

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;
```

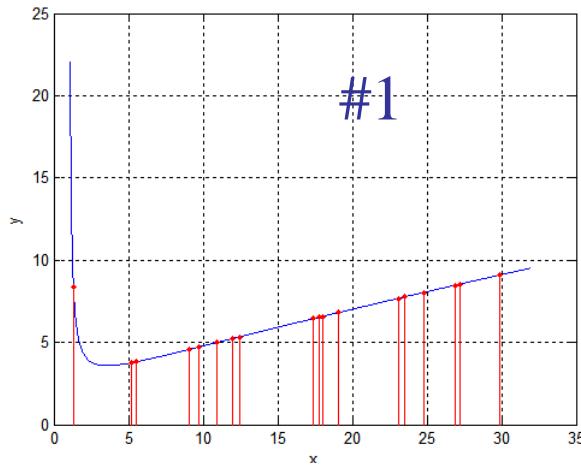
Илустрација алгоритма

		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

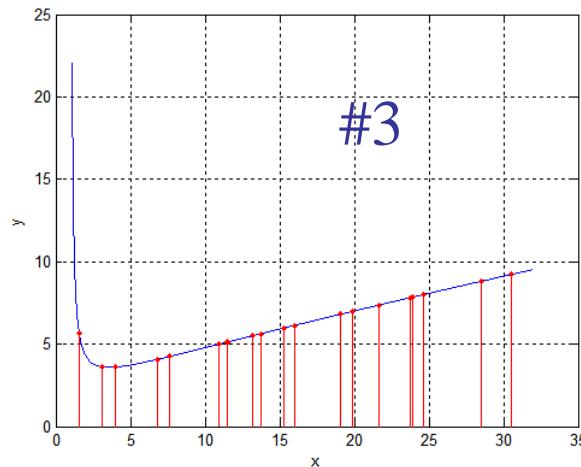
Случајно генерисане партиције и стабла графа

- 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 пресликање таквих низова у стабла графа)
 - Ако граф није потпун, недозвољена стабла се изостављају
 - Постоје рачунарски ефикасније методе

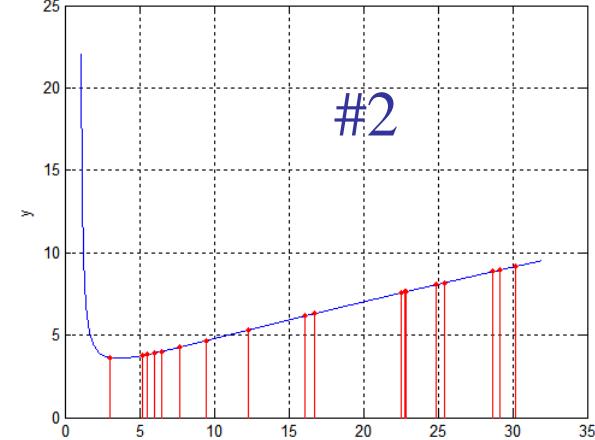
Пример: различита решења при сваком покретању



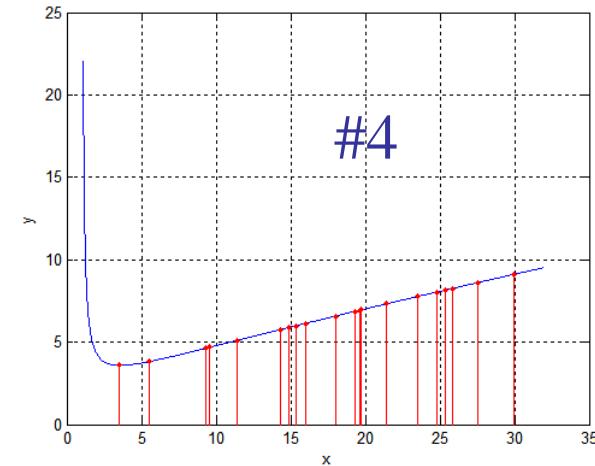
#1



#3

