функције
- $K$  је константа која контролише вредност функције између минимума



# Минимуми различитих дубина



# G2 функција

- Максимизација функције

$$G2(\mathbf{x}) = \left| \frac{\sum_{i=1}^D \cos^4(x_i) - 2 \prod_{i=1}^D \cos^2(x_i)}{\sqrt{\sum_{i=1}^D i \cdot x_i^2}} \right|$$

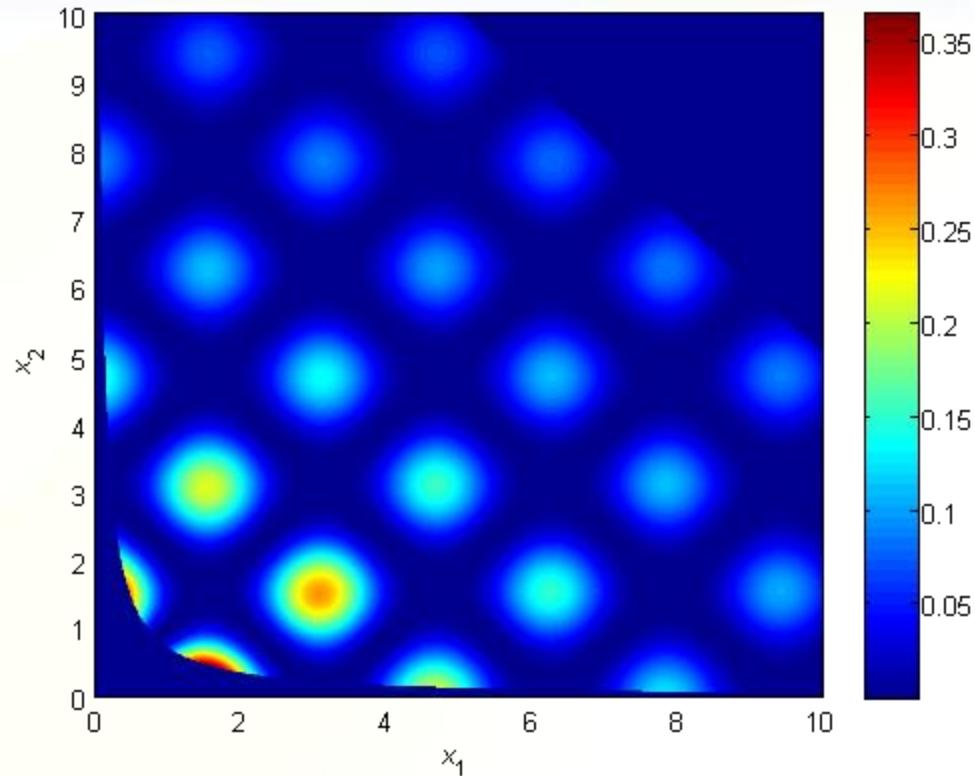
- Под условима:

$$\prod_{i=1}^D x_i \geq 0,75 \quad \sum_{i=1}^D x_i \leq 7,5D \quad 0 \leq x_i \leq 10, \quad i \in 1,2,\dots,D$$

- Нула ако нису испуњени услови

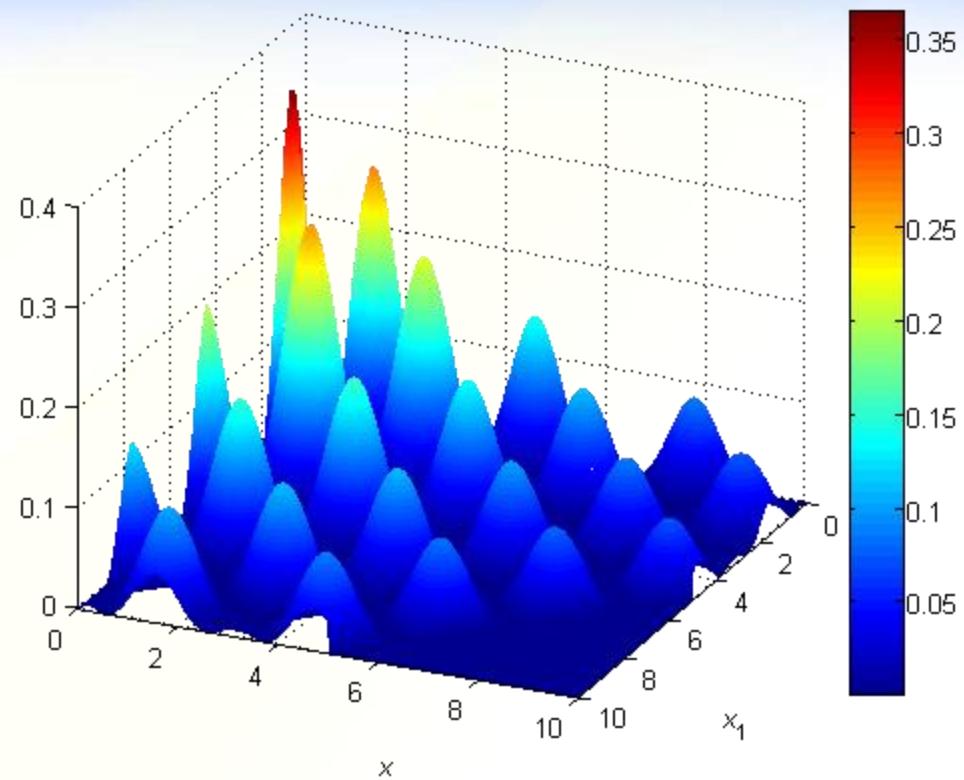
# Функција G2 са 2 променљиве

- Глобални максимум непознат у општем случају
- Око координатног почетка



# G2: изразито тешка функција за оптимизацију

- Изразито “брдовит” терен
- Врло незгодна функција за оптимизацију
- Нотација, друга од 11 тешких функција



# Примери дискретних функција: $f_1$ врло тешка, $f_2$ лакше решива

$$f_1(x_1, x_2, x_3, x_4) = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \quad f_2(x_1, x_2, x_3, x_4) = \sum_{k=1}^4 x_k$$

$x_1$	$x_2$	$x_3$	$x_4$	$f_1$	$f_2$
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	1
0	0	1	1	0	2
0	1	0	0	0	1
0	1	0	1	0	2
0	1	1	0	0	2
0	1	1	1	0	3
1	0	0	0	0	1
1	0	0	1	0	2
1	0	1	0	0	2
1	0	1	1	0	3
1	1	0	0	0	2
1	1	0	1	0	3
1	1	1	0	0	3
1	1	1	1	1	4

# Формирање оптимизационе функције и норме

- Оптимизациона функција је нека функција разлике величина које описују решења
- Примери:
  - разлика између перформанси жељеног и пројектованог електричног уређаја
  - разлика између мерених резултата и функције која их фитује
  - разлика између жељене и добијене цене производа
  - разлика између жељеног и оствареног управљања
  - итд.
- Да ли узети апсолутну вредност разлике, квадрат разлике или неки други степен?
- У оптимизационој функцији у пракси често се појављује **норма** разлике жељеног и оствареног

# Норма: дефиниција

- За задати векторски простор  $V$ , норма је функција  $f: V \rightarrow \mathbb{R}$ , која задовољава следеће услове  $V: \mathbf{x}, \mathbf{y}, \mathbb{R}: a$ 
  - $f(a\mathbf{x}) = |a| f(\mathbf{x})$
  - $f(\mathbf{y} + \mathbf{x}) \leq f(\mathbf{y}) + f(\mathbf{x})$
  - $f(\mathbf{x}) = 0 \iff \mathbf{x} = 0$
- Из претходних услова следи
  - $f(\mathbf{x}) \geq 0$

# Норма: означавање и примери

- $\mathbf{x}$  је  $D$  димензиони вектор разлике између жељеног и пронађеног решења (оптимизација)
- Норма се најчешће означава као

$$\|\mathbf{x}\| := f(\mathbf{x})$$

- $L_1$  (taxicab, Manhattan norm)  
растојање које такси прелази на мрежи

$$\|\mathbf{x}\|_1 := \sum_{k=1}^D |x_k|$$

# Норма: примери и генерализација

- $L_2$  (Еуклидовска норма, растојање)

$$\|\mathbf{x}\|_2 := \sqrt{\sum_{k=1}^D |x_k|^2}$$

- Генерализација:  $L_p$      $\|\mathbf{x}\|_p := \left( \sum_{k=1}^D |x_k|^p \right)^{\frac{1}{p}}, p \geq 1$

- Maximum norm     $\|\mathbf{x}\|_\infty := \max \{|x_1|, |x_2|, \dots, |x_D|\}$

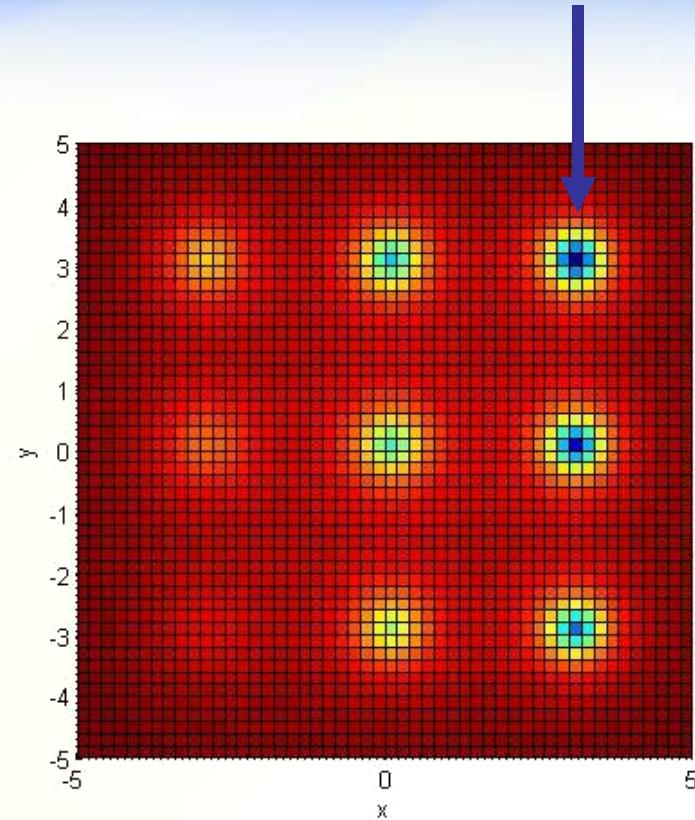
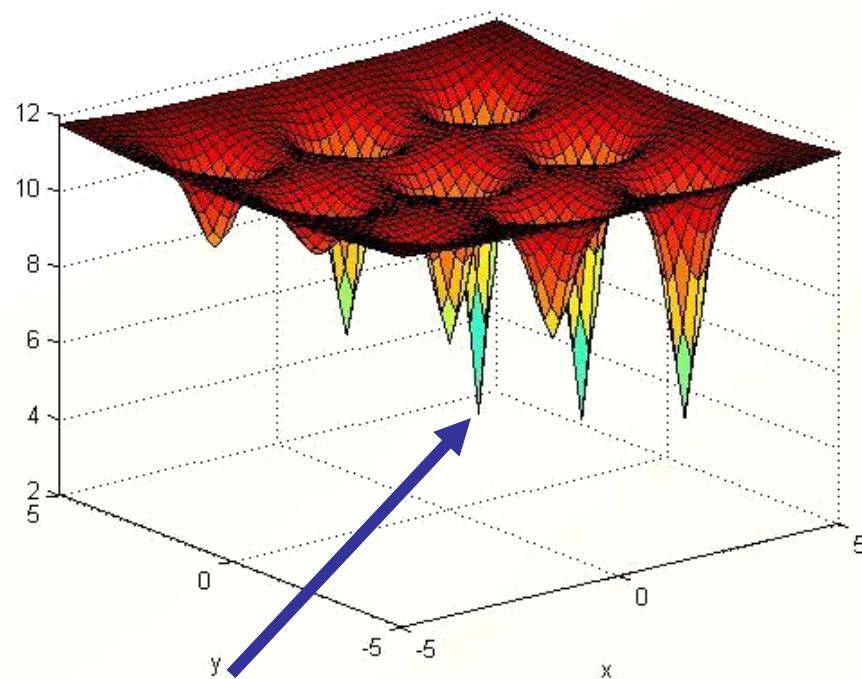
# Дефиниција оптимизационог проблема са једним критеријумом

- Пронађи минимум функције  $f$   
 $\text{minimize}\{ f(\mathbf{x}) \}$   
 $\mathbf{x} = (x_1, x_2, \dots x_D)$   
 $x_k \in R \quad k = 1, 2, \dots D$
- Под условима  
 $\phi_k(\mathbf{x}) = 0 \quad k = 1, 2, \dots E$   
 $\psi_k(\mathbf{x}) \leq 0 \quad k = 1, 2, \dots L$
- Минимизација и максимизација: подразумевамо минимизацију, уколико се не нагласи другачије
- Еквиваленција:  $\text{minimize}\{ f(\mathbf{x}) \} = \text{maximize}\{ -f(\mathbf{x}) \}$

# Глобални минимум: формална дефиниција

- Оптимизациони простор  $S$
- Домен (смислених) решења  $F$ ,  $F \subseteq S$
- Оптимизациона функција  $f(\mathbf{x})$
- Глобални минимум,  $\mathbf{x}_g$ , опт. проблема је
$$\forall \mathbf{y} \in F, f(\mathbf{x}_g) \leq f(\mathbf{y})$$
- Глобални минимум је најбоље могуће решење
- Да ли постоји само један глобални минимум?
  - Може постојати један или више глобалних минимума

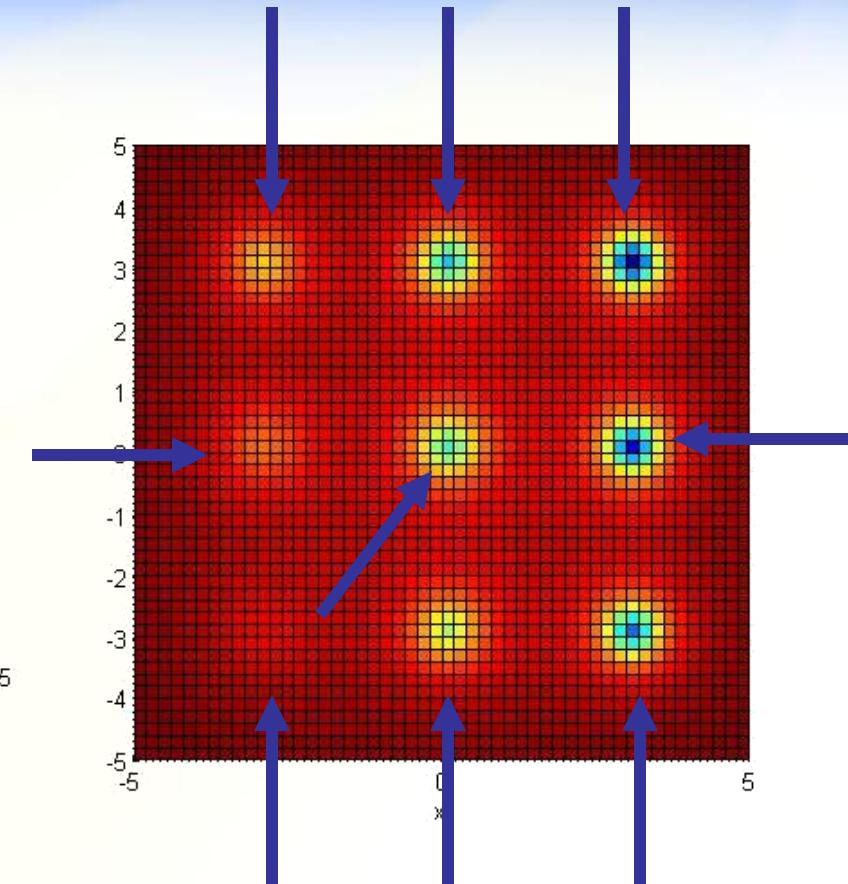
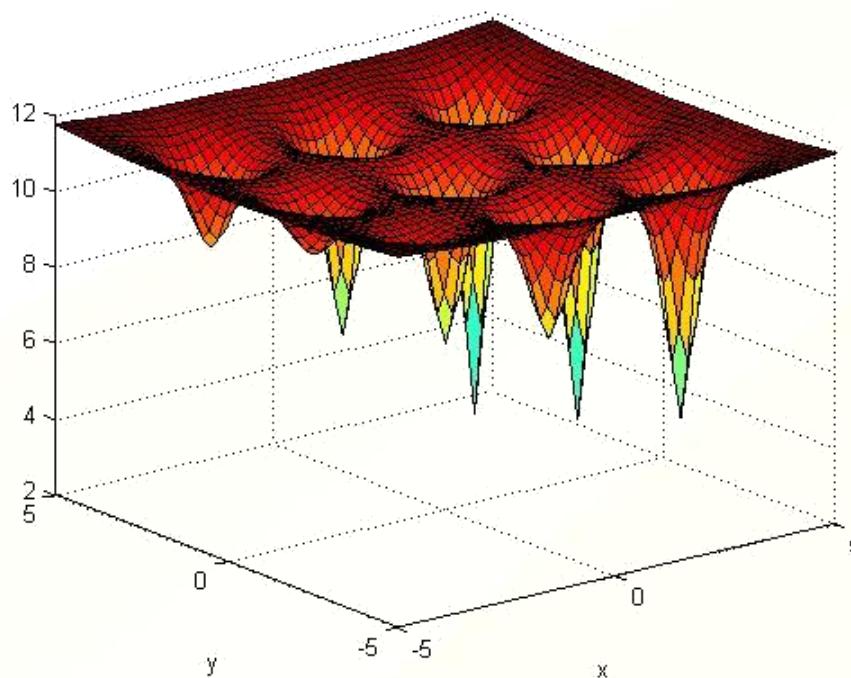
# Пример глобалног минимума



# Локално решење: формална дефиниција

- Епсилон околина решења  $\mathbf{x}$  је  $N(\mathbf{x}) = \{\mathbf{y} \in S, dist(\mathbf{x}, \mathbf{y}) \leq \varepsilon\}$
- За NLP класу проблема:  
Еуклидовско растојање  $dist(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{k=1}^D (x_k - y_k)^2}$
- За SAT класу проблема:  
Хамингово растојање  $dist(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^D x_k \oplus y_k$
- За TSP класу проблема:  
број различитих прелазака  $dist(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{D-1} \begin{cases} 1, & x_k \leftrightarrow x_{k+1} \in \mathbf{y} \\ 0, & x_k \leftrightarrow x_{k+1} \notin \mathbf{y} \end{cases}$
- $dist(\mathbf{x}, \mathbf{y})$  може да се дефинише на много начина!
- Локални минимум  $\mathbf{x}$ , у околини  $N(\mathbf{x})$ :  
 $f(\mathbf{x}) \leq f(\mathbf{y}), \forall \mathbf{y} \in N(\mathbf{x})$

# Примери локалних минимума ( зависи од околине! )



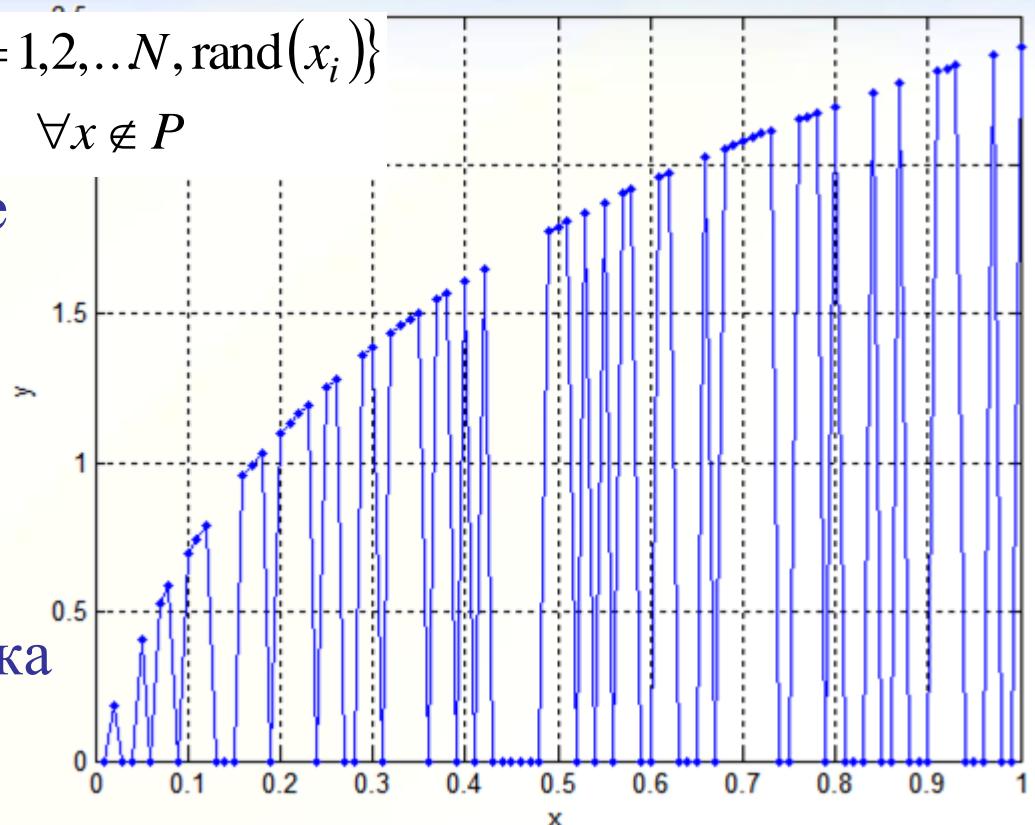
# NFL теорема

- NO-FREE-LUNCH theorem
  - Wolpert D.H.; Macready W.G. "No free lunch theorems for optimization" *Evolutionary Computation IEEE Transactions on* vol.1 no.1 pp. 67-82 Apr 1997
  - уколико су све оптимизационе функције једнако вероватне сви опт. алгоритми су једнако (не)ефикасни
- Зашто бисмо онда разматрали различите оптимизационе алгоритме?

