Zadání

Oscannované zadání práce je v souboru ./pdf/zadani.pdf

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů

Bakalářská práce

Optimalizační algoritmus CMA-ES

Antonín Šulc

Vedoucí práce: Ing. Jan Drchal

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Inteligentní systémy

26. května 2011

Poděkování

Rád bych touto cestou poděkoval všem, kteří mě zasvětili do problematiky evolučních strategií. Mé velké díky patří Ing. Janu Drchalovi, který mě seznámil s celou problematikou. Dále pak Ing. Petru Pošíkovi, Ph.D., který mi v rámci předmětu Aplikace umělé inteligence vysvětlil podstatu evolučních výpočtů a poskytl obecný rámec informací. Rovněž za to, že si na mě vždy našel čas, pokud jsem chtěl cokoli konzultovat. Dále pak RNDr. Petru Olšákovi, který pohotově odpovídal na všechny mé dotazy ohledně matematických problémů souvisejících s algoritmem. Potom své rodině a blízkým přátelům a svému zaměstnavateli Ing. Martinu Rehákovi, Ph.D., kteří se mi snažili pomoci, když jsem začal pociťovat časovou krizi.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 27. 5. 2011 .....

Abstract

This work describes the evolution strategy based on adaptation of covariance matrices for optimization in continuous space. The principle of algorithm dwells in gradual update of covariance matrix and in moving the mean value of normal distribution so as to achieve convergence to some optima in successive generations. This work proposes a detailed analysis of each step of the algorithm including the possible options or modifications and the known implementation. It also includes a description of CMA-ES, IPOP-CMA-ES and σ-**m**-IPOP-CMA-ES implementations in Java programming language into JCool library. A series of experiments were performed using the method, comparing between algorithms available in JCool library, including the progress of algorithm on the selected test functions.

Abstrakt

Tato práce popisuje evoluční strategii založenou na adaptaci kovariančních matic pro optimalizaci v prostoru reálných funkcí. Podstatou algoritmu je postupná modifikace kovarianční matice a přesouvání střední hodnoty normálního rozdělení tak, aby algoritmus konvergoval s postupem generací k některému z optim. V této práci je podrobně analyzován každý krok algoritmu včetně možných variant, modifikací a známých implementací. Dále je pak v práci popsána implementace v jazyce Java do knihovny JCool a to jak algoritmu CMA-ES tak i IPOP-CMA-ES. Na metodě je provedena řada experimentů porovnání mezi algoritmy, které jsou k dispozici v knihovně JCool, včetně grafů průběhu algoritmu na vybrané skupině testovacích funkcí.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

1.1 Evoluční algoritmy, techniky a výpočty

Definice Evoluční algoritmy, techniky a výpočty: Jsou metaheuristické optimalizační algoritmy, jenž jsou podoblastí umělé inteligence. Evoluční algoritmy jsou tvořeny populací, nad níž jsou definovány operátory selekce, křížení a mutace. Každý prvek z populace představuje jedno z možných řešení.

1.1.1 Vlastnosti a popis

Jak bylo řečeno v definici, tak evoluční algoritmus (dále jen EA) je tvořen populací o n jedincích, nad nimiž probíhá v každé generaci g adaptace a evoluce. Úkolem je najít globální optimum. Protože se jedná o metaheuristiku, není zaručeno, že toto vybrané řešení $\mathbf{x}_i$ je optimální. EA vytvořili mezistupeň mezi deterministickými gradientními technikami a technikami náhodného prohledávání.

Jedinec je určitou reprezentací, pro kterého jsme schopni vyhodnotit jeho kvalitu pomocí fitness funkce f. Reprezentace každého jedince $\mathbf{x}_i$ je *genotyp* (složený z atomických *genů*) a reprezentuje určitý *fenotyp*, pro něž jsme schopni vyhodnotit jeho fitness $f(\mathbf{x}_i)$.

Nejznámější EA jsou:

- **Genetický algoritmus** (dále jen GA), v každé generaci jsou jedinci, kteří jsou reprezentováni binárním genotypem. Obecný GA je popsán v algoritmu 1.
- **Genetické programování**, v každé generaci jsou jedinci, kteří jsou reprezentací jednoho z možných postupů (např. struktura, popis atp.) v řešení daného problému. Jejich kvalitu měří schopnost řešit daný problém.
- **Evoluční programování**, je velmi podobné genetickému programování s tím rozdílem, že reprezentace jednoho jedince $\mathbf{x}_i$ je pevně daná. Mění se pouze jeho parametry viz. [19].

input : Fitness funkce $f : D_f \to \mathbf{R}$.
output: Metaheuristický odhad nejlepší hodnoty funkce f