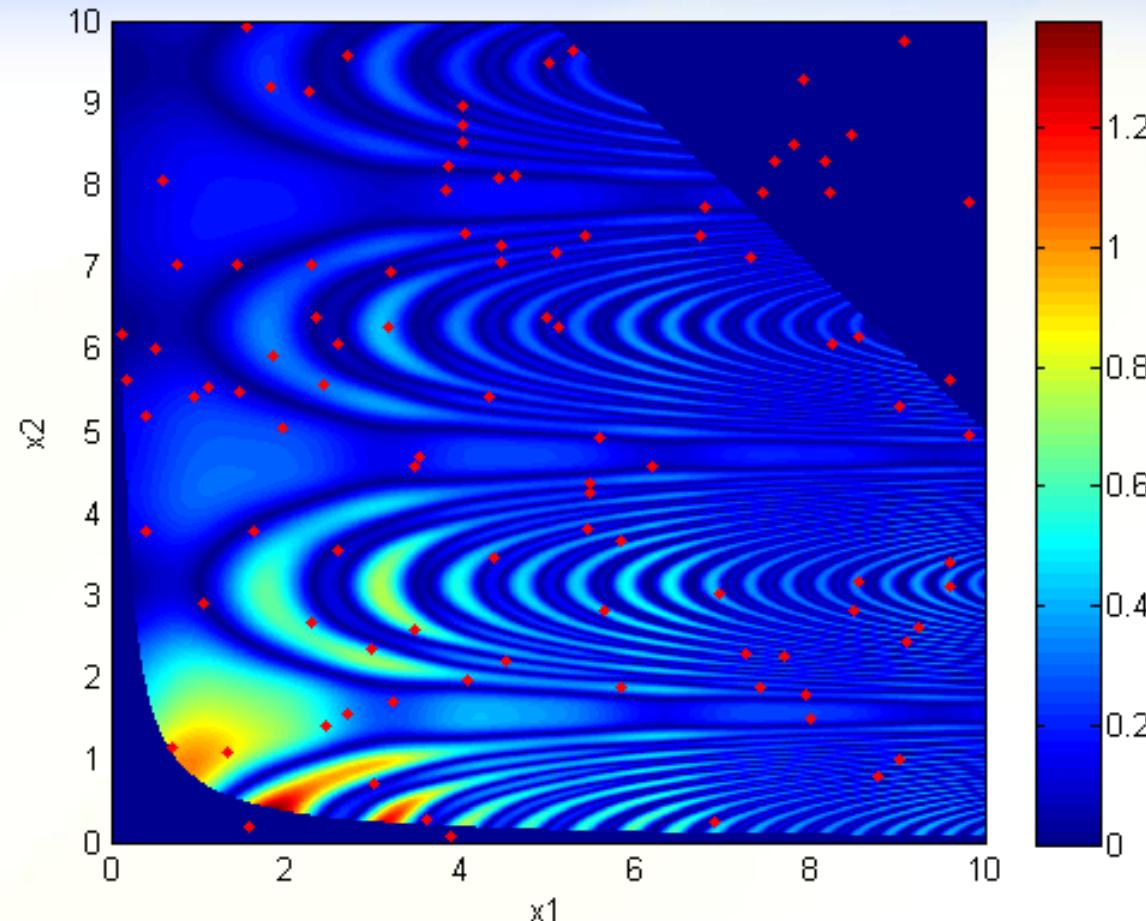


#2



#4

Пример случајног претраживања једне 2D функције (NLP)



О случајним процесима...

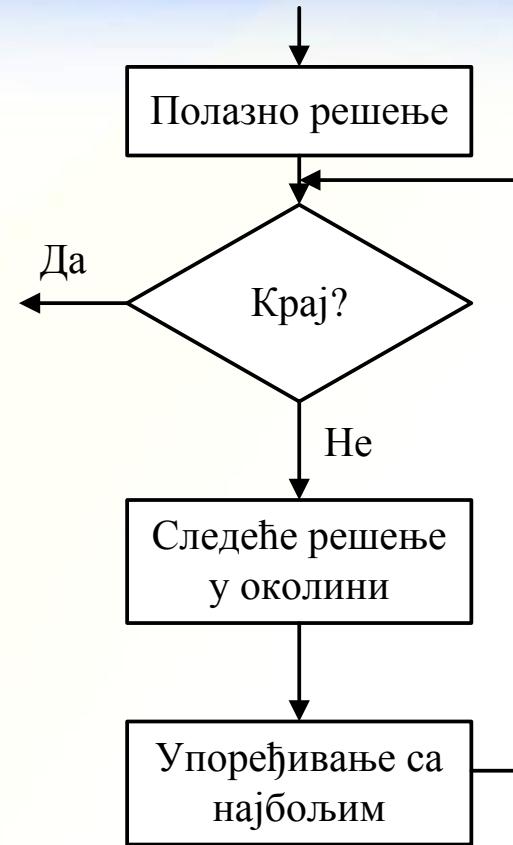
(by Piet Hein)

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

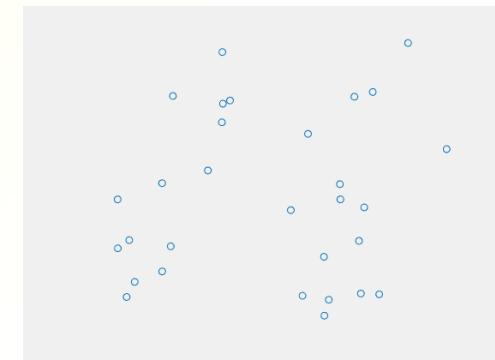
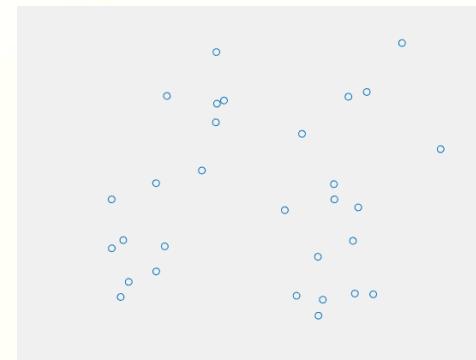
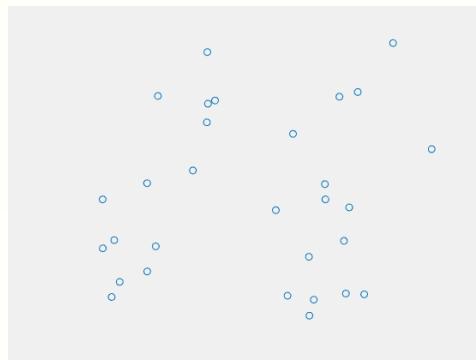
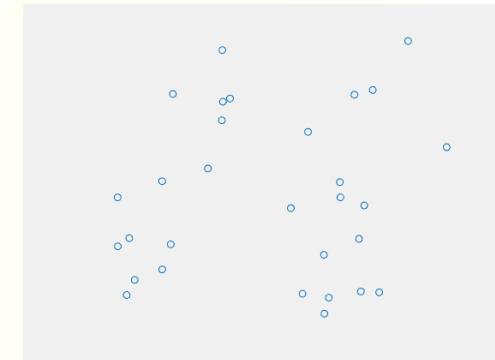
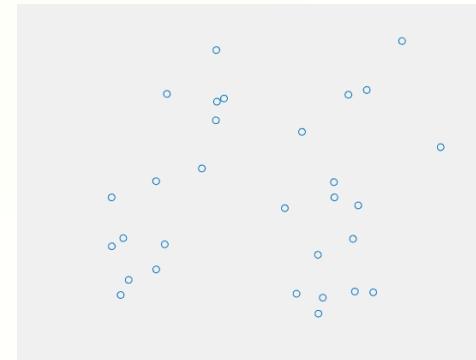
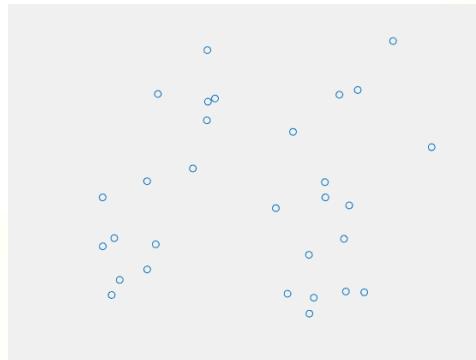
Претрага окolini $k=3$ бита	Све могуће вредности за промену $k=3$ бита
1 0 1 1 0	0 0 0
1 0 1 1 0	0 0 1
1 0 1 1 0	0 1 0