# Пример функције уз NFL теорему

$$f(x) = \begin{cases} 0, & P = \{i = 1, 2, \dots, N, \text{rand}(x_i)\} \\ \ln(1+10x), & \forall x \notin P \end{cases}$$

- Пример мало вероватне оптимизационе функције
- Овакав проблем није добро дефинисан
- Пример: оптимизација линка за пренос података који је недоступан у тренуцима који су случајно генерисани

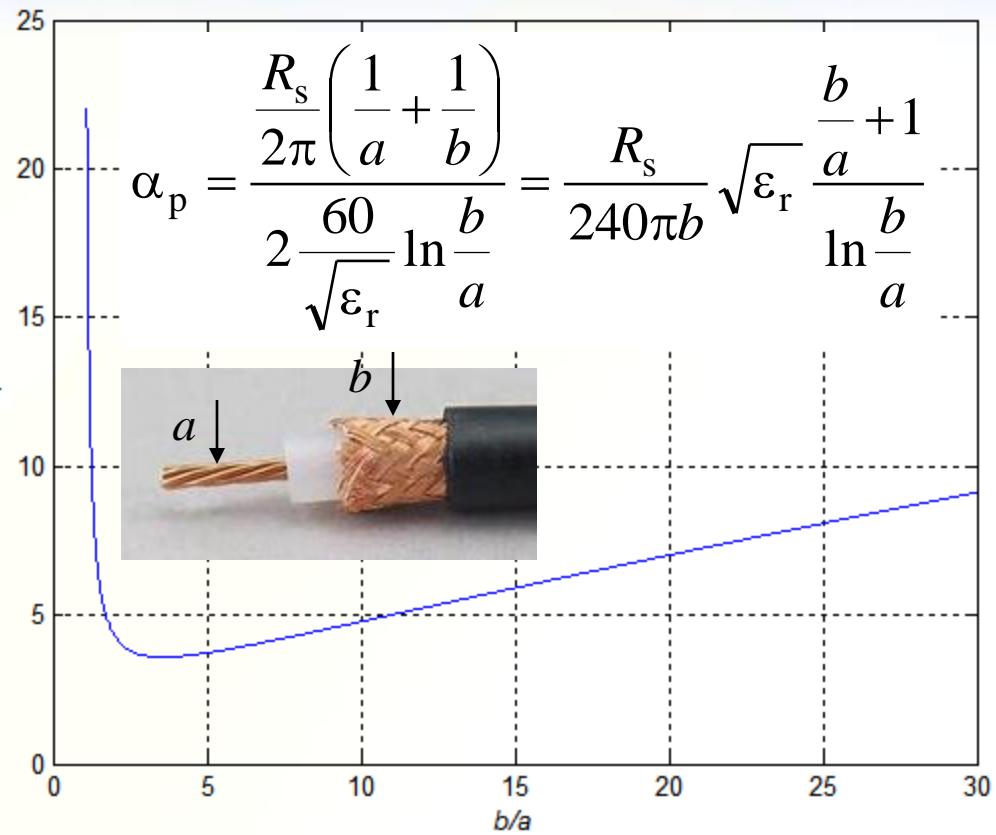


# Критички осврт на NFL теорему (познати докази када НЕ важи)

- D.W. Corne J.D. Knowles “Some multiobjective optimizers are better than the others ” *Proc. of IEEE CEC 2003* pp. 2506-2512.
  - For multiobjective optimization NFL does not hold
- C. Igel M. Toussaint “A No-Free-Lunch theorem for non-uniform distributions of target functions ” *J. Math. Model. Algorithms* 3(4) pp. 313-322 (2004)
  - Classes of functions relevant in practice are not likely to satisfy the NFL scenario
- A. Auger O. Teytaud “Continuous lunches are free plus the design of optimal optimization algorithms ” *Algorithmica* 57 2010 pp. 121-146.
  - For continuous domains NFL does not hold

# Пример инжењерског проблема: минимизација слабљења кабла

- Коаксијални кабл
  - полупречници  $a$  и  $b$
  - површинска отпорност  $R_s$
  - задато  $b$  (габарит)
- Пронађи  $b/a$  тако да је  $\alpha_p$  минимално
- Непрекидна и диференцијабилна опт. функција
- Постоји јасна узрочно-последична зависност побуде ( $b/a$ ) и одзива ( $\alpha_p$ )



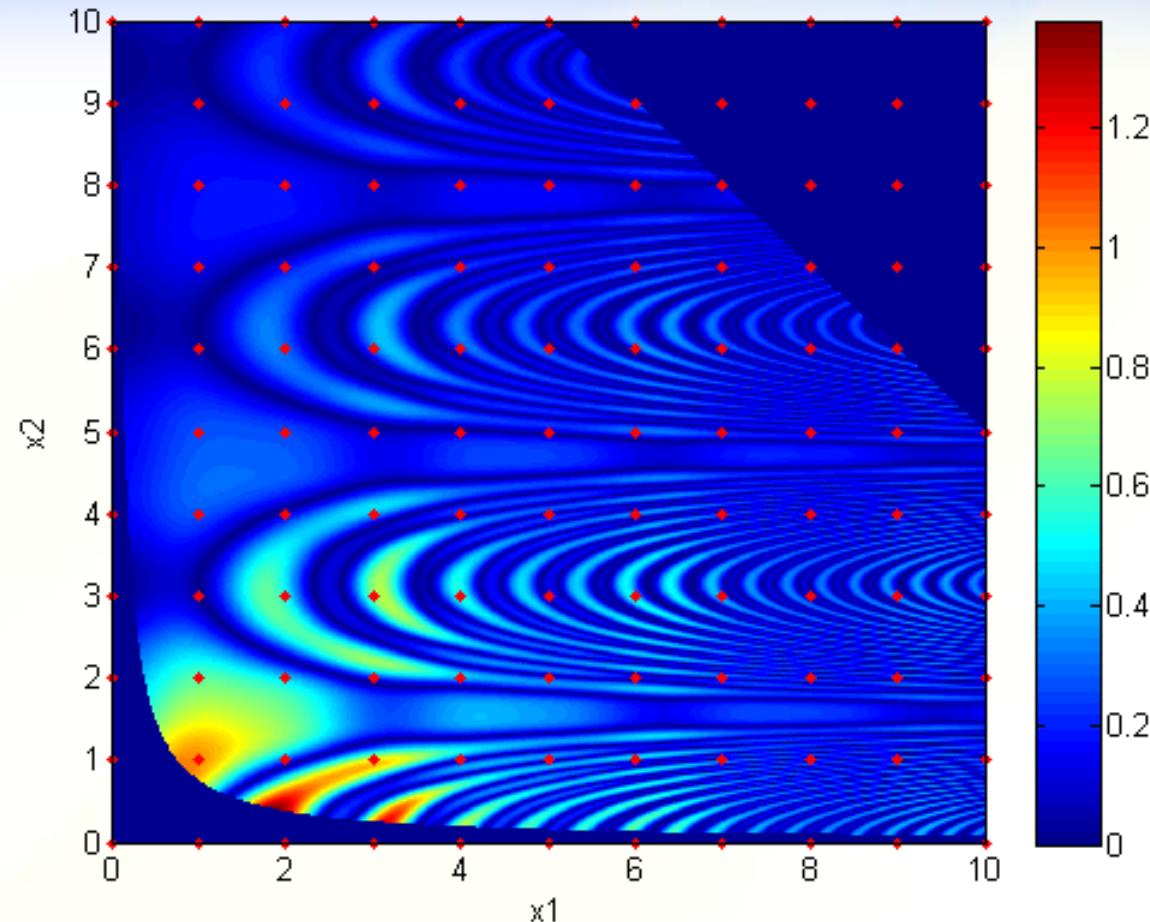
# Инжењерска пракса: неки алгоритми су БОЉИ

- Пракса:
  - оптимизациона функција је смислено формулисана
  - оптимизациони проблем је једнозначен
  - поновљивост везе улаз-излаз “црне кутије”
- Инжењерски проблем подразумева **узрочно-последичну везу**
  - Неке оптимизационе функције су много вероватније од других
- У инжењерској пракси:  
 неки оптимизациони алгоритми су знатно бољи од других алгоритама
  - У зависности од проблема који се решава

# Систематско претраживање: континуалне променљиве

- За сваки оптимизациони параметар задаје се број корака (или дужина)
- Вишедимензионална мрежа се формира у оптимизационом простору
- У сваком чвору мреже израчунава се оптимизациона функција
- За  $D$ -димензионални оптимизациони простор укупан број итерација је  $k_1 * k_2 * \dots * k_D$  где је  $k_i$  број корака по димензији  $i$
- Сложеност алгоритма је  $O(N^D)$   
 $N$  број корака по једној димензији  
 $D$  број димензија оптимизационог простора

# Систематско претраживање једне континуалне функције



# Систематско претраживање: дискретне променљиве

- Проверити СВА дискретна стања у оптимизационом простору
- Примери:
  - испитати сва дискретна стања прекидача
  - испитати све варијације са понављањем
  - испитати све комбинације без понављања
  - испитати све пермутације без понављања
  - ...
- Истинитосне таблице при испитивању таутологије у логици

# Варијације са понављањем (Систематско претраживање SAT)

- На колико начина се може распоредити  $n$  елемената на  $k$  места?
- Број начина распоређивања је  $n^k$
- Пун назив: варијација са понављањем  $k$ -те класе скупа  $X_n$  (Енг:  $n$ -tuples)
- Формирање као број са  $k$  цифара а цифре су  $1 \dots n$ :

1111    1112    ...    111n  
1121    1122    ...    112n  
...  
 $n$ nn1     $n$ nn2    ...     $n$ nnn

```
void variations_with_repetition(int n, int k)
{
    int q;
    int *P = new int [k];
    for(int i=0; i<k ;i++)
        P[i]=0;

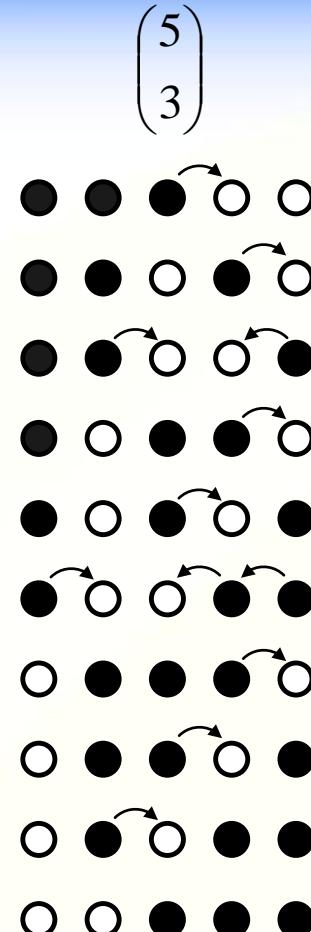
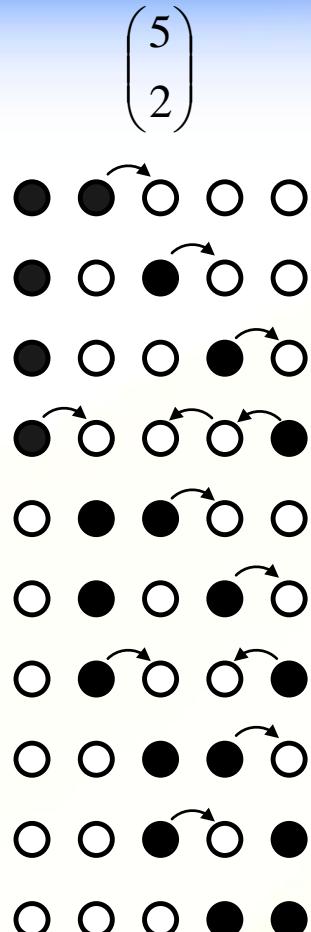
    do
    {
        for(int i=0; i<k; i++)
            printf("%5d ",P[i]+1);
        printf("\n");

        q = k-1;
        while (q >= 0)
        {
            P[q]++;
            if (P[q] == n)
            {
                P[q] = 0;
                q--;
            }
            else
                break;
        }
    } while (q >= 0);
    delete [] P;
}
```

# Комбинације без понављања

- Подскуп од  $k$  елемента  
бирамо из скупа од  $n$  елемената  
(редослед у подскупу од  $k$  елемената није битан)
- Број могућих избора је  $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ ,  $0 \leq k \leq n$ ,  $0!=1$
- Пун назив: комбинација  $k$ -те класе скупа  $X_n$   
(енг:  $k$ -combinations)
- Основа за имплементацију:
  - Поставити  $k$  елемената на прва слободна места са леве стране
  - Померити крајњи десни у десно за по једно место док може
  - Када више нема слободних места десно  
померити први слободни елемент са десне стране  
за једно место десно, а остале десно од њега ставити  
иза њега редом
  - Поновити процедуру док сви елементи не буду десно

# Илустрација и C/C++ имплементација



```
1 #include <stdio.h>
2
3 void combinations_without_repetition(int n, int k)
4 {
5     int i,j;
6     bool b;
7     int *P = new int[k];
8
9     for (i = 0; i < k; i++)
10        P[i] = i+1;
11
12    do
13    {
14        for (i = 0; i < k; i++)
15            printf("%3d ", P[i]);
16        printf("\n");
17
18        b = false;
19        for (i = k-1; i >= 0; i--)
20        {
21            if (P[i] < n - k + 1 + i)
22            {
23                P[i]++;
24                for (j = i + 1; j < k; j++)
25                    P[j] = P[j - 1] + 1;
26
27                b = true;
28                break;
29            }
30        }
31    } while (b);
32
33    if(P!=NULL) delete[] P;
34 }
```

# Пермутације без понављања (Систематско претраживање TSP)

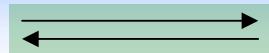
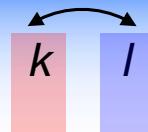
- На колико начина је могуће распоредити  $n$  елемената на  $n$  места (без понављања елемената)?
- Број различитих распореда је  $n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 2 \cdot 1$
- Пун назив: пермутације скупа  $X_n$  (енг: permutation of set)
- Алгоритам изабран по критеријуму **најбржег извршавања** и исписивања пермутација у **лексикографском поретку**

1	2	3	4	3	1	2	4
1	2	4	3	3	1	4	2
1	3	2	4	3	2	1	4
1	3	4	2	3	2	4	1
1	4	2	3	3	4	1	2
1	4	3	2	3	4	2	1
2	1	3	4	4	1	2	3
2	1	4	3	4	1	3	2
2	3	1	4	4	2	1	3
2	3	4	1	4	2	3	1
2	4	1	3	4	3	1	2
2	4	3	1	4	3	2	1

# Алгоритам: основна идеја

- Полазимо од пермутације  
 $P = \{P[0] \ P[1] \ \dots P[n - 1]\}$
- Пронађи највеће  $k$  ( $0 \leq k < n - 1$ )  
тако да је  $P[k] < P[k+1]$
- Пронађи највеће  $l$  ( $l > k$  и  $l \leq n - 1$ )  
тако да је  $P[l] > P[k]$
- Заменити вредности  $P[l]$  и  $P[k]$
- Обрнути редослед елемената  
 $P[k+1] \ P[k+2] \ \dots \ P[n - 1]$

# Алгоритам: илустрација



1	2	3	$k$	4	$k$	5		$\rightarrow$	1	2	3	5	4	$\rightarrow$	1	2	3	5	4
1	2	$k$	3	5	$k$	4		$\rightarrow$	1	2	4	5	3	$\rightarrow$	1	2	4	3	5
1	2	4	$k$	3	$k$	5		$\rightarrow$	1	2	4	5	3	$\rightarrow$	1	2	4	5	3
1	$k$	4	$k$	5	$k$	3		$\rightarrow$	1	2	5	4	3	$\rightarrow$	1	2	5	3	4
1	2	5	$k$	3	$k$	4		$\rightarrow$	1	2	5	4	3	$\rightarrow$	1	2	5	4	3
$k$			$k$		$k$			$\rightarrow$	1	3	5	4	2	$\rightarrow$	1	3	2	4	5
1	3	2	$k$	4	$k$	5		$\rightarrow$	1	3	2	5	4	$\rightarrow$	1	3	2	5	4

# Имплементација у C/C++

```
1 #include <stdio.h>
2 int next_permutation(const int N, int *P)
3 {
4     int s;
5     int* first=&P[0];
6     int* last=&P[N-1];
7     int* k=last-1;
8     int* l=last;
9     // find largest k so that P[k]<P[k+1]
10    while(k>first)
11    {
12        if(*k<*(k+1))
13        {
14            break;
15        }
16        k--;
17    }
18    // if no P[k]<P[k+1], P is the last permutation in lexicographic order
19    if(*k>*(k+1))
20    {
21        return 0;
22    }
23    // find largest l so that P[k]<P[l]
24    while(l>k)
25    {
26        if(*l>*k)
27        {
28            break;
29        }
30        l--;
31    }
32    // swap P[l] and P[k]
33    s= *k;
34    *k=*l;
35    *l=s;
36    // reverse the remaining P[k+1]...P[N-1]
37    first=k+1;
38    while(first<last)
39    {
40        s==*first;
41        *first==*last;
42        *last=s;
43
44        first++;
45        last--;
46    }
47
48    return 1;
49 }
```

```
51 void main(void)
52 {
53     int N=5;
54     int* P=new int [N];
55
56     //initialize the first permutation
57     for(int i=0; i<N; i++)
58         P[i]=i+1;
59
60     do
61     {
62         for(int i=0; i<N; i++)
63             printf("%2d ",P[i]);
64         printf("\n");
65     }while(next_permutation(N,P));
66
67     delete [] P;
68 }
```

# Колико је брз алгоритам?

- Колико највише пермутација може да се изврши за:
  - 1 min, а колико за
  - 60 min?
- Колико времена је потребно за  $N = 20$ ?  
 $(20! \approx 2,4 \cdot 10^{18})$
- Да ли постоји бржи алгоритам?