1 0 1 1 0	0 1 1
1 0 1 1 0	1 0 0
1 0 1 1 0	1 0 1
1 0 1 1 0	1 1 0
1 0 1 1 0	1 1 1
1 0 1 1 0	
1 0 1 1 0	

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!)

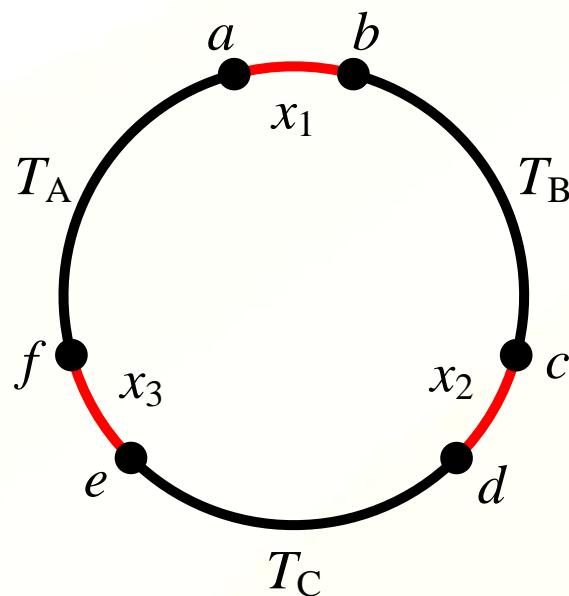
Илустрација проналажења стабла са минималним збиром тежина

- 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



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

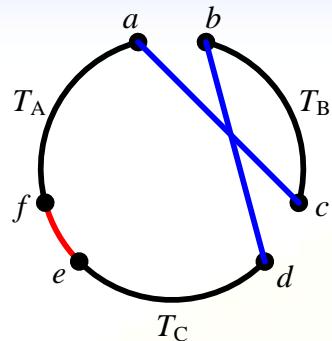
$$L(T) = f(T)$$

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

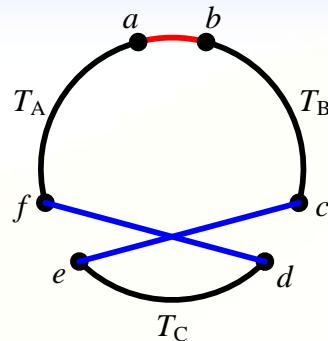
$$f(T) < f(T_0)$$

3-ОПТ: СВИ МОГУЋИ начини повезивања

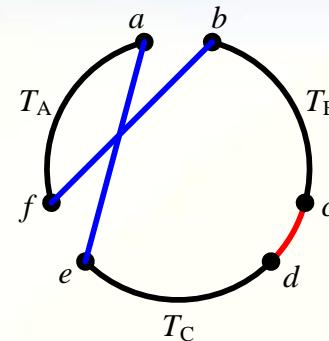