# Друге имплементације

- **C/C++**

```
#include <algorithm>
std::next_permutation(...)
```

- **Python**

```
1  from itertools import permutations
2  perm = permutations([1, 2, 3, 4])
3  for i in list(perm):
4      print(i)
```

- **MATLAB**

```
perms ([1 2 3 4])
```

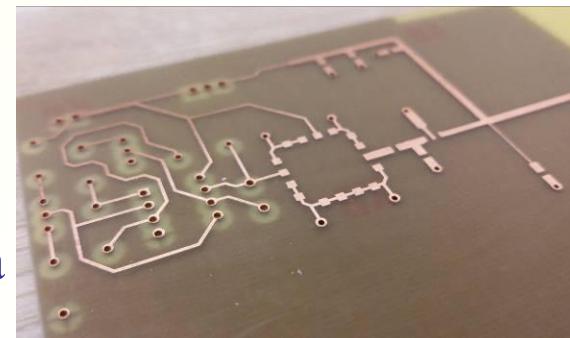
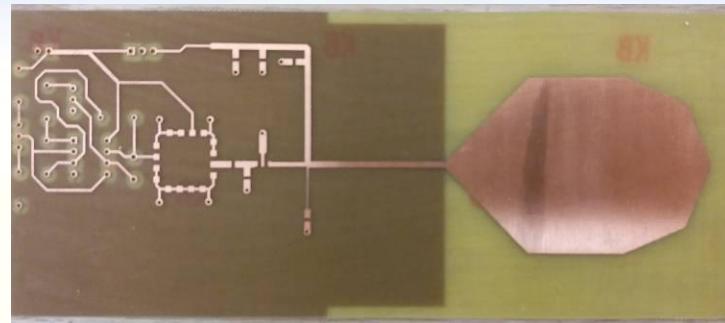
- **Wolfram Mathematica**

```
Permutations[{a b c}]
```

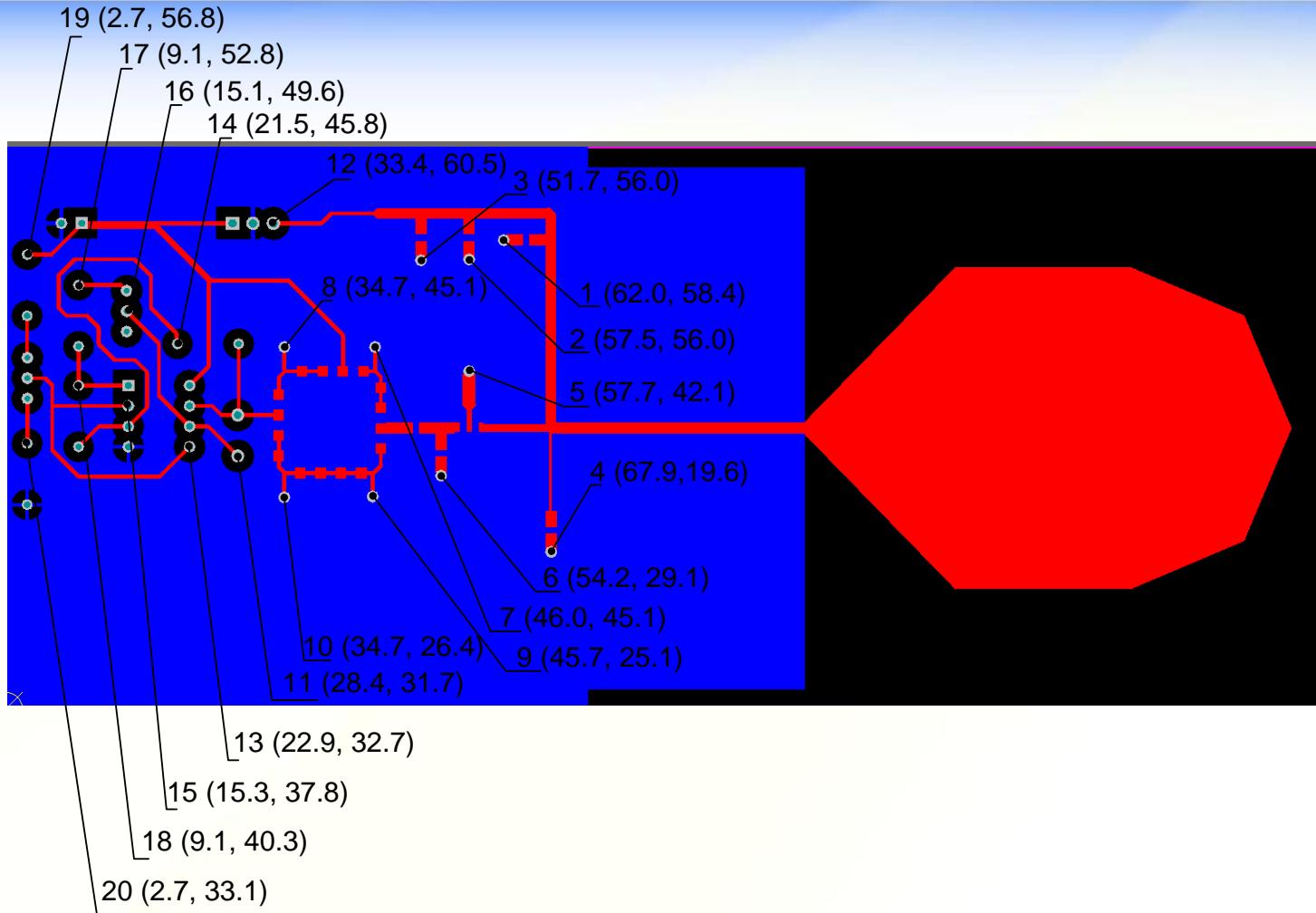
# Задатак за вежбе

## (Оптимално бушење плочице)

- Штампана плочица се израђује аутоматски
- На њој постоји више рупа истог пречника које је потребно избушити
- Аутоматска бургија има коначну брзину позиционирања
- Израђује се велики број идентичних плочица
- Уштеда времена израде једне плочице је од великог практичног интереса (множи се бројем плочица у серији)
- Потребно је минимизирати пут који пређе бургија (и минимизирати време израде)
- Бургија се не враћа на место полазне рупе када заврши један циклус бушења (у пракси враћа се истом путањом уназад)
- Координате рупа на слици су дате у милиметрима



# Координате тачка у [мм] на штампаниј плочици



# Задатак за вежбе (детаљи и израда)

- Пронаћи дужину најкраћег пута и записати редослед обилазака рупа за (а) првих 8 и (б) првих 12 рупа
  - Задатак се своди на TSP проблем са
    - 8 градова ( $8! = 40\ 320$  путања)
    - 12 градова ( $12! = 479\ 001\ 600$  путања)
  - Растојање између два града је  $L_2$  норма (Еуклидовско растојање)
  - Бургија може да крене из било које рупе (од првих 8 (а) или 12 (б))
  - У код за формирање свих пермутација додати
    - рачунање дужине путање
    - поређење са претходно нађеном најкраћом путањом и
    - памћење (испис) најкраће пронађене путање
- \* [5] БОНУС: написати ПАРАЛЕЛНУ имплементацију програма за потпуну претрагу TSP проблема и помоћу њега пронаћи најкраћи пут за првих 15 рупа ( $15! \approx 1,3 \times 10^{12}$ )

\*\* Lin-Kernighan heuristics [LKH] је једна од најбољих хеуристика за TSP

# Партиције природног броја

- Партиција (подела/разбијање) природног броја  $n$  је низ позитивних природних бројева  $a_1 \geq a_2 \geq \dots \geq a_m \geq 1$  тако да је  $n = a_1 + a_2 + \dots + a_m$
- При томе је:  $1 \leq m \leq n$  и  $n \geq 1$
- Лексикографски поредак
  - прва партиција је  $a_1 = n$
  - последња партиција је  $1 + 1 + \dots + 1 = n$

Све партиције  $n = 7$

7							
6	1						
5	2						
5	1	1					
4	3						
4	2	1					
4	1	1	1				
3	3	1					
3	2	2					
3	2	1	1				
3	1	1	1	1			
2	2	2	1				
2	2	1	1	1			
2	1	1	1	1	1		
1	1	1	1	1	1	1	1

# C/C++ код за генерирање свих партиција природног броја

- Партиција се чува у низу  $p[0], p[1], \dots, p[n-1]$
- Уколико је  $p[k] > 0$  тада је  $p[k]$  члан партиције
- Пронађи највеће  $k$  тако да је  $p[k] > 1$  и нађи број јединица  $q$
- $p[k] = p[k] - 1, q = q + 1$ 
  - Ако је  $p[k] \geq q$ :  $p[k+1] = q$
  - Ако је  $q > p[k]$ :  
 $p[k+1] = p[k]$   
 $q = q - p[k], k = k + 1$   
поновити корак докле год може

```
1 #include <stdio.h>
2 
3 bool next_partition(int* p, int n)
4 {
5     int k,i;
6     int q=0;
7 
8     if(p[0]==0) // the first one
9     {
10        p[0]=n;
11        return true;
12    }
13    else // finding k
14    {
15        for(i=0; i<n; i++)
16        {
17            if(p[i]!=0) k=i;
18            else break;
19        }
20    }
21    while (k>=0 && p[k]==1)
22    {
23        q++;
24        k--;
25    }
26 
27    if (k<0) return false;
28 
29    p[k]--;
30    q++;
31 
32    while(q > p[k])
33    {
34        p[k+1] = p[k];
35        q = q - p[k];
36        k++;
37    }
38 
39    p[k+1] = q;
40    k++;
41 
42    for(i=k+1; i<n; i++)
43        p[i]=0;
44 
45    return true;
46 }
47 
```

```
95 void driver_next_partition(void)
96 {
97     int n=7;
98     int* p=new int [n];
99 
100    for(int i=0; i<n; i++)
101        p[i]=0;
102 
103    while(next_partition(p, n))
104    {
105        //print current partition
106        for(int i=0; i<n && p[i]>0; i++)
107            printf("%2d ", p[i]);
108        printf("\n");
109    }
110    delete [] p;
111 }
```

# Илустрација алгоритма

Све партиције  $n = 7$

---

7	k=0, q=0+1: p[0]=7-1, p[1]=q=1
6 1	k=0, q=1+1: p[0]=6-1, p[1]=q=2
5 2	k=1, q=0+1: p[1]=2-1, p[2]=q=1
5 1 1	k=0, q=2+1: p[0]=5-1, p[1]=q=3
4 3	k=1, q=0+1: p[1]=3-1, p[2]=q=1
4 2 1	k=1, q=1+1: p[1]=2-1, p[2]=q=1
4 1 1 1	k=0, q=3+1: p[0]=4-1, p[1]=p[0], p[2]=q-p[1]
3 3 1	...
3 2 2	
3 2 1 1	
3 1 1 1 1	k=0, q=3+1: p[0]=3-1, p[1]=p[0], p[2]=q-p[1]
2 2 2 1	...
2 2 1 1 1	
2 1 1 1 1 1	
1 1 1 1 1 1 1	k=-1, q=7: нема наредне партиције

# Укупан број партиција: не постоји аналитички израз

The man who  
knew infinity

$$P(n) \approx \frac{1}{4n\sqrt{3}} e^{\pi\sqrt{\frac{2n}{3}}}$$

P( 1)=	1	P(21)=	792	P(41)=	44583	P(61)=	1121505
P( 2)=	2	P(22)=	1002	P(42)=	53174	P(62)=	1300156
P( 3)=	3	P(23)=	1255	P(43)=	63261	P(63)=	1505499
P( 4)=	5	P(24)=	1575	P(44)=	75175	P(64)=	1741630
P( 5)=	7	P(25)=	1958	P(45)=	89134	P(65)=	2012558
P( 6)=	11	P(26)=	2436	P(46)=	1055558	P(66)=	2323520
P( 7)=	15	P(27)=	3010	P(47)=	124754	P(67)=	2679689
P( 8)=	22	P(28)=	3718	P(48)=	147273	P(68)=	3087735
P( 9)=	30	P(29)=	4565	P(49)=	173525	P(69)=	3554345
P(10)=	42	P(30)=	5604	P(50)=	204226	P(70)=	4087968
P(11)=	56	P(31)=	6842	P(51)=	239943	P(71)=	4697205
P(12)=	77	P(32)=	8349	P(52)=	281589	P(72)=	5392783
P(13)=	101	P(33)=	10143	P(53)=	329931	P(73)=	6185689
P(14)=	135	P(34)=	12310	P(54)=	386155	P(74)=	7089500
P(15)=	176	P(35)=	14883	P(55)=	451276	P(75)=	8118264
P(16)=	231	P(36)=	17977	P(56)=	526823	P(76)=	9289091
P(17)=	297	P(37)=	21637	P(57)=	614154	P(77)=	10619863
P(18)=	385	P(38)=	26015	P(58)=	715220	P(78)=	12132164
P(19)=	490	P(39)=	31185	P(59)=	831820	P(79)=	13848650
P(20)=	627	P(40)=	37338	P(60)=	966467	P(80)=	15796476

# Партиције природног броја са тачно $m$ чланова

- $a_1 \geq a_2 \geq \dots \geq a_m \geq 1$  тако да је  $n = a_1 + a_2 + \dots + a_m$
- При томе је:  $2 \leq m \leq n$  задато
- Једноставна имплементација:  
прескочити партиције  $P(n)$  чији је број чланова различит од  $m$ 
  - Постоје и ефикасније имплементације

Све партиције

$n = 11 \ m = 4$

8	1	1	1
7	2	1	1
6	3	1	1
6	2	2	1
5	4	1	1
5	3	2	1
5	2	2	2
4	4	2	1
4	3	3	1
4	3	2	2
3	3	3	2

# Партиције скупа

- Партиција скупа  $S$  је подела  $S$  на подскупове  $S_m$ ,  $1 \leq m \leq |S|$  тако да важи:  
$$S = S_1 \cup S_2 \cup \dots \cup S_m$$
$$S_i \neq \{\emptyset\}, \quad 1 \leq i \leq m$$
$$i \neq j \Rightarrow S_i \cap S_j = \{\emptyset\}, \quad 1 \leq i, j \leq m$$
- $|S|$  је број елемената скупа  $S$

Све партиције

$S = \{1, 2, 3, 4\}$

-----  
{1,2,3,4}  
{1,2,3}, {4}  
{1,2,4}, {3}  
{1,2}, {3,4}  
{1,2}, {3}, {4}  
{1,3,4}, {2}  
{1,3}, {2,4}  
{1,3}, {2}, {4}  
{1,4}, {2,3}  
{1}, {2,3,4}  
{1}, {2,3}, {4}  
{1,4}, {2}, {3}  
{1}, {2,4}, {3}  
{1}, {2}, {3,4}  
{1}, {2}, {3}, {4}

# Пресликање партиције скупа

- Посматрајмо низ  $a_1, a_2, \dots, a_n$  који задовољава следеће услове:  
 $a_1 = 0$  и  
 $a_{j+1} \leq 1 + \max(a_1, \dots, a_j),$   
 $1 \leq j < n$
- Сваки следећи елемент низа може највише за 1 да буде већи од највећег елемента пре њега у низу  
(енг: restricted growth string)
- Једнозначно пресликање RGS на партицију скупа!
- Број партиција:  $\max(a_i) + 1$

Сви могући   Све партиције	низови $n=4$   $S = \{1, 2, 3, 4\}$
0 0 0 0	$\rightarrow \{1, 2, 3, 4\}$
0 0 0 1	$\rightarrow \{1, 2, 3\}, \{4\}$
0 0 1 0	$\rightarrow \{1, 2, 4\}, \{3\}$
0 0 1 1	$\rightarrow \{1, 2\}, \{3, 4\}$
0 0 1 2	$\rightarrow \{1, 2\}, \{3\}, \{4\}$
0 1 0 0	$\rightarrow \{1, 3, 4\}, \{2\}$
0 1 0 1	$\rightarrow \{1, 3\}, \{2, 4\}$
0 1 0 2	$\rightarrow \{1, 3\}, \{2\}, \{4\}$
0 1 1 0	$\rightarrow \{1, 4\}, \{2, 3\}$
0 1 1 1	$\rightarrow \{1\}, \{2, 3, 4\}$
0 1 1 2	$\rightarrow \{1\}, \{2, 3\}, \{4\}$
0 1 2 0	$\rightarrow \{1, 4\}, \{2\}, \{3\}$
0 1 2 1	$\rightarrow \{1\}, \{2, 4\}, \{3\}$
0 1 2 2	$\rightarrow \{1\}, \{2\}, \{3, 4\}$
0 1 2 3	$\rightarrow \{1\}, \{2\}, \{3\}, \{4\}$

# C/C++ код за генерирање свих партиција скупа

- RGS:  $a_i = 0, 0 \leq i \leq n - 1$
- Максималне вредности елемената  $b_0 = 0$ ,  
 $b_i = \max(b_{i-1}, a_{i-1} + 1)$   
 $1 \leq i \leq n - 1$
- Полазејќи од последњег елемента  $a_i$  проверити да ли  $a_i \leq b_i$ 
  - може:  $a_i = a_i + 1$  и одредити нове вредности за све  $b_i$
  - не може:  $a_i = a_i + 1$  и поновити корак за елемент  $i - 1$
- Крај је када нема елемента који се може повеќати за један

```
45 bool next_partition_of_set(int* a, int n)
46 {
47     int j;
48     int* b = new int [n];
49
50     for(j=0; j<n; j++)
51         b[j]=j==0 ? 0 : std::max(b[j-1],a[j-1]+1);
52
53     for(j=n-1; j>=0; j--)
54     {
55         if(a[j]<b[j])
56         {
57             a[j]++;
58             break;
59         }
60         else
61             a[j]=0;
62     }
63     if(j==-1)
64         return false;
65
66     delete [] b;
67     return true;
68 }

70 void driver_next_partition_of_set(void)
71 {
72     int n=4;
73     int* a = new int [n];
74     int i;

75     for(i=0; i<n; i++)
76         a[i]=0;

77     do
78     {
79         for(i=0; i<n; i++)
80             printf("%2d ",a[i]);
81         printf("\n");
82     }while(next_partition_of_set(a, n));

83
84     delete [] a;
85 }
```

# Партиција скупа на тачно $m$ подскупова

- Важи све што и за партицију скупа једино је  $m$  задато
  - Једноставна имплементација:  
узети само RGS које задовољавају  $\max(a_i) + 1 = m$
  - Постоје ефикасније имплементације које су алгоритамски сложеније
  - Бонус [5 поена]: осмислiti и имплементирati ефикасан алгоритам за проналажење свих партиција скупа од  $N$  елемената на тачно  $m$  подскупова ( $m \leq N$ ). Проверити резултате са следећег слајда тим кодом.
- Све партиције  
 $S = \{1, 2, 3, 4\}$   
 $m = 2$
- 
- $\{1, 2, 3\}, \{4\}$   
 $\{1, 2, 4\}, \{3\}$   
 $\{1, 2\}, \{3, 4\}$   
 $\{1, 3, 4\}, \{2\}$   
 $\{1, 3\}, \{2, 4\}$   
 $\{1, 4\}, \{2, 3\}$   
 $\{1\}, \{2, 3, 4\}$

# Број партиција скупа

- Тачно  $k$  подскупова
  - Стирлингови бројеви друге врсте  $S(n, k) = S_n^{(k)} = \binom{n}{k}$

$S(1, 1) = 1$   
 $S(2, 1) = 1 \quad S(2, 2) = 1$   
 $S(3, 1) = 1 \quad S(3, 2) = 3 \quad S(3, 3) = 1$   
 $S(4, 1) = 1 \quad S(4, 2) = 7 \quad S(4, 3) = 6 \quad S(4, 4) = 1$   
 $S(5, 1) = 1 \quad S(5, 2) = 15 \quad S(5, 3) = 25 \quad S(5, 4) = 10 \quad S(5, 5) = 1$   
 $S(6, 1) = 1 \quad S(6, 2) = 31 \quad S(6, 3) = 90 \quad S(6, 4) = 65 \quad S(6, 5) = 15 \quad S(6, 6) = 1$   
 $S(7, 1) = 1 \quad S(7, 2) = 63 \quad S(7, 3) = 301 \quad S(7, 4) = 350 \quad S(7, 5) = 140 \quad S(7, 6) = 21 \quad S(7, 7) = 1$   
 $S(8, 1) = 1 \quad S(8, 2) = 127 \quad S(8, 3) = 966 \quad S(8, 4) = 1701 \quad S(8, 5) = 1050 \quad S(8, 6) = 266 \quad S(8, 7) = 28 \quad S(8, 8) = 1$   
 $S(9, 1) = 1 \quad S(9, 2) = 255 \quad S(9, 3) = 3025 \quad S(9, 4) = 7770 \quad S(9, 5) = 6951 \quad S(9, 6) = 2646 \quad S(9, 7) = 462 \quad S(9, 8) = 36 \quad S(9, 9) = 1$   
 $S(10, 1) = 1 \quad S(10, 2) = 511 \quad S(10, 3) = 9330 \quad S(10, 4) = 34105 \quad S(10, 5) = 42525 \quad S(10, 6) = 22827 \quad S(10, 7) = 5880 \quad S(10, 8) = 750 \quad S(10, 9) = 45 \quad S(10, 10) = 1$

- Све могуће партиције
  - Белови бројеви

$$B_n = \sum_{k=1}^n S(n, k)$$

$B(1) =$	1	$B(8) =$	4140
$B(2) =$	2	$B(9) =$	21147
$B(3) =$	5	$B(10) =$	115975
$B(4) =$	15	$B(11) =$	678570
$B(5) =$	52	$B(12) =$	4213597
$B(6) =$	203	$B(13) =$	27644437
$B(7) =$	877	$B(14) =$	190899322

# Партиције код којих је редослед битан

- Композиција природног броја је партиција код које је редослед сабирача битан
- Пример  $n = 7$ :  $3+2+1+1$  или  $1+1+2+3$
- Број композиција броја  $n$  је  $2^{n-1}$   
(једнозначно пресликање  $\{0,1\}^{n-1}$ )  
$$(1 \overbrace{* 1 * 1 * \dots * 1}^n), * \in \{+, |\}$$
- Партиција скупа са редоследом подскупова:  
генерисати партицију скупа +  
пермутација свих подскупова

# Претраживање по стаблима графа: комплетан граф

- Комплетан неусмерен граф са  $n$  чворова ( $K_n$ ): између сваког паре чворова постоји грана

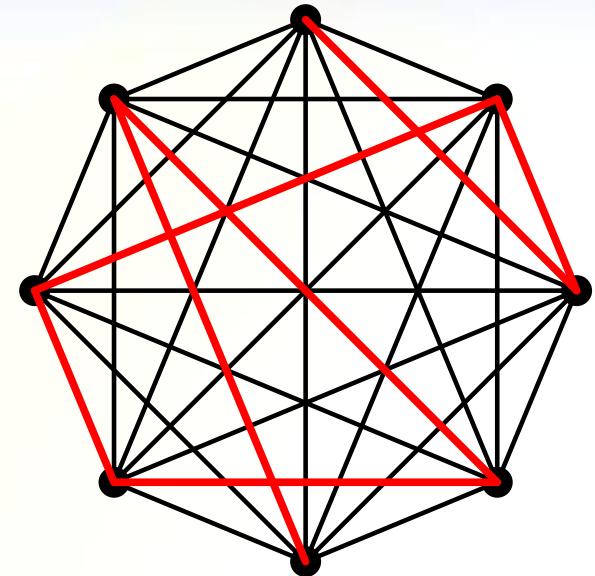
- Укупан број грана је

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

$K_1$	$K_2$	$K_3$	$K_4$
•	•	•	•
$K_5$	$K_6$	$K_7$	$K_8$

# Стабло графа

- Стабло графа (енг. spanning tree) је подскуп од  $n - 1$  грана које
  - повезују све чворове графа (постоји пут између произвољног пара чворова) и
  - нема петљи (до једног чвора може се доћи само на један начин)
- Колико има различитих стабала комплетног графа  $K_n$ ?

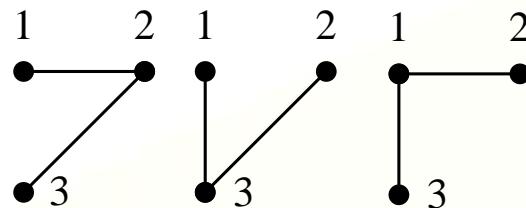


# Примери стабла комплетних графова $K_2$ , $K_3$ и $K_4$

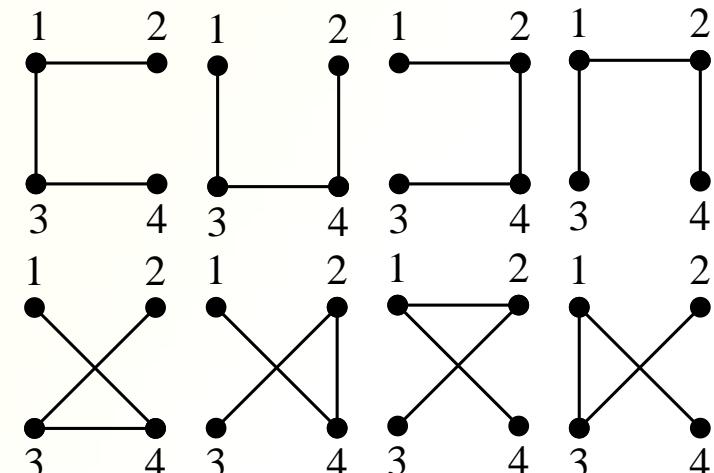
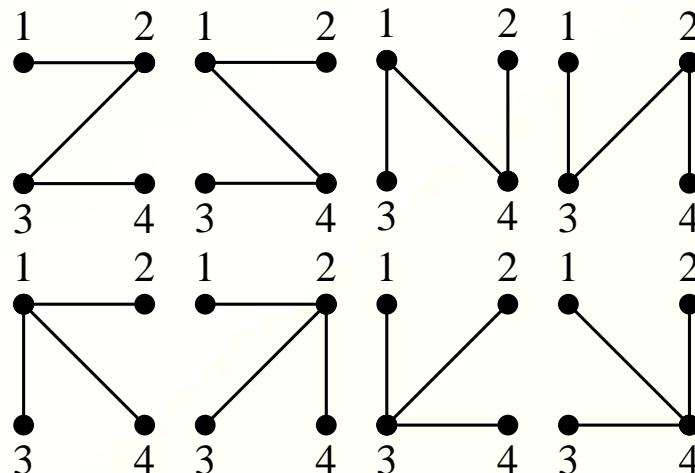
$K_2$ :



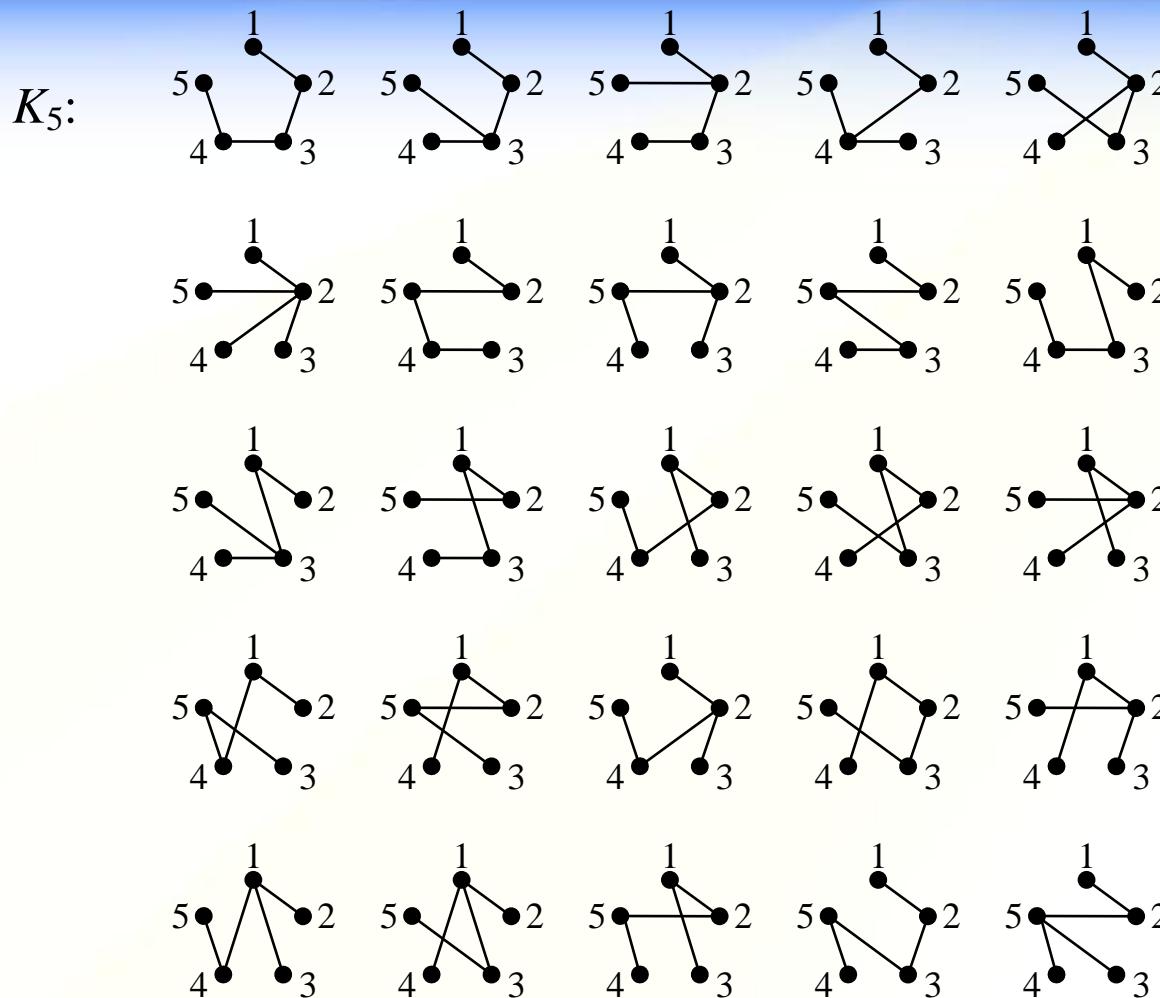
$K_3$ :



$K_4$ :



# Пример стабла $K_5$ : 1/5 стабала (цикличка пермутација индекса)



# Колики је број стабала графа?

- Број стабала комплетног графа је  $n^{n-2}$  (Cayley's formula)
- $n^{n-2} > n!$  ако је  $n \geq 5$
- Претраживање по свим стаблима је сложенији проблем од TSP!
- Уколико граф није комплетан број стабала је мањи од  $n^{n-2}$

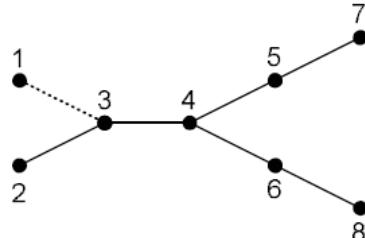
# Како генерисати сва стабла комплетног графа?

- Секвенце: варијације са понављањем  $n \geq 2$  елемената на  $n - 2$  места
  - $n = 4$  (број стабала  $K_4$  је 16):  
 $(1\ 1)\ (1\ 2)\ (1\ 3)\ (1\ 4)\ (2\ 1)\ (2\ 2)\ (2\ 3)\ (2\ 4)$   
 $(3\ 1)\ (3\ 2)\ (3\ 3)\ (3\ 4)\ (4\ 1)\ (4\ 2)\ (4\ 3)\ (4\ 4)$
- Постоји **једнозначно пресликање** између оваквих секвенци и стабала комплетног графа са  $n$  чворова (Prüfer)

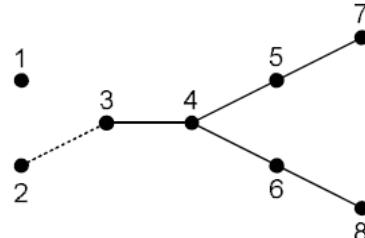
# Кодовање стабала у секвенце: алгоритам

- Лист = чвор из кога полази само једна грана
- Пronaћи лист стабла са  
најмањим редним бројем
- Записати редни број чвора са којим је  
повезан нађени лист (јединствени сусед)
- Избацити лист из стабла
- Поновити процедуру  $n - 2$  пута

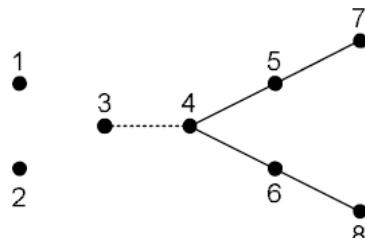
# Кодовања стабла у секвенце: пример



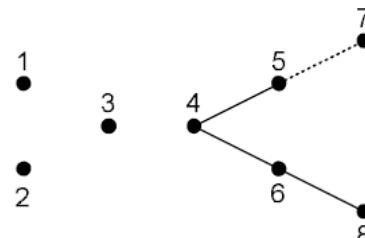
(a)  $P = (3)$



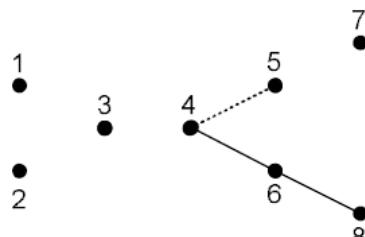
(b)  $P = (3, 3)$



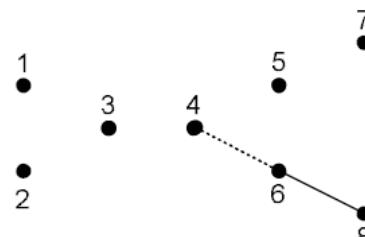
(c)  $P = (3, 3, 4)$



(d)  $P = (3, 3, 4, 5)$



(e)  $P = (3, 3, 4, 5, 4)$

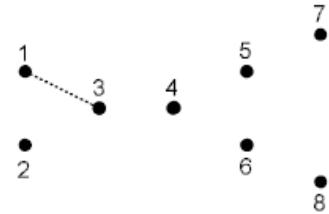


(f)  $P = (3, 3, 4, 5, 4, 6)$

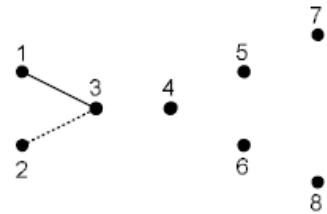
# Декодовање секвенце у стабло: алгоритам

- Задата је секвенца  $P = (p_1, p_2, \dots, p_{n-2})$
- Почети са низом чворова  
 $V = (v_1, v_2, \dots, v_n) = (1, 2, \dots, n)$
- Понављати за  $i = 1, 2, \dots, n - 2$ 
  - Пронаћи најмањи редни број чвора  $v_i$  који се не појављује у секвенци  $P$
  - Повезати  $v_i$  са  $p_i$
  - Избацити  $v_i$  из  $V$  и  $p_i$  из  $P$
- Последњи корак је повезивање преостала два елемента из  $V$

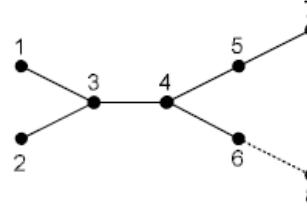
# Декодовање секвенце у стабло: пример



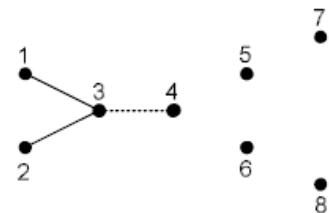
(a)  $P = (\underline{3}, 3, 4, 5, 4, 6)$ ;  $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$



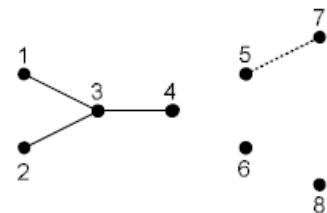
(b)  $P = (\underline{3}, 4, 5, 4, 6)$ ;  $V = \underline{2}, 3, 4, 5, 6, 7, 8$



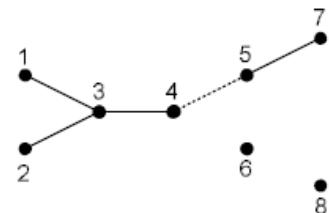
(g)  $P = ()$ ;  $V = \underline{6}, 8$



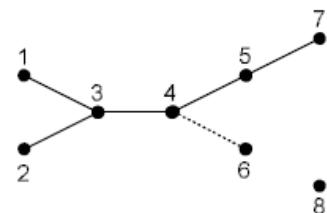
(c)  $P = (\underline{4}, 5, 4, 6)$ ;  $V = \underline{3}, 4, 5, 6, 7, 8$



(d)  $P = (\underline{5}, 4, 6)$ ;  $V = \{4, 5, 6, \underline{7}, 8\}$



(e)  $P = (\underline{4}, 6)$ ;  $V = \{4, \underline{5}, 6, 8\}$



(f)  $P = (\underline{6})$ ;  $V = \{\underline{4}, 6, 8\}$

# С/С++ код за декодовање

```
1 #include <stdio.h>
2
3 void SequenceToSpanningTree(int* P, int len, int* T)
4 {
5     int i,j,q=0;
6     int n=len+2;
7     int* V=new int [n];
8
9     for(i=0; i<n; i++)
10        V[i]=0;
11
12    for(i=0; i<len; i++)
13        V[P[i]-1] += 1;
14
15    for(i=0; i<len; i++)
16    {
17        for (j=0; j<n; j++)
18        {
19            if (V[j]==0)
20            {
21                V[j]=-1;
22                T[q++]=j+1;
23                T[q++]=P[i];
24                V[P[i]-1]--;
25                break;
26            }
27        }
28    }
29
30    j=0;
31    for(i=0; i<n; i++)
32    {
33        if(V[i]==0 && j==0)
34        {
35            T[q++]=i+1;
36            j++;
37        }
38        else if(V[i]==0 && j==1)
39        {
40            T[q++]=i+1;
41            break;
42        }
43    }
44
45    delete [] V;
46 }
47
```

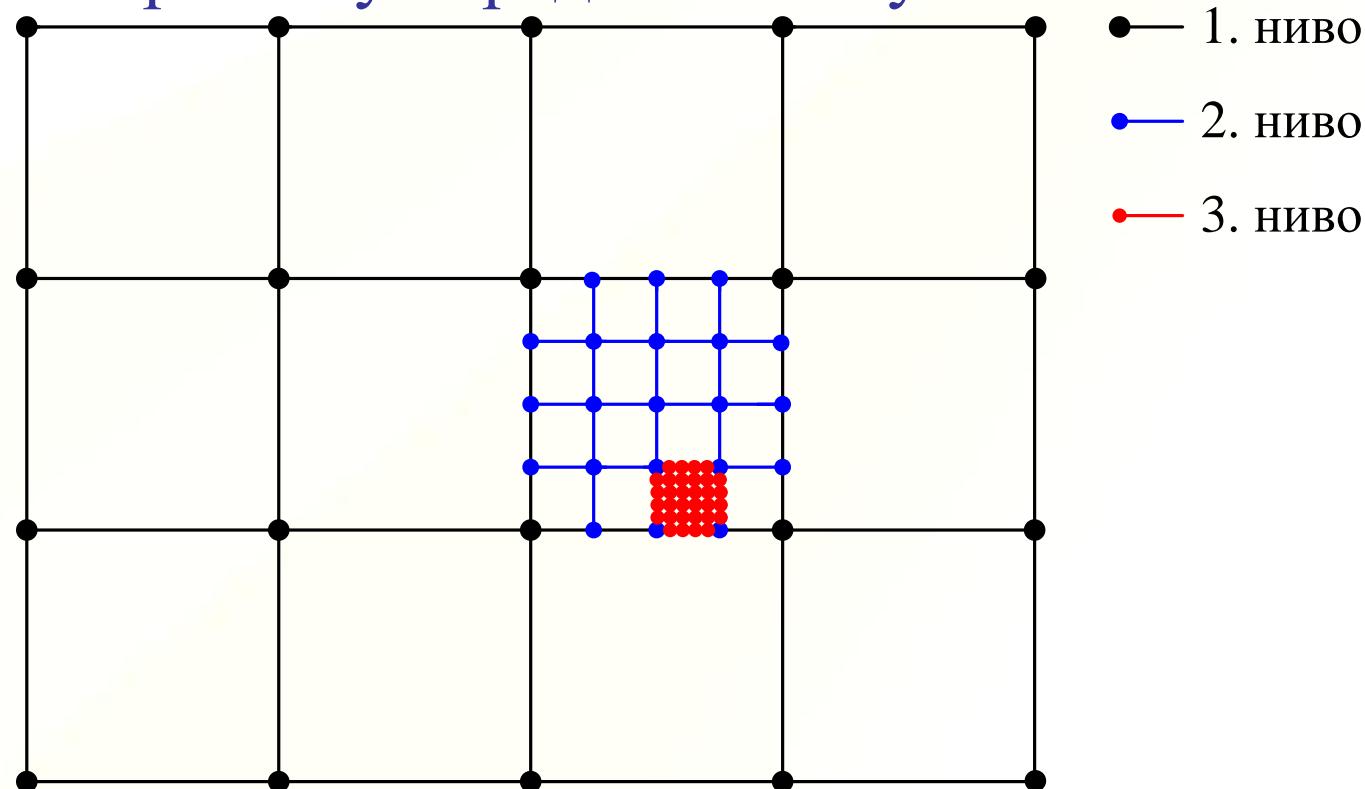
```
48 void main(void)
49 {
50     int P[] = { 1, 2, 2 };
51     int len = sizeof(P) / sizeof(P[0]);
52
53     int* T = new int [2*(len+1)];
54     SequenceToSpanningTree(P, len, T);
55
56     for(int i=0; i<2*(len+1); i++)
57     {
58         printf(" %d",T[i]);
59         if((i+1)%2==0 && i<2*len)
60             printf(" - ");
61     }
62     printf("\n");
63
64     delete [] T;
65
66 }
```