#1: $T_A R_B T_C$



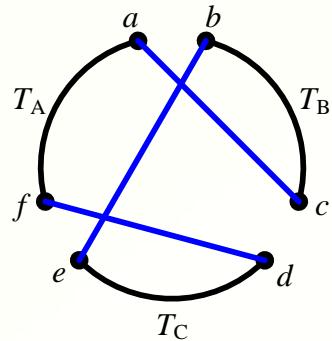
#2: $T_A T_B R_C$



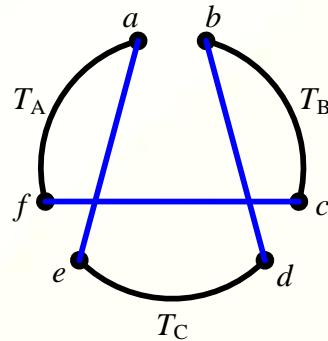
#3: $R_A T_B T_C$



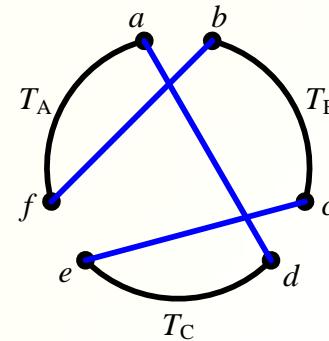
#4: $T_A R_B R_C$



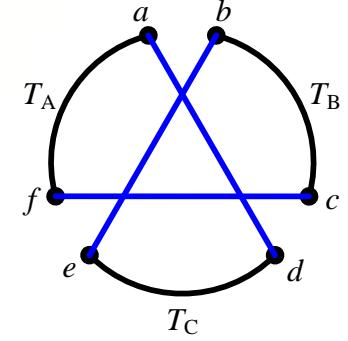
#5: $T_A R_C R_B$



#6: $T_A T_C R_B$

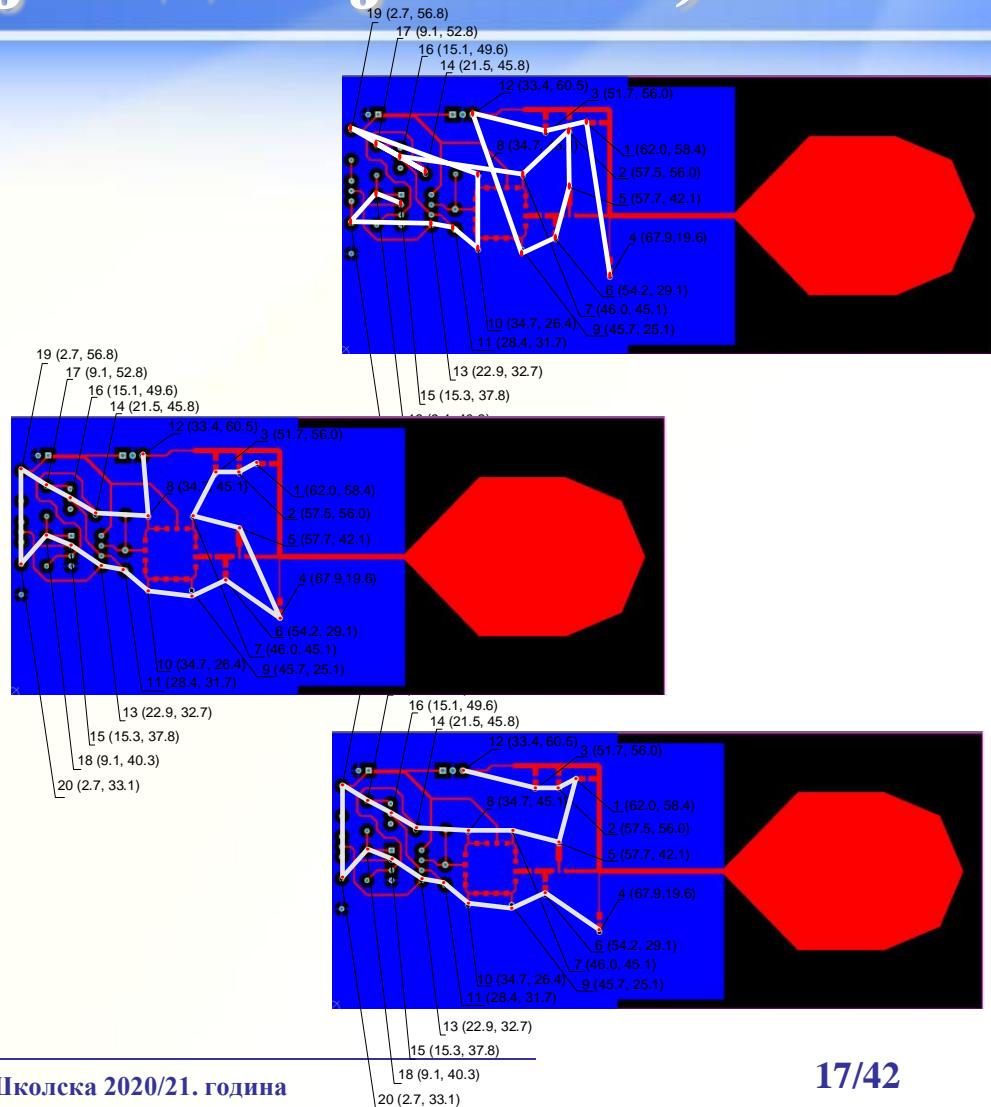


#7: $T_A T_C T_B$



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

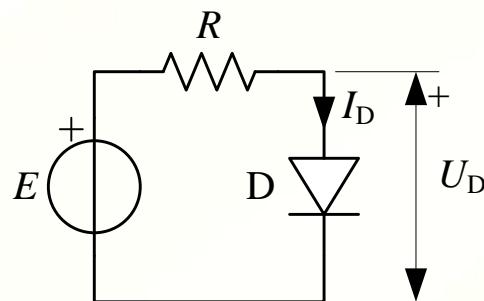
- 3-опт је специјални случај k -опт
 - $k = D$ (број градова) је потпуна претрага
- 3-опт укључује 2-опт (важи генерално)
- Које три везе пресећи?