# Систематско претраживање: особине и употреба

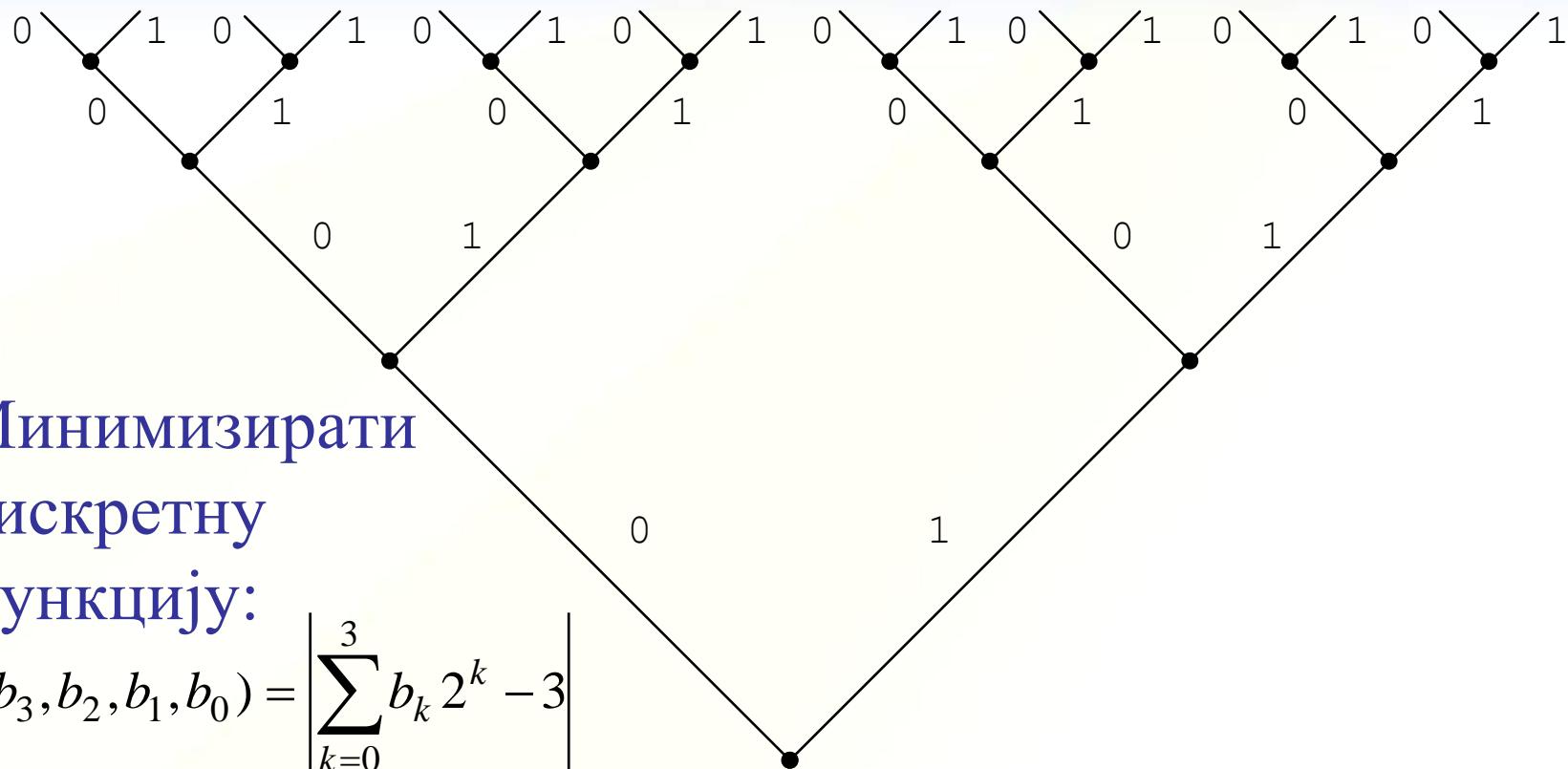
- Уколико желимо да докажемо да смо нашли **глобални оптимум** (најбоље могуће решење) једини начин је да **систематски претражимо** читав оптимизациони простор у општем случају
- Једно решење (једна тачка оптимизационог простора) се једном и само једном проверава
- Континуалне променљиве (NLP):
  - систематско претраживање подразумева коначан корак за сваку димензију (тачност)
  - решење смо нашли са тачношћу која је пропорционална кораку
- Дискретне променљиве (SAT и TSP):
  - сва решења су проверена

# Варијације: хијерархијско систематско претраживање

- Проценити део простора и претражити га са мањим кораком у наредном нивоу

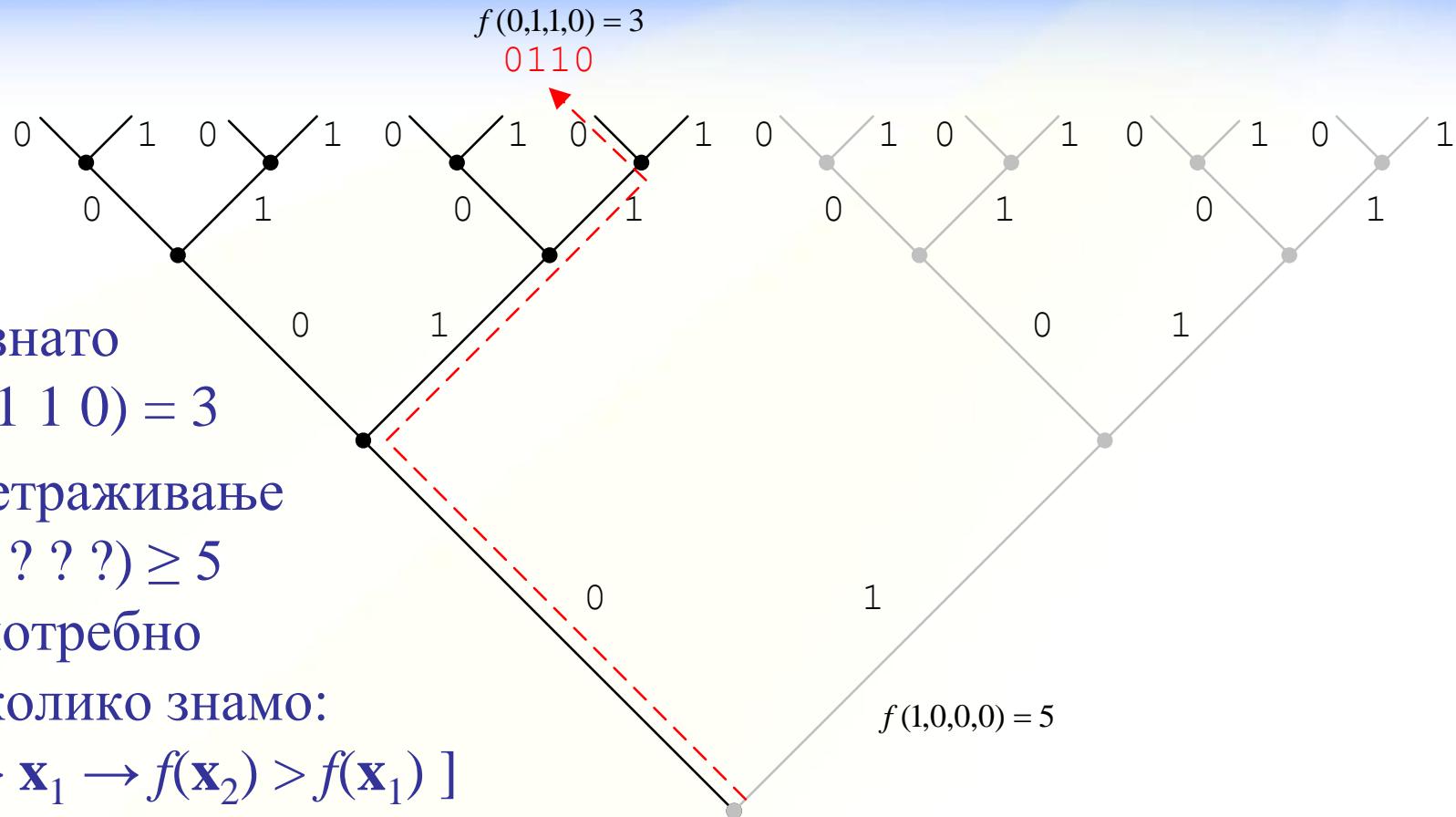


# Варијације: гранање и одсецање (branch & cut backtracking)



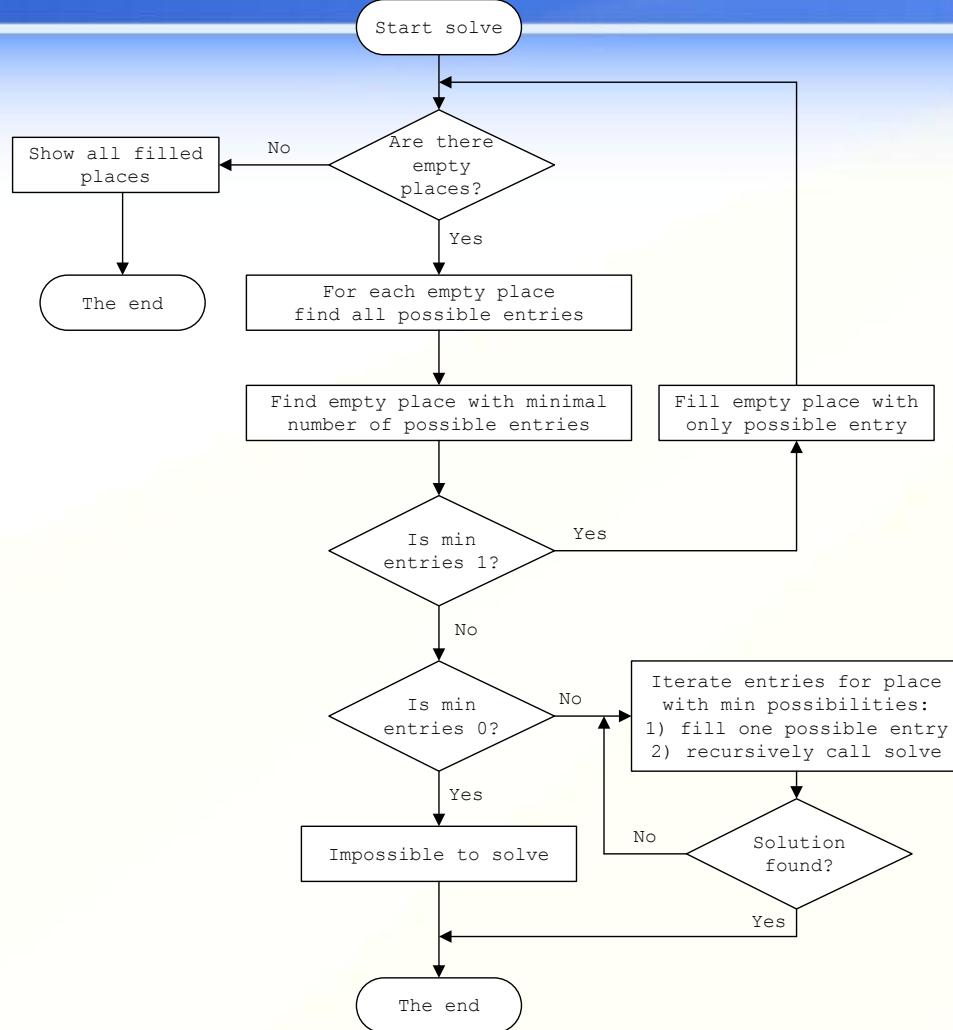
# Гранање и одсецање: прескочити део простора

- Познато  
 $f(0\ 1\ 1\ 0) = 3$
- Претраживање  
 $f(1\ ?\ ?\ ?) \geq 5$   
непотребно  
[ уколико знамо:  
 $\mathbf{x}_2 > \mathbf{x}_1 \rightarrow f(\mathbf{x}_2) > f(\mathbf{x}_1)$  ]
- Простор смањен 2 пута!



# SUDOKU:

## гранање и одсецање



1	4							
8	9							
2								
2	8	5						
1								
		7						
7		9	3					
6	1	4						
	2							
5	6	9	8	7	2	3	1	4
3	1	8	4	6	9	2	5	7
7	2	4	3	1	5	6	8	9
9	4	3	7	2	1	8	6	5
1	5	7	9	8	6	4	2	3
2	8	6	5	3	4	7	9	1
4	7	1	6	5	8	9	3	2
6	9	2	1	4	3	5	7	8
8	3	5	2	9	7	1	4	6

[ 33] Solving time: 0.061 [sec]

# Други називи и особине систематског претраживања

- Други називи за систематско претраживање
  - Grid search
  - Brute-force search
  - Parameter sweep
  - Exhaustive search
  - Enumeration
  - Generate & test
- Најбољи могући приступ  
уколико можемо да сачекамо да се претрага заврши
  - оптимизациони простор је мали у односу на расположиве ресурсе
- Уколико претрага траје недопустиво дugo није од користи
  - оптимизациони простор је велики у односу на рачунарске ресурсе
- Погодан за извршавање у  
паралели на рачунарима са више процесора (језгара)

# Случајно претраживање

- На случајан начин генеришу се тачке у којима се рачуна функција грешке
- Најчешће се користи генератор са униформном расподелом
- Неефикасан начин оптимизације јер су претходно и наредно израчунавање оптимизационе функције независни (иста тачка може да се испита више пута)
- Добар начин за грубу претрагу простора
- Сложеност  $O(N)$   
 $N$  број одбирача оптимизационог простора

# Генератори случајних бројева

- Рачунарско генерирање случајних бројева је нетривијалан задатак
- Генератори који постоје у библиотекама пролазе строге тестове случајности само делимично!
- Доступне функције:
  - C/C++: `rand()`, `srand()`, `boost/random...`
  - MATLAB: `rand()`, `randn()`, `randi()` ...
  - Python: `random.randint`, `random.uniform...`

# Једноставна рутина за генерирање целих случајних бројева

- Опсег  $a \leq \xi \leq b$
- Ограничение:  
`RAND_MAX`
- Ограничение зависи од компајлера  
(VS2017 `RAND_MAX = 32768`)
- Уколико је потребан већи опсег
  - генерисати произволјан низ бита {0, 1} и претворити га у цео број
  - генерисати два цела броја (или више), надовезати бите и интерпретирати нови низ као нови (већи) цео број

```
int random_int(int a, int b)
{
    int res;
    res = a + rand() % (b + 1 - a);
    return res;
}
```

# Једноставна рутина за генеришење реалних случајних бројева

- Опсег  $a \leq \xi \leq b$
- Разлика између два суседна броја који се генеришу (грануларност) је  $1/\text{RAND\_MAX}$
- Уколико је потребна већа грануларност:
  - Опсег  $[a \ b]$  поделити на више подопсега и сабрати (случајне) бројеве из подопсега

```
double random_float(double a, double b)
{
    double res;
    res = a + (double)rand() / (double)(RAND_MAX / (b-a));
    return res;
}
```

# C++11 <random>

```
#include <random>
#include <iostream>

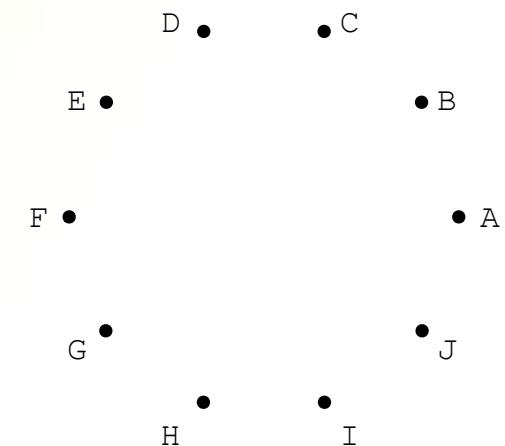
int main() {
    std::random_device rd;
    std::mt19937 mt(rd());

    std::uniform_real_distribution<double> random_float(1.0, 10.0);
    for (int i = 0; i<16; ++i)
        std::cout << random_float(mt) << "\n";

    std::uniform_int_distribution<int> random_int(1, 5);
    for (int i = 0; i<16; ++i)
        std::cout << random_int(mt) << "\n";
}
```

# Задатак

- Десет градова потребно је повезати мрежом за дистрибуцију електричне енергије
  - Сви градови морају бити повезани
  - При повезивању није дозвољено да постоје затворени путеви (петље)
  - Цена повезивања сваког пара градова дата је у табели (табела је симетрична)
  - Свако гранање од једног града ка **четири или више** градова повећава цену за  $(g - 3) \cdot 100$  јединица цене
  - Пронађи оптималан начин повезивања градова коришћењем **потпуне претраге** (минимизирати цену повезивања)



# Цена повезивања (табела је симетрична)

	A	B	C	D	E	F	G	H	I	J
A	0	374	200	223	108	178	252	285	240	356
B	374	0	255	166	433	199	135	95	136	17
C	200	255	0	128	277	821	180	160	131	247
D	223	166	128	0	430	47	52	84	40	155
E	108	433	277	430	0	453	478	344	389	423
F	178	199	821	47	453	0	91	110	64	181
G	252	135	180	52	478	91	0	114	83	117
H	285	95	160	84	344	110	114	0	47	78
I	240	136	131	40	389	64	83	47	0	118
J	356	17	247	155	423	181	117	78	118	0

# Поступак решавања

- Проблем се може решити претраживање по комплетном графу  $K_{10}$
- Потребно је претражити сва стабла овог графа
- Максималан број претрага је  $n^{n-2} = 10^8$
- Написати код за израчунавање оптимизационе функције:  $f(\mathbf{x}) = \sum_{k=1}^9 c_k + \sum_{n=1}^{10} \begin{cases} g(n) < 4, & 0 \\ g(n) \geq 4, & 100(g(n)-3) \end{cases}$   
 $c_k$  је “цена” гране са редним бројем  $k$   
(тј., вредност из табеле за полазни и крајњи чвор  $k$ -те гране)  
плус “пенал” за сваки чвор стабла у којем постоји гранање четири или више грана ( $g(n)$  је број грана које се стичу у чвиру  $n$ )
- Написати код који генерише све секвенце које се могу једнозначно пресликати у стабла графа (варијације са понављањем)
- Искористити рутину за пресликање секвенце у стабло графа
- Написати код који извршава потпуну претрагу
- Пронаћи и записати у ASCII фајл
  - минималну цену мреже за повезивање
  - путању за повезивање (стабло графа)свака грана графа је дефинисана са два чвора  $X$  и  $Y$   
решење треба да садржи девет грана (стабло) графа

# Случајно генерисана варијација са понављањем

- По један случајно изабран елемент из скупа од  $n$  елемената постављамо на свако од  $k$  места
  - Понављање елемената је дозвољено
- Имплементација: за свако од  $k$  места генеришемо један случајан цео број из скупа од  $n$  елемената
- Случајан вектор од  $n$  реалних бројева из опсега  $[x_{\min}, x_{\max}]$ 
  - На формално идентичан начин генерише се случајно изабран вектор реалних бројева
  - Једина разлика је што се користи генератор случајних реалних бројева из датог опсега

```
38 void driver_random_variation(void)
39 {
40     srand((unsigned int)time(NULL));
41
42     int n=10;
43     int k=4;
44     int s=15;
45
46     int* r = new int [k];
47
48     for(int i=0; i<s; i++)
49     {
50         for(int j=0; j<k; j++)
51         {
52             r[j]=random_int(0,n-1);
53             printf("%2d ",r[j]);
54         }
55         printf("\n");
56     }
57
58     delete [] r;
59 }
60
61 void driver_random_vector(void)
62 {
63     srand((unsigned int)time(NULL));
64
65     double xmin=-1.0;
66     double xmax=+1.0;
67     int k=4;
68     int s=15;
69
70     double* r = new double [k];
71
72     for(int i=0; i<s; i++)
73     {
74         for(int j=0; j<k; j++)
75         {
76             r[j]=random_float(xmin,xmax);
77             printf("% .2f ",r[j]);
78         }
79         printf("\n");
80     }
81
82     delete [] r;
83 }
```

# Случајно генерисана комбинација без понављања

- Из скупа од  $n$  елемената случајно изабрати  $k$  различитих елемената
  - Понављање није дозвољено
- За свако  $i = 0, 1, \dots, k - 1$  случајно генерисати цео број  $0 \leq r \leq n - 1 - i$
- Изабрати елемент на месту  $r$  који већ није изабран
- Записати изабране елементе

```
136 void random_combination(int n, int k, int* P)
137 {
138     if(k>n) return;
139
140     int i,j,r,c;
141     int* Q = new int [n];
142
143     for(i=0; i<n; i++)
144         Q[i]=0;
145
146     for(i=0; i<k; i++)
147     {
148         r = random_int(0,n-1-i);
149         c=0;
150         for(j=0; j<n; j++)
151         {
152             if(Q[j]==0)
153             {
154                 if(r==c)
155                 {
156                     Q[j]++;
157                     break;
158                 }
159                 c++;
160             }
161         }
162     }
163
164     c=0;
165     for(i=0; i<n; i++)
166     {
167         if(Q[i]==1)
168         {
169             P[c]=i+1;
170             c++;
171         }
172     }
173     delete [] Q;
174 }
```

```
176 void driver_random_combination(void)
177 {
178     int n=5;
179     int k=3;
180
181     int* P = new int [k];
182
183     for(int i=0; i<100; i++)
184     {
185         random_combination(n,k,P);
186
187         for(int i=0; i<k; i++)
188             printf("%2d ",P[i]);
189         printf("\n");
190
191     }
192     delete [] P;
193 }
```