 - највише $\binom{D}{3} \sim O(D^3)$ ако је D велико постаје непрактично
 - издвојити неке, али које?
- Генерализација
Lin-Kerningan-Helsgaun хеуристика

NLP: континуалне оптимизационе функције

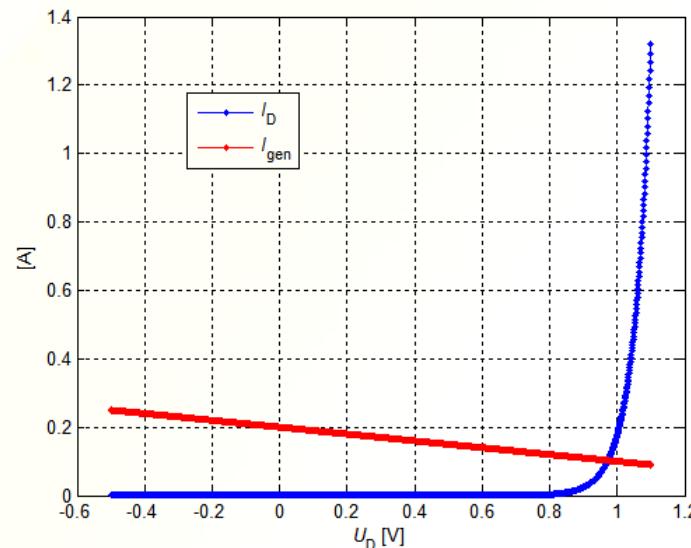
- Проблем:

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

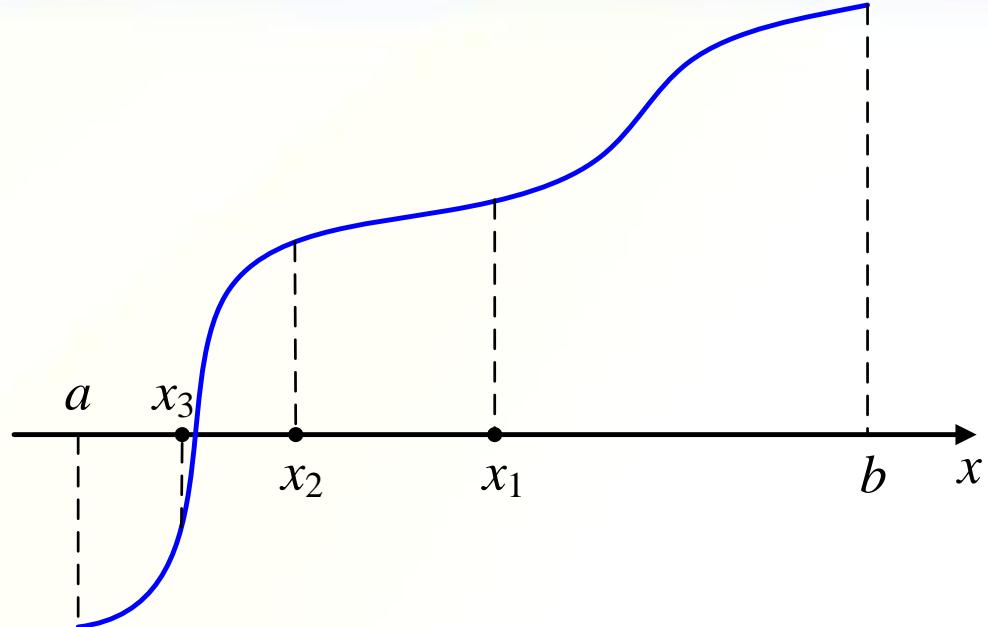
$$f_{\text{opt}}(U_D) = \left| \frac{E - U_D}{R} - I_0 \left(e^{\frac{U_D}{U_0}} - 1 \right) \right| \quad \min(f_{\text{opt}}(U_D)), U_D \in \Re$$

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?

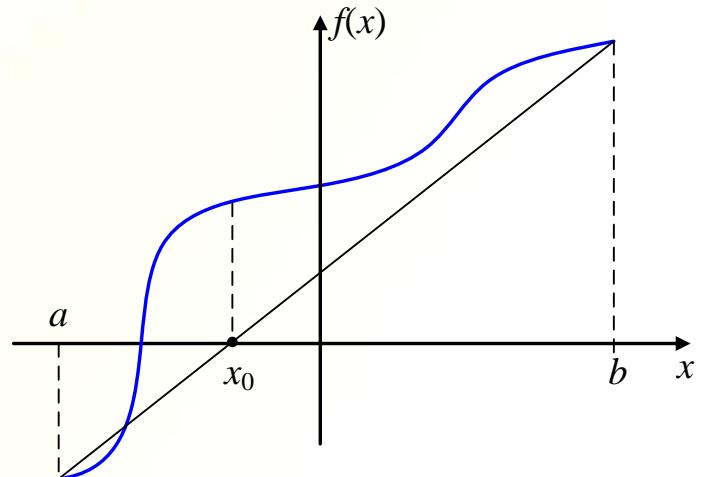
Regula falsi: основна идеја и скица извођења

- Нула се тражи у пресеку линеарне апроксимације функције и апсцисе

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

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

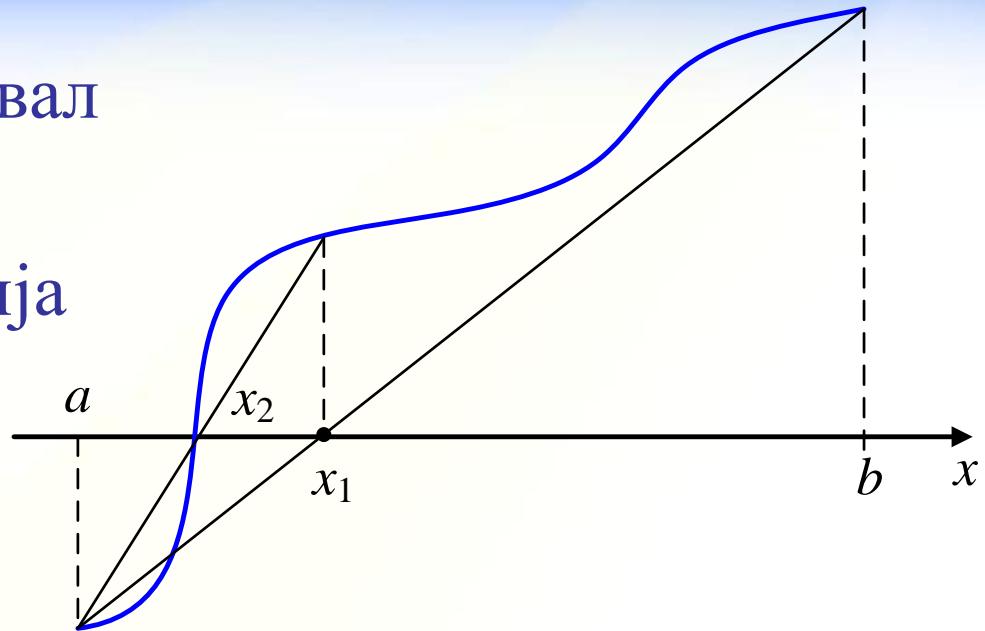
$$x_0 = \frac{af(b) - bf(a)}{f(b) - f(a)}$$



Regula Falsi (метод сечице)

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

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



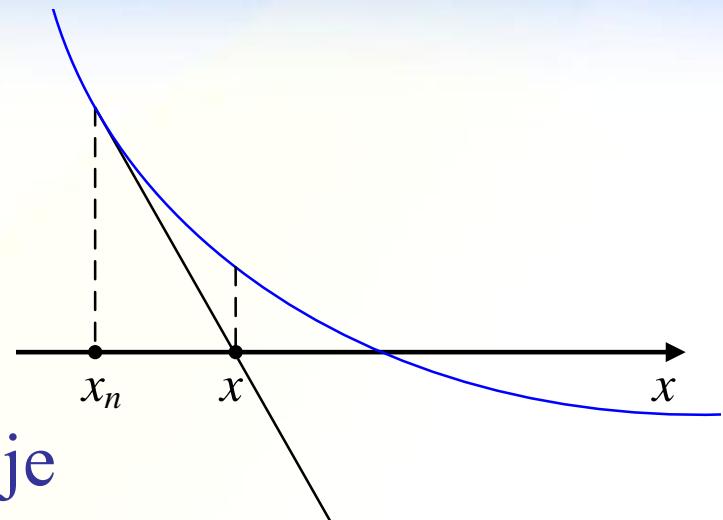
- Нови интервал:
 $[a, x_1]$ ако $f(a)f(x_1) < 0$ или $[x_1, b]$ ако $f(x_1)f(b) < 0$
- Поновити док се нула не одреди са задатом тачношћу

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

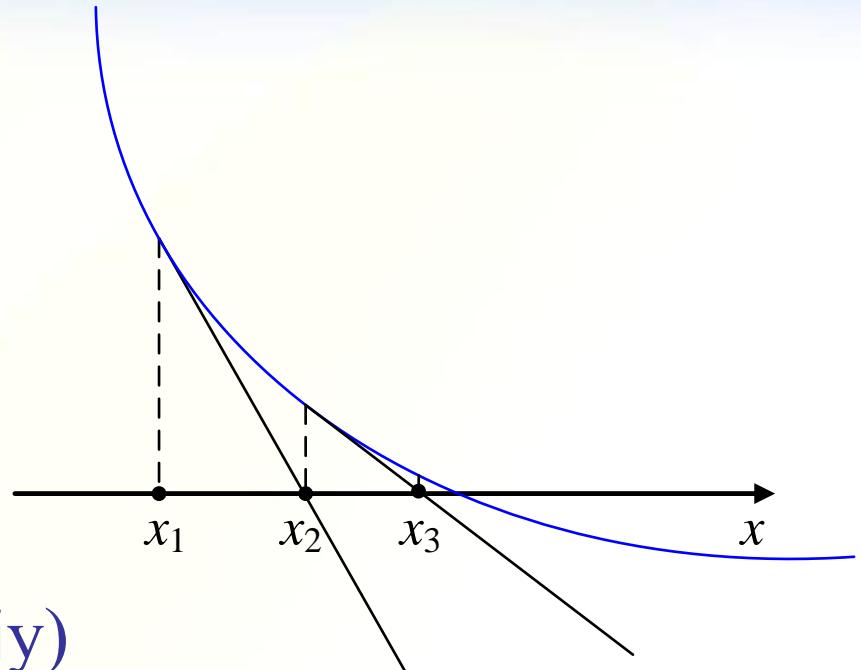


$$y(x) = f'(x_n)(x - x_n) + f(x_n)$$

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