# Илустрација алгоритма

$Q = (0, 0, 0, 0, 0, 0, 0, 0)$

`random_int(0, 6) = 5`       $Q = (0, 0, 0, 0, 0, \textcolor{red}{1}, 0)$

`random_int(0, 5) = 5`       $Q = (0, 0, 0, 0, 0, \textcolor{red}{1}, \textcolor{red}{1})$

`random_int(0, 4) = 0`       $Q = (\textcolor{red}{1}, 0, 0, 0, 0, \textcolor{red}{1}, \textcolor{red}{1})$

`random_int(0, 3) = 3`       $Q = (\textcolor{red}{1}, 0, 0, 0, \textcolor{red}{1}, \textcolor{red}{1}, \textcolor{red}{1})$

$P = (1, 5, 6, 7)$

- Бројеви могу да се директно записују у  $P$ 
  - сортира се  $P$  на крају (ефикасније али захтева сортирање)

# Случајно генерисана пермутација

- Полазећи од низа  $a_0, a_1, \dots, a_{n-1}$  генерисати случајну пермутацију
- За свако  $i = n - 1, n - 2, \dots, 1$
- Случајно генерисати цео број  $j$ ,  $0 \leq j \leq i$
- Заменити вредности  $a_j \leftrightarrow a_i$
- $n$  замена по паровима не даје добру статистику!

```
88 void random_permutation(int n, int *p)
89 {
90     int i,j,s;
91     for(i=n-1; i>0; i--)
92     {
93         j = random_int(0, i);
94         if(i!=j)
95         {
96             s = p[i];
97             p[i]=p[j];
98             p[j]=s;
99         }
100    }
101 }
102
103 void driver_random_permutation(void)
104 {
105     int n=3;
106     int* p=new int [n];
107     int* s=new int [n];
108     int T=1000;
109
110     // initialization
111     for(int i=0; i<n; i++)
112     {
113         p[i]=i+1;
114         s[i]=0;
115     }
116
117     for(int i=0; i<T; i++)
118     {
119         random_permutation(n, p);
120
121         // print-out
122         for(int j=0; j<n; j++)
123             printf("%2d ",p[j]);
124             printf("\n");
125
126         s[p[0]-1]++;
127     }
128
129     for(int i=0; i<n; i++)
130         printf("%2.5f\n",s[i]*1.0/T);
131
132     delete [] p;
133     delete [] s;
```

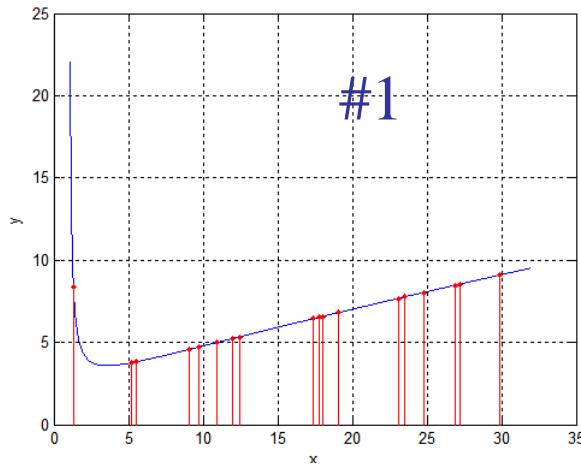
# Илустрација алгоритма

		Основна идеја						Имплементација на задатом низу	
		Низ неискоришћених			Нови низ				
		1	2	3	4	5	6	7	
random_int(0, 6)=6		1	2	3	4	5	6	7	1 2 3 4 5 6 7
random_int(0, 5)=2		1	2	3	4	5	6	7	1 2 6 4 5 3 7
random_int(0, 4)=4		1	2	4	5	6		6 3 7	1 2 6 4 5 3 7
random_int(0, 3)=1		1	2	4	5			2 6 3 7	1 4 6 2 5 3 7
random_int(0, 2)=0		1	4	5				1 2 6 3 7	6 4 1 2 5 3 7
random_int(0, 1)=1		4	5					5 1 2 6 3 7	6 4 1 2 5 3 7
random_int(0, 0)=0		4						4 5 1 2 6 3 7	6 4 1 2 5 3 7

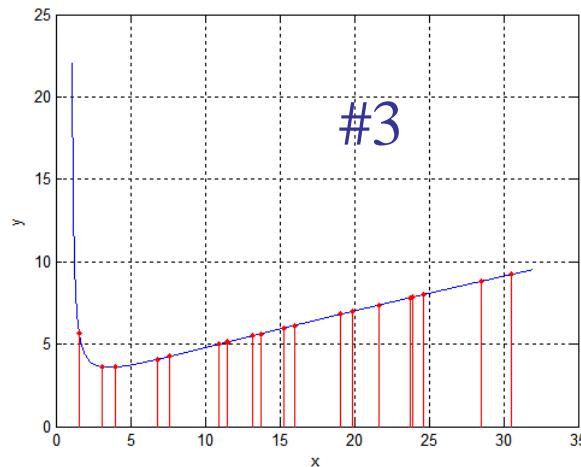
# Случајно генерисане партиције и стабла графа

- Случајно генерисана партиција броја  $n$ :
  - Пронаћи укупан број партиција броја, генерисати случајан број од 1 до  $n$  и генерисати одговарајућу партицију алгоритмом за генерисање свих партиција
  - Једноставан али рачунарски захтеван приступ
- Случајно генерисана партиција скупа  $S$ 
  - Генерисати случајан RGS (restricted growth string), сваки елемент је између нуле и највећи пре њега +1
- Случајно генерисано стабло потпуног графа  $K_n$ 
  - Генерисати низ од  $n - 2$  елемента на чијим местима може да буде  $1, 2, \dots, n$  (видети пресликовање таквих низова у стабла графа)
  - Ако граф није потпун, недозвољена стабла се изостављају
    - Постоје рачунарски ефикасније методе

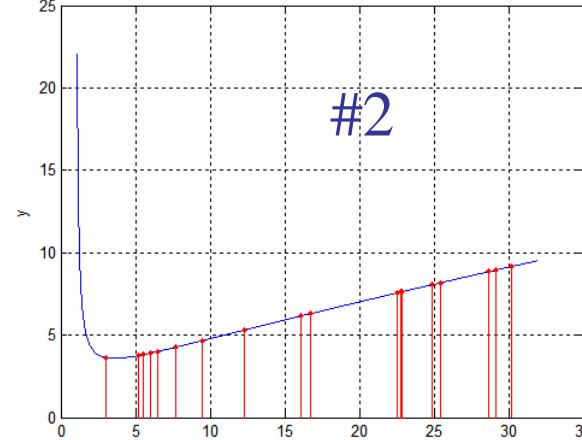
# Пример: различита решења при сваком покретању



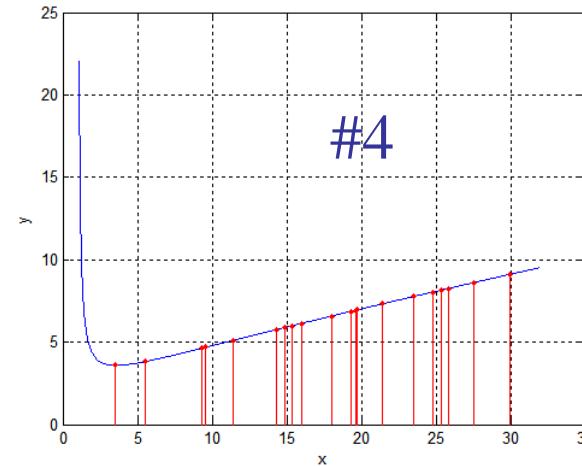
#1



#3

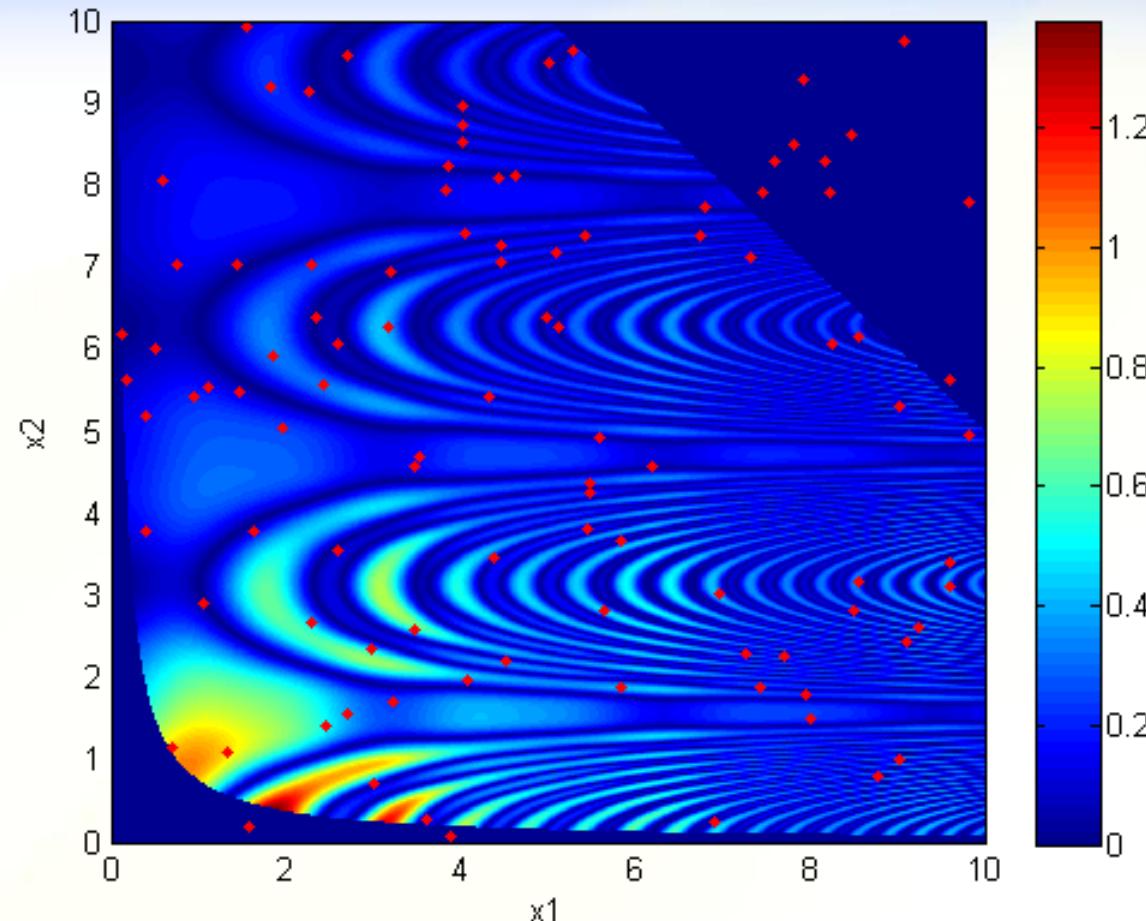


#2



#4

# Пример случајног претраживања једне 2D функције (NLP)



# О случајним процесима...

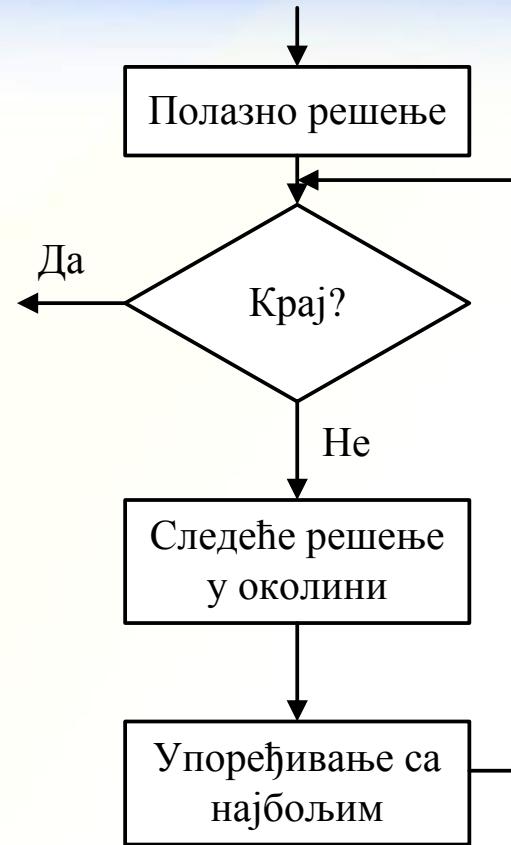
## (by Piet Hein)

Whenever you're **called on to make up your mind**  
and you're hampered by **not having any**  
**the best way to solve** the dilemma you'll find  
is simply by **spinning a penny**.

No - **not so that chance shall decide** the affair  
while you're passively standing there moping;  
**but the moment the penny is up** in the air  
you suddenly **know what you're hoping**.

# Локални оптимизациони алгоритми

- Енг: hill-climbing, down-hill, greedy, local optimization...
- Претражују околину полазног решења
- Заснивају се на итеративном поправљању полазног решења
- Генерализовани блок дијаграм је приказан десно



# Слабости Hill-Climbing Алгоритама

- По правилу проналазе само локални оптимум
- Нема информације о томе колико смо далеко или близу глобалног оптимума
- Крајње решење зависи од полазног
- Генерално, није могуће предвидети максималан број потребних итерација (зависи од оптимизационе функције)

# SAT локално претраживање

- Дефинисати околину која се претражује
- Хамингово растојање  
(број бита,  $k$ , колико сме да се промени)
- Ако је број димензија  $D$  генерисати  
све могуће промене
  - број избора  $k$  бита од  $D$
  - пута број распореда бита на  $k$  места
- Памтити решења и не проверавати  
више пута исто решење

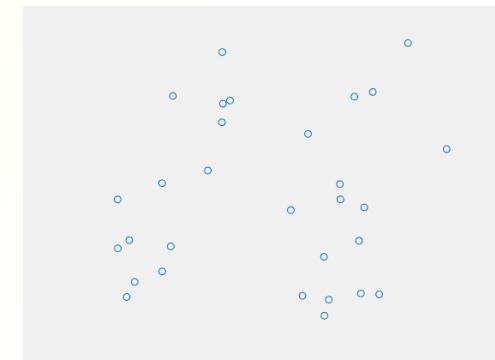
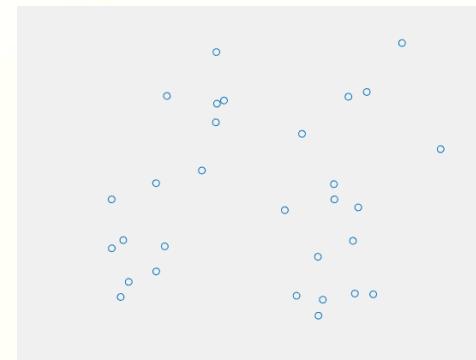
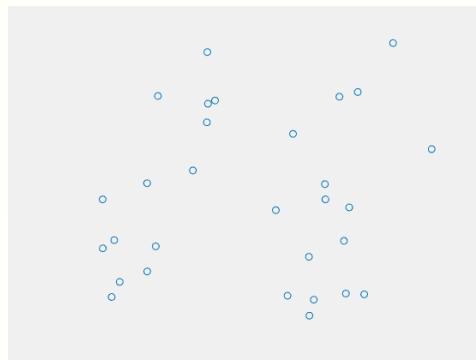
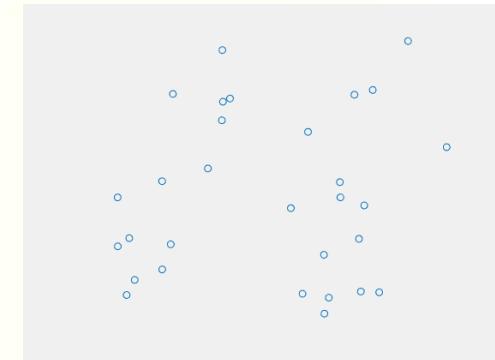
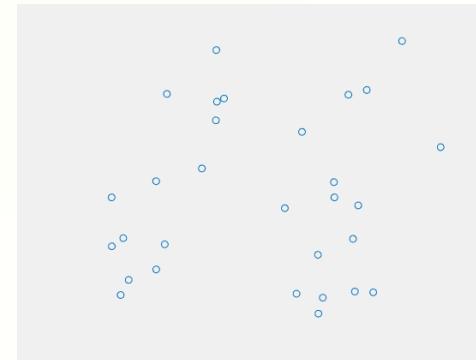
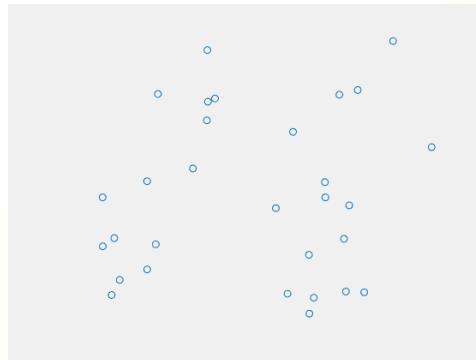
Претрага окolini $k=3$ бита	Све могуће вредности за промену $k=3$ бита
1 0 1 1 0	0 0 0
1 0 1 1 0	0 0 1
1 0 1 1 0	0 1 0
1 0 1 1 0	0 1 1
1 0 1 1 0	1 0 0
1 0 1 1 0	1 0 1
1 0 1 1 0	1 1 0
1 0 1 1 0	1 1 1
1 0 1 1 0	
1 0 1 1 0	

# Графови: локално претраживање (minimum spanning tree)

- Prim's algorithm (Jarnik's algorithm, Prim-Dijkstra algorithm, DJP algorithm)
  - локално претраживање по графовима
  - проналажење MST
- Кораци алгоритма:
  - изабрати један чврт графа (произвољно) и уврстити га у стабло
  - пронаћи грану са најмањом тежином између било ког чвора стабла и чворова који нису у стаблу и уврстити одговарајући чврт и грану у стабло
  - поновити претходни корак док сви чворови нису у стаблу
- Ово је истовремено и алгоритам за проналажење MST
- Променом тежина грана графа могуће је генерисати различита стабла графа овим алгоритмом (ако се тежине грана додеље на случајан начин, добија се случајно генерисано стабло графа!)

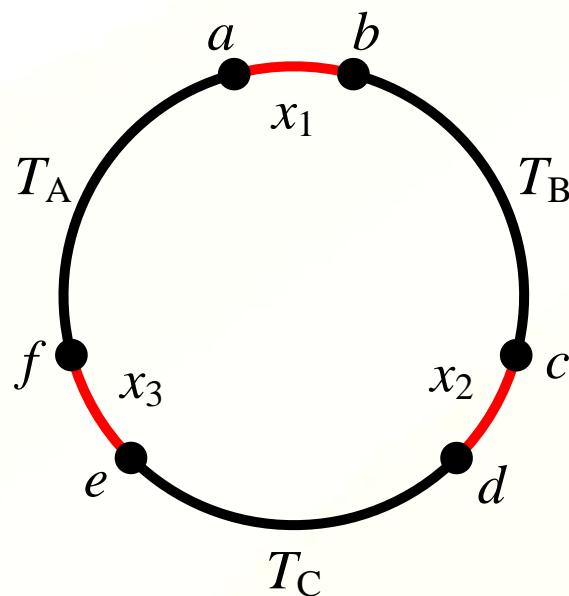
# Илустрација проналажења стабла са минималним збиром тежина

- Полазак из различитог чвора даје исти резултат (исто стабло)
- У коришћеном примеру тежина гране је растојање између чворова
- Временска сложеност зависи од структуре података за чување графа



# TSP Локално претраживање З-опт: основна идеја

- Проблем TSP класе, полазно решење (пермутација)  $T_0$
- Изаберемо три везе  $(x_1, x_2, x_3)$  које прекинемо  $(a,b), (c,d), (e,f)$
- Три преостала подниза  $T_A, T_B, T_C$  повезујемо на све могуће начине



$$T_0 = T_A + x_1 + T_B + x_2 + T_C + x_3$$

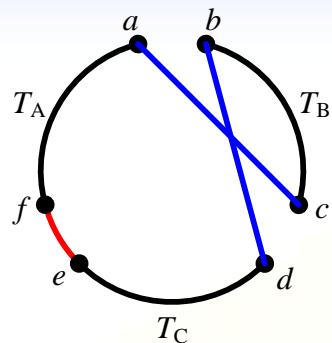
$$L(T) = f(T)$$

$$T_q = (n_1, n_2, \dots, n_k) \rightarrow R_q = (n_k, n_{k-1}, \dots, n_1)$$
$$q \in \{A, B, C\}$$

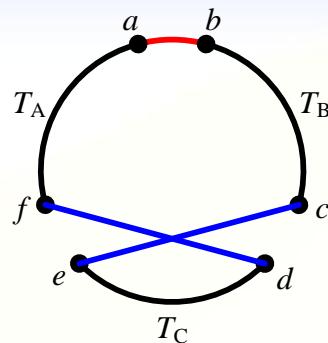
$$f(T) < f(T_0)$$

# 3-ОПТ: СВИ МОГУЋИ начини повезивања

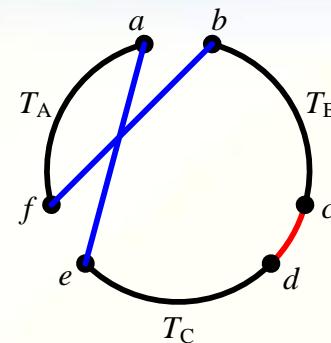
#1:  $T_A R_B T_C$



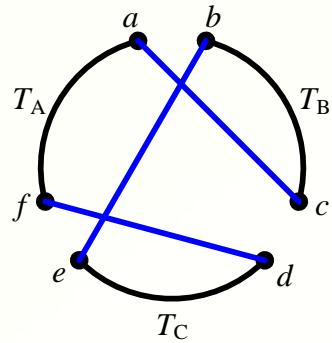
#2:  $T_A T_B R_C$



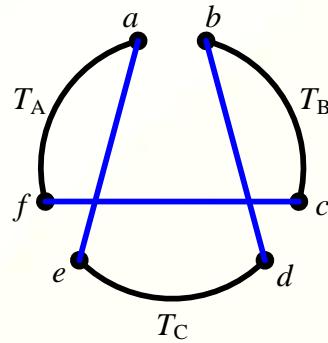
#3:  $R_A T_B T_C$



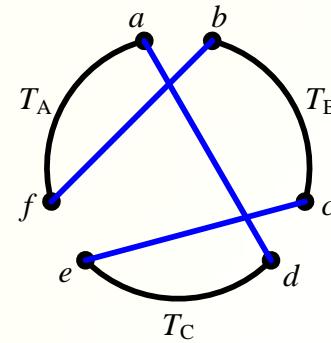
#4:  $T_A R_B R_C$



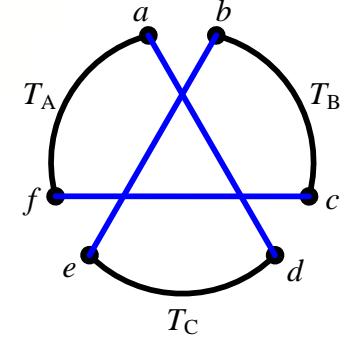
#5:  $T_A R_C R_B$



#6:  $T_A T_C R_B$

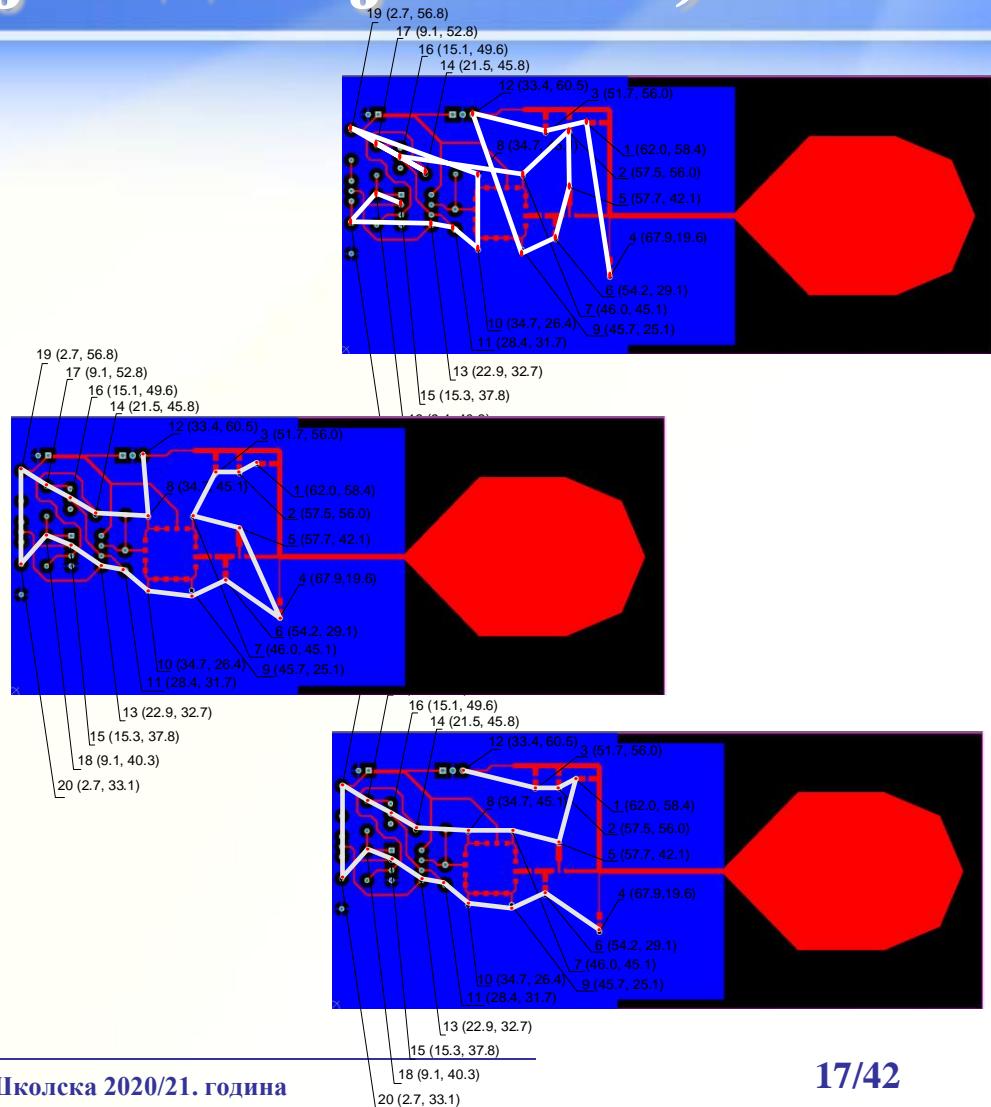


#7:  $T_A T_C T_B$



# 3-опт: пример бушења 20 рупа (2,5 % лошије од најбољег)

- 1 000 000 случајних покушаја  
 $f(\mathbf{x}) = 335,91 \text{ mm}$
- 10 000 случајних и 3-опт за најбољу  
 $f(\mathbf{x}) = 210,58 \text{ mm}$ 
  - Свако покретање даје другачије решење!
- Најбоље решење  
 $f(\mathbf{x}) = 205,136 \text{ mm}$



# 3-опт: примена

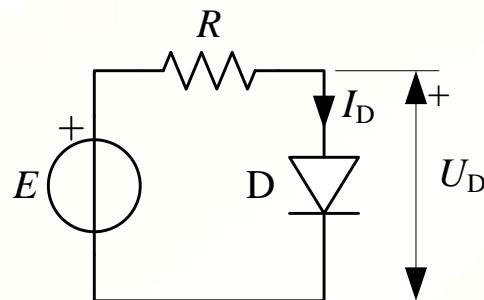
- 3-опт је специјални случај  $k$ -опт
  - $k = D$  (број градова) је потпуна претрага
- 3-опт укључује 2-опт (важи генерално)
- Које три везе пресећи?
  - највише  $\binom{D}{3} \sim O(D^3)$  ако је  $D$  велико постаје непрактично
  - издвојити неке, али које?
- Генерализација  
Lin-Kerningan-Helsgaun хеуристика

# NLP: континуалне оптимизационе функције

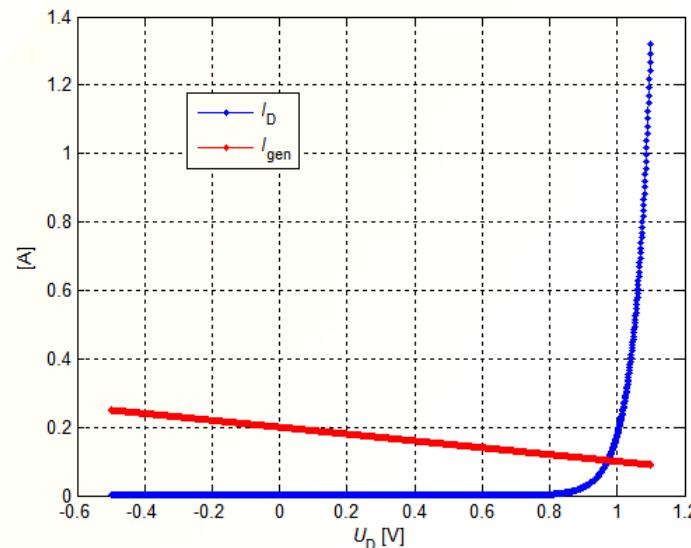
- Проблем:

Карактеристика диоде дата је изразом  $I_D = I_0(e^{U_D/U_0} - 1)$ , где је  $U_0 = 50 \text{ mV}$  и  $I_0 = 1 \text{ nA}$ . Електромоторна сила генератора је  $E = 2 \text{ V}$ , а отпорност  $R = 10 \Omega$ .

- Израчунати радну тачку диоде



$$\frac{E - U_D}{R} = I_0 \left( e^{\frac{U_D}{U_0}} - 1 \right)$$



# Оптимизација и решавање трансцендентних једначина

- Континуална функција са једном променљивом

- Трансцендентна једначина  $\frac{E - U_D}{R} - I_0 \left( e^{\frac{U_D}{U_0}} - 1 \right) = 0$

- Знамо да постоји тачно једно решење

- Оптимизациони проблем,  $L_1$  - норма

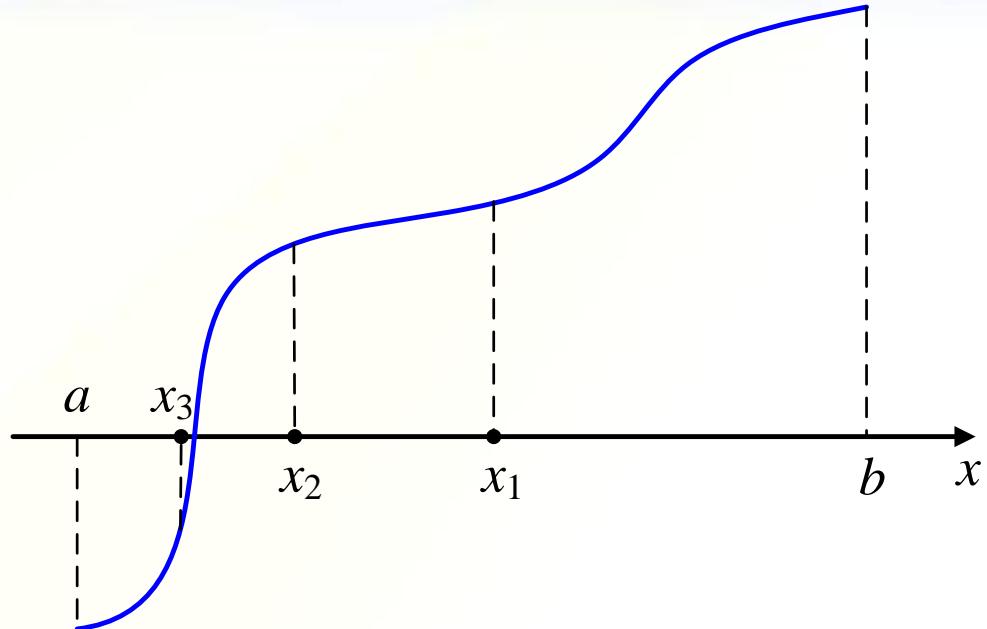
$$f_{\text{opt}}(U_D) = \left| \frac{E - U_D}{R} - I_0 \left( e^{\frac{U_D}{U_0}} - 1 \right) \right| \quad \min(f_{\text{opt}}(U_D)), U_D \in \Re$$

# Класични методи (NLP: континуалне функције)

- Метод половљења интервала (енглески: bisection/bracketing method)
- Метод сечице (regula-falsi)
- Њутнов метод (апроксимација тангентама)
- Брентов метод (апроксимација параболама)
- Лагранжови мултипликатори
- ККТ услови
- Градијентни метод
- Хесијан
  - BFGS (Broyden–Fletcher–Goldfarb–Shanno) метод

# Половљење интервала

- Услов:  
постоји интервал  
 $f(a)f(b) < 0$
- $x_1 = 0,5 (a + b)$
- Нови интервал:  
 $[a, x_1]$  ако  $f(a)f(x_1) < 0$   
 $[x_1, b]$  ако  $f(x_1)f(b) < 0$
- Поновити процедуру  
док се нула не одреди са задатом тачношћу



# Половљење интервала брзо конвергира

- У кораку  $n$  интервал је  $\frac{b-a}{2^n}$
- Свака итерација смањује интервал 2 пута
- Изузетно робустан
- Једноставан за програмирање
- Не захтева познавање извода функције!
- Може да се примени и ако је функција са шумом или са целим бројевима
- Захтева познавање почетног интервала
- Генерализација на вишедимензионе проблеме?

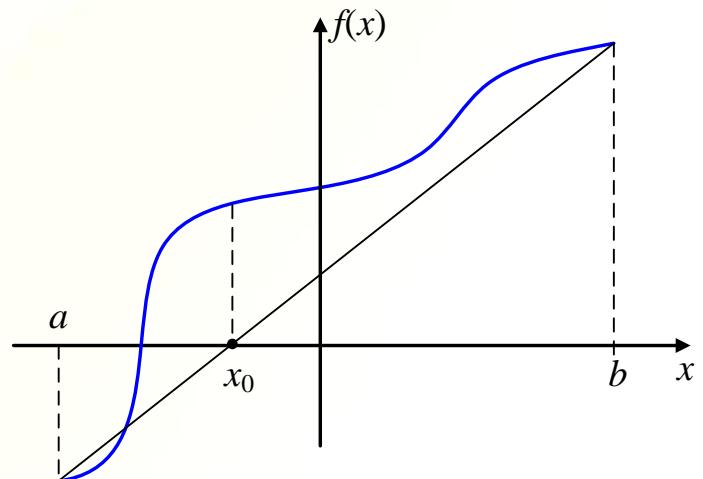
# Regula falsi: основна идеја и скица извођења

- Нула се тражи у пресеку линеарне апроксимације функције и апсцисе

$$f(x) = \frac{f(b) - f(a)}{b - a}(x - a) + f(a)$$

$$0 = \frac{f(b) - f(a)}{b - a}(x_0 - a) + f(a)$$

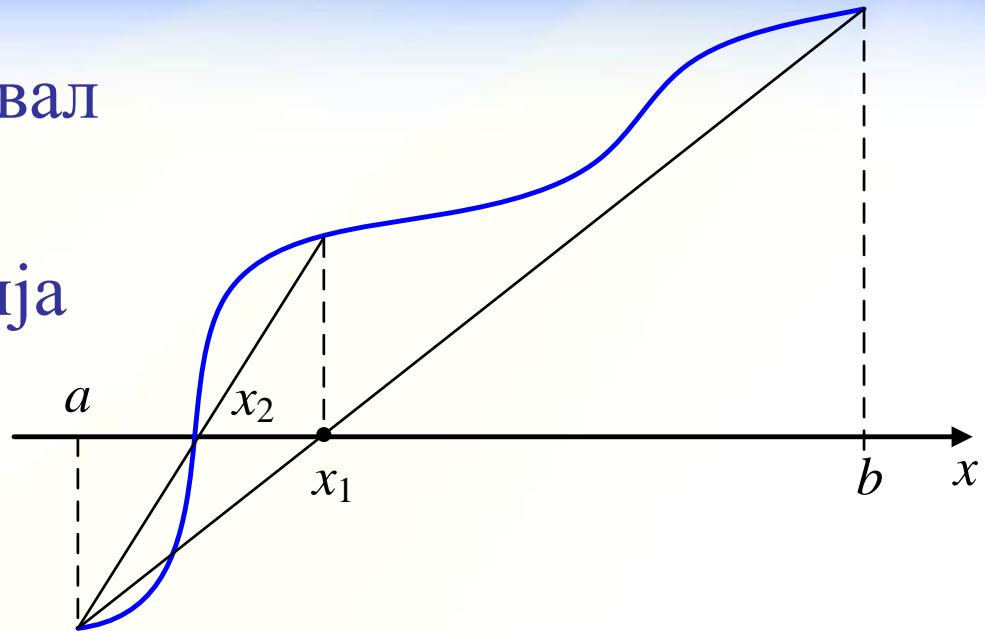
$$x_0 = \frac{af(b) - bf(a)}{f(b) - f(a)}$$



# Regula Falsi (метод сечице)

- Услов: постоји интервал  $f(a)f(b) < 0$
- Следећа апроксимација

$$x_1 = \frac{af(b) - bf(a)}{f(b) - f(a)}$$



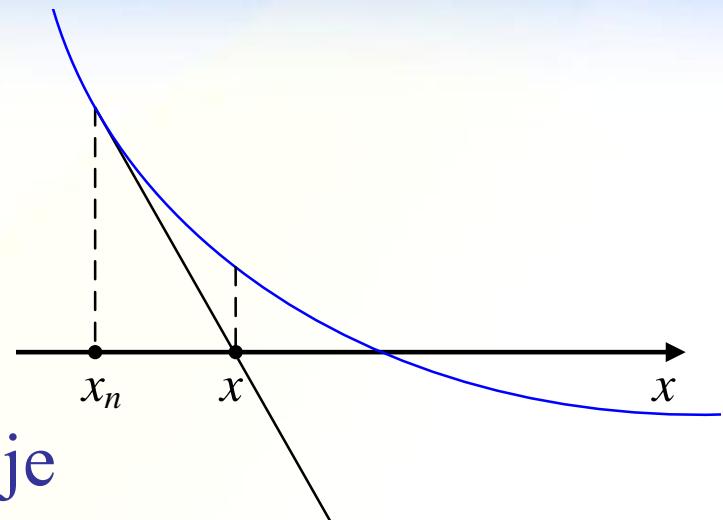
- Нови интервал:  
 $[a, x_1]$  ако  $f(a)f(x_1) < 0$  или  $[x_1, b]$  ако  $f(x_1)f(b) < 0$
- Поновити док се нула не одреди са задатом тачношћу

# Regula falsi: употреба

- Потребно је знати почетни интервал у којем постоји једна нула, а функција мења знак
- Конвергенција је нешто бржа од метода половине интервала
- Мало сложенији за програмирање од половине интервала  
(постоји формула која се програмира)
- Генерализација на вишедимензионе проблеме?

# Њутнов метод: основна идеја и скица извођења

- Такође је познат под именом Newton-Raphson
- Позната функција  $f(x)$  и њен први извод  $f'(x)$
- Једначина тангенте у тачки  $x_n$  је



$$y(x) = f'(x_n)(x - x_n) + f(x_n)$$

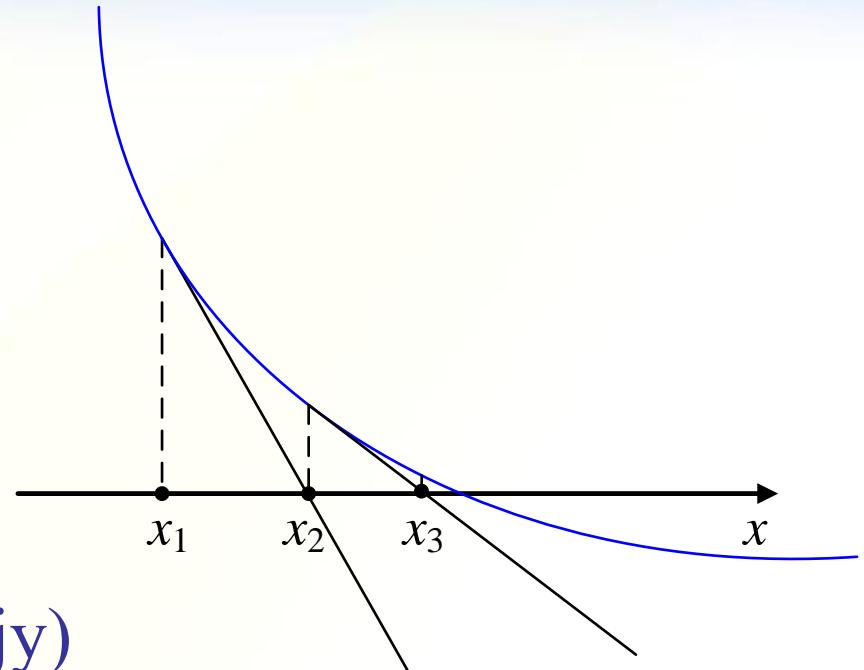
$$0 = f'(x_n)(x - x_n) + f(x_n) \quad \Rightarrow x = x_n - \frac{f(x_n)}{f'(x_n)}$$

# Њутнов метод (апроксимација тангентом)

- Следећа апроксимација:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

- Захтева добро полазно решење (тангента “води” ка нули, није тако у општем случају)



- Потребно је израчунати извод функције, што није увек једноставно

# Њутнов метод: употреба

- Изузетно брза конвергенција
- Тачно или апроксимативно израчунавање извода ограничава употребу
- Уколико функција садржи локални минимум, алгоритам може да дивергира
- Функције са нумеричким шумом нису погодне за овај метод због израчунавања извода
- Генерализација на вишедимензионе проблеме?