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 $f_k = f(x_k)$

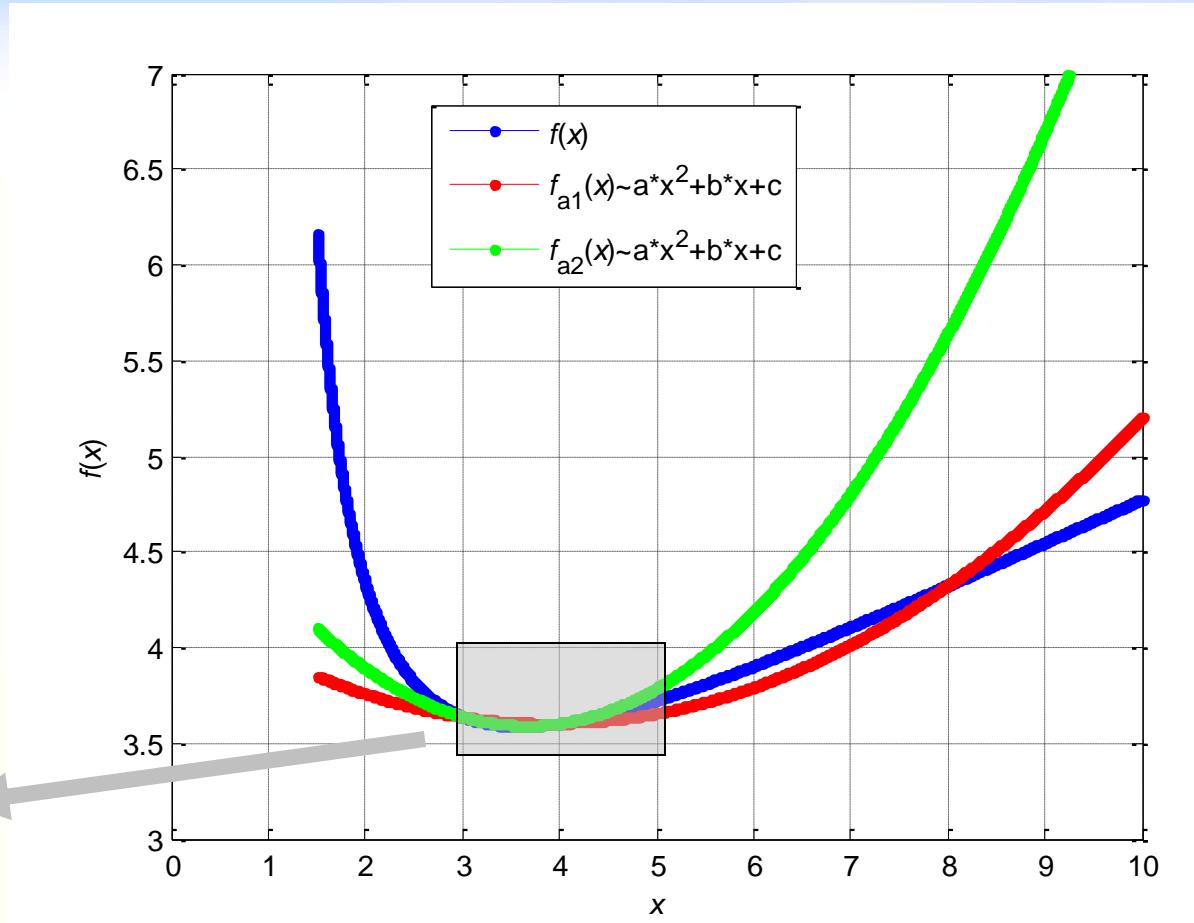
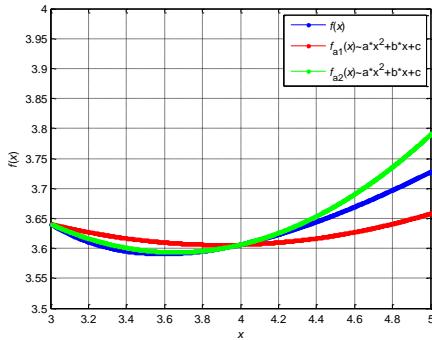
$$\begin{aligned}f_1 &= ax_1^2 + bx_1 + c \\f_2 &= ax_2^2 + bx_2 + c \\f_3 &= ax_3^2 + bx_3 + c\end{aligned}\Rightarrow \begin{bmatrix}x_1^2 & x_1 & 1 \\x_2^2 & x_2 & 1 \\x_3^2 & x_3 & 1\end{bmatrix} \begin{bmatrix}a \\b \\c\end{bmatrix} = \begin{bmatrix}f_1 \\f_2 \\f_3\end{bmatrix}$$

- 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}



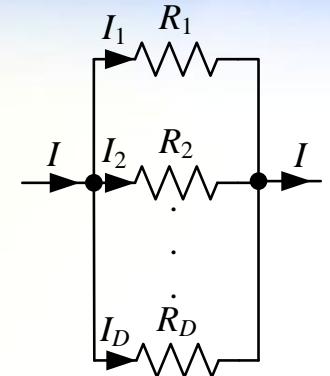
Лагранжови мултипликатори

- 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 λ_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

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

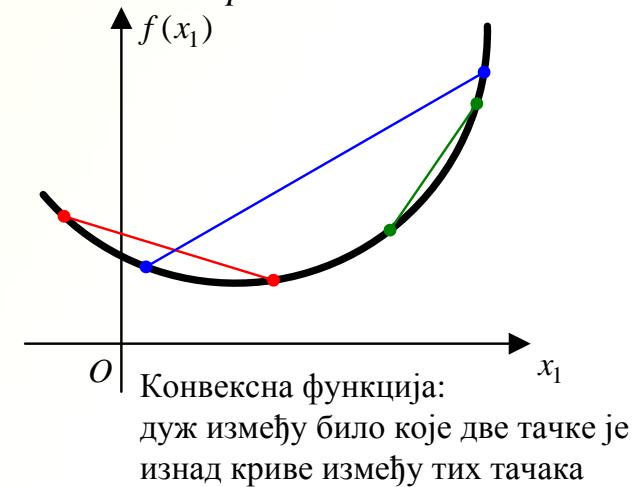
Коначно

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

Исти резултат добија се применом другог Кирхофовог закона!