# Њутнов метод: генерализација

- Полазимо од система

$D$  нелинеарних једначина, са  $D$  непознатих

$$\left. \begin{array}{l} f_1(x_1, x_2, \dots, x_D) = f_1(\mathbf{x}) = 0 \\ f_2(x_1, x_2, \dots, x_D) = f_2(\mathbf{x}) = 0 \\ \vdots \\ f_D(x_1, x_2, \dots, x_D) = f_D(\mathbf{x}) = 0 \end{array} \right\} \rightarrow \mathbf{f}(\mathbf{x}) = 0$$

- Развијемо једну једначину у Тејлоров ред

$$f_p(\mathbf{x} + \Delta\mathbf{x}) = f_p(\mathbf{x}) + \sum_{k=1}^D \frac{\partial f_p}{\partial x_k} \Delta x_k + O(\Delta\mathbf{x}^2) \quad p = 1, 2, \dots, D$$

# Њутнов метод: генерални итеративни поступак

- Занемарујући чланове  $\Delta\mathbf{x}^2$ , пресек тангенте по једној координати је

$$\sum_{k=1}^D \frac{\partial f_p}{\partial x_k} \Delta x_k = -f_p(\mathbf{x}) \Rightarrow \Delta x_k = -\left( \sum_{k=1}^D \frac{\partial f_p}{\partial x_k} \right)^{-1} f_p(\mathbf{x})$$

- По свим координатама

$$\Delta\mathbf{x} = -\mathbf{J}_f^{-1}\mathbf{f}(\mathbf{x}) \Rightarrow \mathbf{x} + \Delta\mathbf{x} = \mathbf{x} - \mathbf{J}_f^{-1}\mathbf{f}(\mathbf{x})$$

$$\mathbf{J}_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_D} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_D}{\partial x_1} & \frac{\partial f_D}{\partial x_2} & \dots & \frac{\partial f_D}{\partial x_D} \end{bmatrix}$$

- Итеративно

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{J}_f^{-1}\mathbf{f}(\mathbf{x}_i)$$

# Брентов метод (квадратна интерполяција)

- Услов: функција има само један минимум (или максимум) у посматраном интервалу
- Квадратна апроксимација на основу 3 тачке
- Конструисање параболе
- Рачунање минимума параболе
- Понављање метода  
док се не постигне потребна тачност
- Брентов метод обједињује
  - половљење интервала,
  - метод сечице и
  - квадратну интерполяцију

# Квадратна апроксимација

- Три (различита) одбирка функције  $f_k = f(x_k)$

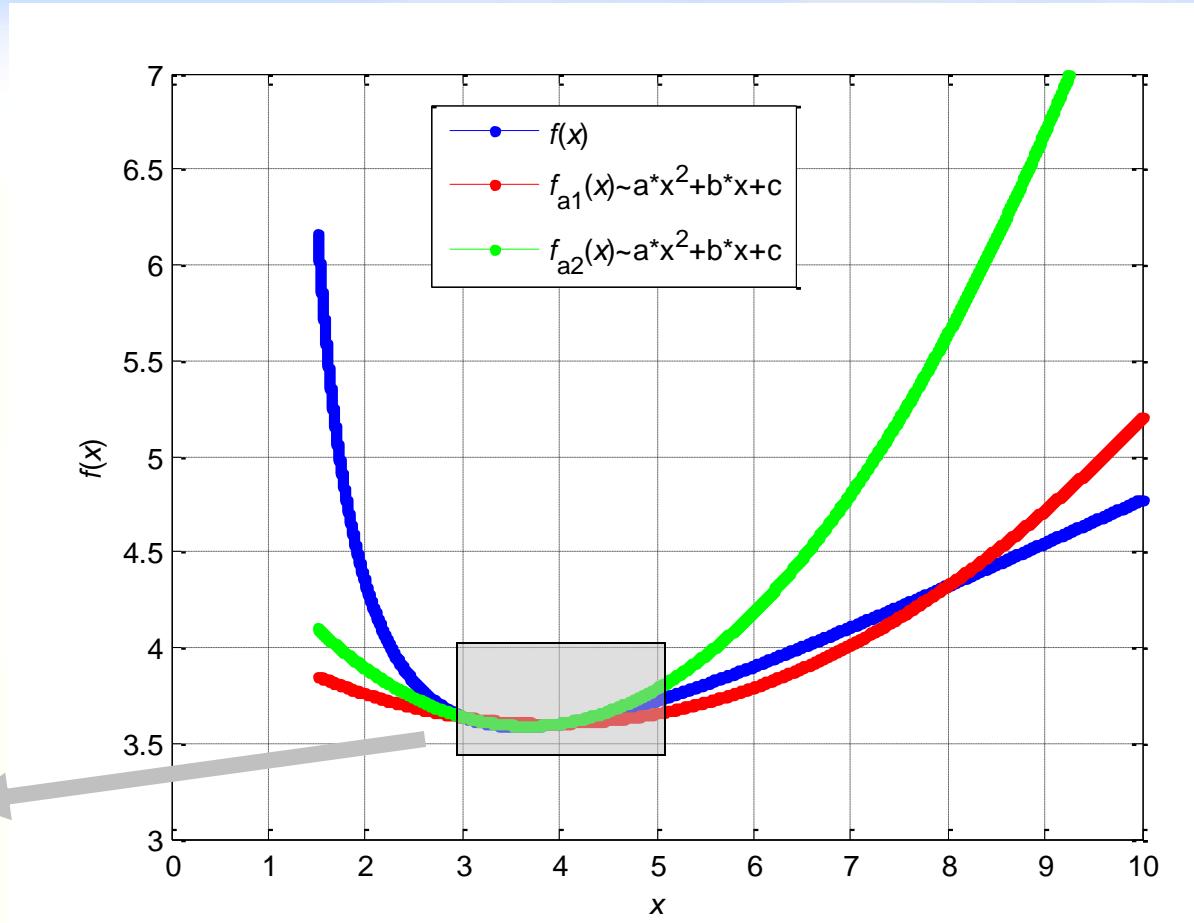
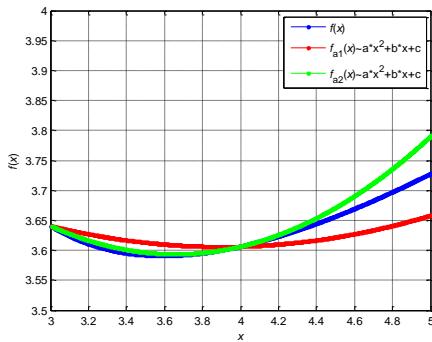
$$\begin{aligned}f_1 &= ax_1^2 + bx_1 + c \\f_2 &= ax_2^2 + bx_2 + c \\f_3 &= ax_3^2 + bx_3 + c\end{aligned}\Rightarrow \begin{bmatrix}x_1^2 & x_1 & 1 \\x_2^2 & x_2 & 1 \\x_3^2 & x_3 & 1\end{bmatrix} \begin{bmatrix}a \\b \\c\end{bmatrix} = \begin{bmatrix}f_1 \\f_2 \\f_3\end{bmatrix}$$

- Коефицијенти  $\begin{bmatrix}a \\b \\c\end{bmatrix} = \begin{bmatrix}x_1^2 & x_1 & 1 \\x_2^2 & x_2 & 1 \\x_3^2 & x_3 & 1\end{bmatrix}^{-1} \begin{bmatrix}f_1 \\f_2 \\f_3\end{bmatrix}$

- Квадратна апроксимација  $f \approx ax^2 + bx + c$
- Пронађемо минимум  $x_{\min} = -\frac{b}{2a}$

# Пример квадратне апроксимације

- Функција  $f$
- Прва аппрокс.  $f_{a1}$
- Друга аппрокс.  $f_{a2}$



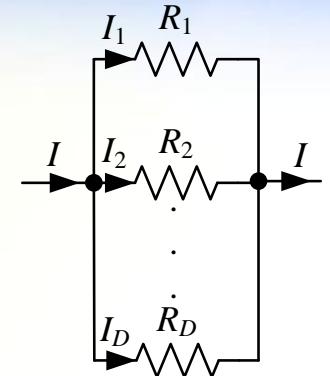
# Лагранжови мултипликатори

- Решавамо проблем  $\text{minimize } f(\mathbf{x}), \mathbf{x} = (x_1, x_2, \dots x_D)$   
$$g_k(\mathbf{x}) = 0, k = 1, 2, \dots m \quad (m < D)$$
- $f(\mathbf{x})$  и  $g_k(\mathbf{x})$  су непрекидна и имају све парцијалне изводе
- Формирали помоћну (нову опт.) функцију  $F(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \sum_{k=1}^m \lambda_k g_k(\mathbf{x})$
- Коефицијенти  $\lambda_k$  су Лагранжови мултипликатори
- Опт. проблем са условима сведен на проблем без услова  
$$\frac{\partial F}{\partial x_1} = 0, \frac{\partial F}{\partial x_2} = 0, \dots, \frac{\partial F}{\partial x_D} = 0; \quad g_1 = 0, g_2 = 0, \dots, g_m = 0$$

компактан запис: 
$$\nabla F = 0 \Leftrightarrow \frac{\partial F}{\partial x_1} \mathbf{i}_{x_1} + \frac{\partial F}{\partial x_2} \mathbf{i}_{x_2} + \dots + \frac{\partial F}{\partial x_D} \mathbf{i}_{xD} = 0$$
- Тачке које задовољавају овај систем једначина су екстремуми (минимуми, максимуми или седласте тачке)
- Проверити вредности свих добијених екстремума

# Пример примене Лагранжових мултипликатора

- У једном чвору електричног кола струја  $I$  дели се у  $D$  паралелно везаних грана
- У свакој грани је један отпорник познате отпорности  $R_1, R_2, \dots, R_D$
- Пронађи струје грана  $I_1, I_2, \dots, I_D$ . тако да је електрична енергија (и снага) овог кола минимална
- Формални запис опт. проблема  $\begin{aligned} & \text{minimize } f(\mathbf{x}) = R_1 I_1^2 + R_2 I_2^2 + \dots + R_D I_D^2 \\ & \mathbf{x} = (I_1, I_2, \dots, I_D) \\ & g_1(\mathbf{x}) = I_1 + I_2 + \dots + I_D - I = 0 \end{aligned}$



# Решење: други Кирхофов закон одговара минимизацији енергије

Помоћна функција је

$$F(I_1, I_2, \dots, I_D, \lambda) = R_1 I_1^2 + R_2 I_2^2 + \dots + R_D I_D^2 - \lambda(I_1 + I_2 + \dots + I_D - I)$$

Парцијални изводи по оптимизационим променљивима су

$$\frac{\partial F}{\partial I_1} = 2R_1 I_1 - \lambda = 0, \quad \frac{\partial F}{\partial I_2} = 2R_2 I_2 - \lambda = 0, \dots \quad \frac{\partial F}{\partial I_D} = 2R_D I_D - \lambda = 0$$

$$I_1 = \frac{\lambda}{2R_1}, \quad I_2 = \frac{\lambda}{2R_2}, \dots, \quad I_D = \frac{\lambda}{2R_D}$$

Уврштавањем израза за струје у услов  $I = g_1 = 0$  добија се

$$\frac{\lambda}{2} \left( \frac{1}{R_1} + \frac{1}{R_2} + \dots + \frac{1}{R_D} \right) = I \Rightarrow \lambda = 2I \left( \frac{1}{R_1} + \frac{1}{R_2} + \dots + \frac{1}{R_D} \right)^{-1} = 2IR_e$$

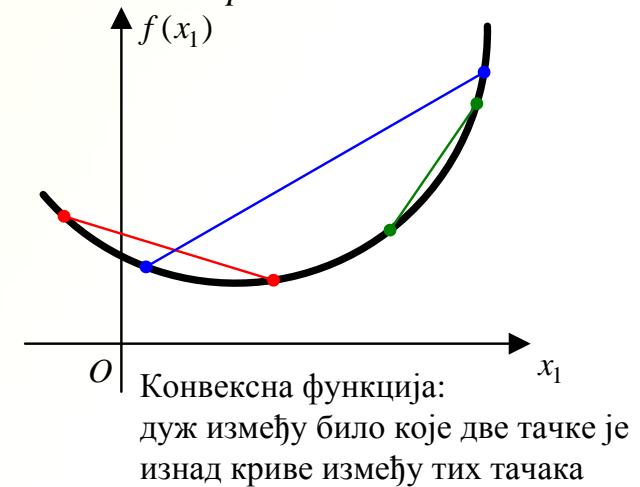
Коначно

$$I_1 = \frac{R_e}{R_1} I, \quad I_2 = \frac{R_e}{R_2} I, \dots, \quad I_D = \frac{R_e}{R_D} I$$

Исти резултат добија се применом другог Кирхофовог закона!

# Karush–Kuhn–Tucker (генерализација Лагранжових мул.)

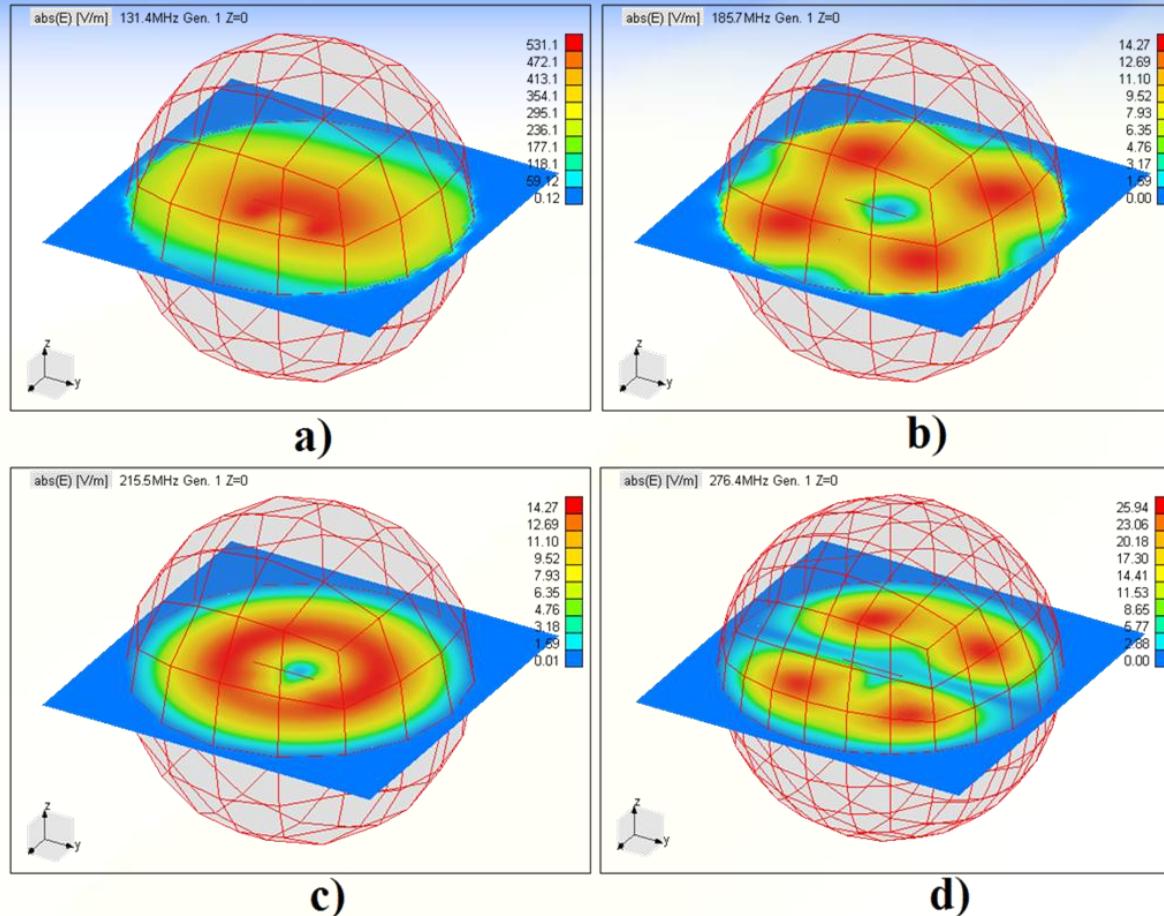
- Решавамо проблем:  $\begin{aligned} & \text{minimize } f(\mathbf{x}), \quad \mathbf{x} = (x_1, x_2, \dots, x_D) \\ & g_k(\mathbf{x}) \leq 0, \quad k = 1, 2, \dots, m \end{aligned}$
- $f, g, h$  су континуалне диференцијабилне функције, а  $f$  је конвексна функција
- Помоћна функција  $F(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \sum_{k=1}^m \mu_k g_k(\mathbf{x}) + \sum_{p=1}^l \lambda_p h_p(\mathbf{x})$
- $\mathbf{x}_0$  је оптимално решење:
  - (1)  $\nabla F(\mathbf{x}_0) + \sum_{k=1}^m \mu_k \nabla g_k(\mathbf{x}_0) + \sum_{p=1}^l \lambda_p \nabla h_p(\mathbf{x}_0) = 0$
  - (2)  $g_k(\mathbf{x}_0) \leq 0, \quad h_p(\mathbf{x}_0) = 0$
  - (3)  $\mu_k \geq 0$
  - (4)  $\sum_{k=1}^m \mu_k g_k(\mathbf{x}_0) = 0$



# Ограничења класичних метода

- Оптимизациона функција не мора имати нулу (ако је потребно пронаћи минимум, онда се тражи нула извода)
- Изводи оптимизационе функције не морају нужно да буду познати, а могуће је и да не постоје!
- Генерализација у случају више променљивих није једноставна за све наведене алгоритме
- За све наведене, сем метода половљења, постоје ситуације када методи дивергирају или стану

# Задатак: одређивање резонантних модова у сферној шупљини



A.J. Krneta, D.I. Olcan, D.H. Trout, "On calculating resonant frequencies using general-purpose method-of-moments code," 29th Annual Review of Progress in Applied Computational Electromagnetics ACES 2013, March 24-28, 2013., Monterey, CA, pp. 804-809.

# Поставка

- Одређивање резонантних модова у сферном резонатору своди се на израчунавање нула сферних Беселових функција
- Сферне Беселове функције прве врсте реда  $n$ ,  $j_n(x)$ , дефинисане су као

$$j_n(x) = \sqrt{\frac{\pi}{2x}} J_{n+\frac{1}{2}}(x),$$

где је  $J_{n+\frac{1}{2}}(x)$  (обична) Беселова функција прве врсте реда  $n + \frac{1}{2}$

- Израчунати нуле  $x_{np0} > 0$ ,  $j_n(x_{np0}) = 0$  **сферних Беселових** функција за редни број нуле  $p = 1, 2$  и ред функције  $n = 1, 2$
- Нуле је потребно одредити са апсолутном тачношћу  $\pm 10^{-12}$

# Имплементација

- Сферне Беселове функције прве врсте реда  $n$ 
  - C/C++17: `<cmath>`  
`std::sph_bessel`  
(`unsigned int n, double z`)
  - Python:  
`scipy.special.spherical_jn`  
(`n, z, derivative=False`)
- Нацртати сферне Беселове функције задатог реда за аргумент из опсега [0,20]
  - Решење је Python код који црта одговарајући график
- Коришћењем једног од класичних метода оптимизације израчунати све потребне нуле са задатом тачношћу
  - Решење је код који израчунава тражене нуле и
  - ASCII фајл са добијеним вредностима за тражене нуле