Karush–Kuhn–Tucker (генерализација Лагранжових мул.)

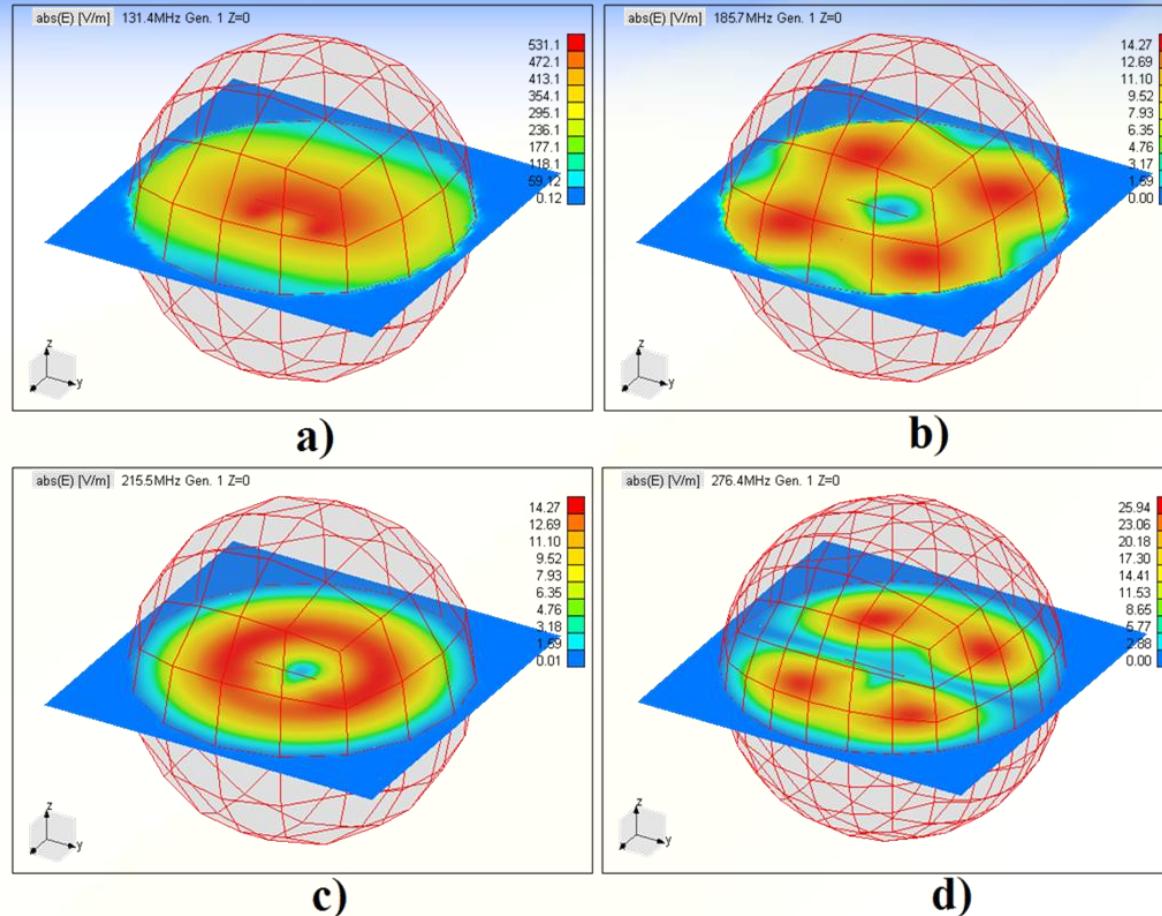
- Решавамо проблем: $\begin{aligned} & \text{minimize } f(\mathbf{x}), \quad \mathbf{x} = (x_1, x_2, \dots, x_D) \\ & g_k(\mathbf{x}) \leq 0, \quad k = 1, 2, \dots, m \end{aligned}$
- f, g, h су континуалне диференцијабилне функције, а f је конвексна функција
- Помоћна функција $F(\mathbf{x}, \boldsymbol{\mu},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