$g := 0$;
$(\forall \mathbf{x}_i = rand\,()) \in \mathbf{X}^{(0)}$;
vyhodnoť kvalitu $f\,(\mathbf{x}_i)\ \forall \mathbf{x}_i \in \mathbf{X}^{(0)}$;
while *zastavovací podmínka není platná* **do**
  $\mathbf{X}^{(g)}_{selekce} = selekce\left(\mathbf{X}^{(g)}\right) \subseteq \mathbf{X}^{(g)}$;
  $\mathbf{X}^{(g)}_{operatory} = operatory\left(\mathbf{X}^{(g)}_{selected}\right)$;
  vyhodnoť kvalitu $f\,(\mathbf{x}_i)\ \forall \mathbf{x}_i \in \mathbf{X}^{(g)}_{operatory}$;
  $\mathbf{X}^{(g+1)} = vyber\_nejlepsi\left(\mathbf{X}^{(g)}_{operatory}\right)$;
  $g := g + 1$;
end
. Funkce *vyber_ nejlepsi* nemusí být vždy použita.

Algorithm 1: Základní genetický algoritmus

- **Evoluční strategie** (dále jen ES), v každé generaci jsou jedinci reprezentováni vektory $\mathbf{x}_i$, ale na rozdíl od GA je genotyp přímou reprezentací (fenotypem) hodnoty vstupující do fitness funkce $f\,(\mathbf{x}_i)$. Vlastnosti nových jedinců v populaci jsou převážně navzájem korelované s původní populací a s ohledem na jejich kvalitu.

1.1.2 Operátory genetických algoritmů a evolučních strategií

V následujícím pojednání se pokusím vysvětlit principy a postupy operátorů používaných v EA. Genetické operátory nelze zobecnit pro všechny algoritmy. Každý operátor je charakteristický tím, že vybírá z populace (selekce), vytváří novou informaci v populaci (mutace) nebo sdílí informaci mezi jedinci (křížení, rekombinace).

1.1.2.1 Selekce

Definice Selekce: Je operátor, jenž vybírá jedince z populace, na než bude použit genetický operátor

---

Selekční operátory vybírají lepší jedince s vyšší pravděpodobností.

Například u GA existuje operátor selekce ruletovým kolem [5]. Tento selekční operátor setřídí populaci a ke každému jedinci vypočte poměr jeho kvality vůči celkové kvalitě populace. Tento poměr tvoří poměr výseče na tzv. ruletovém kole. Následně se na tomto ruletovém kole vybere náhodné místo, které ukazuje na vybraného jedince.

Někdy může být výběr i deterministický, jako např. u ES, která je založená na adaptaci kovariančních matic (dále jen CMA-ES). Selekční operátor z celé populace vybere množinu o velikosti μ nejlepších jedinců a z nich vypočte váženou střední hodnotu.

1.1.2.2 Křížení, rekombinace

Definice Křížení, rekombinace: Je, je n-nární operátor, který provádí nad dvěma genotypy vzájemnou výměnu informace mezi jedinci.

Nejběžnější operátor křížení je jednobodové křížení v GA. Termín jednobodové křížení nad dvěma chromozomy délky N si lze představit tak, že zvolím náhodný bod $0 < r < N$, přičemž všechny geny v reprezentaci jednoho jedince se od prvního do r-tého genu nahradí geny druhého jedince a geny druhého jedince se od r-tého do posledního (N-tého) genu nahradí geny na odpovídajících pozicích chromosomu prvního jedince. Výsledkem jsou dva chromozomy, dva noví jedinci. Geometricky si lze jednobodové křížení představit tak, že nově vytvoření jedinci z křížení se mohou nacházet na hlavních osách každého z jedinců. Princip je na obrázku 1.1.2.2 a na obrázku 1.1.2.2 kde je geometrická interpretace pro binární reprezentaci genotypu. Křížení lze samozřejmě zobecnit na m-bodové křížení, mezi n jedinci. Nejběžněji používaný v GA je ovšem binární jedno-až-dvou bodový operátor křížení.

V případě, že je ES označena jako $(\mu/\varrho, \lambda)$ nebo $(\mu/\varrho + \lambda)$ a hodnota parametru ϱ je větší než 2, může docházet k tzv. multirekombinaci. Multirekombinaci se také někdy říká diskrétní křížení [25]. Multirekombinace je operátor, kde do křížení vstupuje ϱ jedinců z μ vybraných jedinců. Multirekombinace probíhá tak, že z populace selekcí vybere μ jedinců, z nichž se náhodně vybere ϱ jedinců. Následně z této množiny vybere operátor každý gen náhodně (prvek vektoru reprezentace) 1 až N z ϱ jedinců. Proces ilustruje obrázek 1.1.2.2.

Křížení probíhá u všech ES, které mají $\varrho \geq 2$, pokud $\varrho > 2$ mluvíme o tzv. multirekombinaci. Podstatným rozdílem operátoru křížení u GA a ES je, že výsledkem křížení jsou opět dva jedinci, zatímco u ES vygenerujeme nového jedince pouze jednoho nebo více, ale s podobnými vlastnostmi.

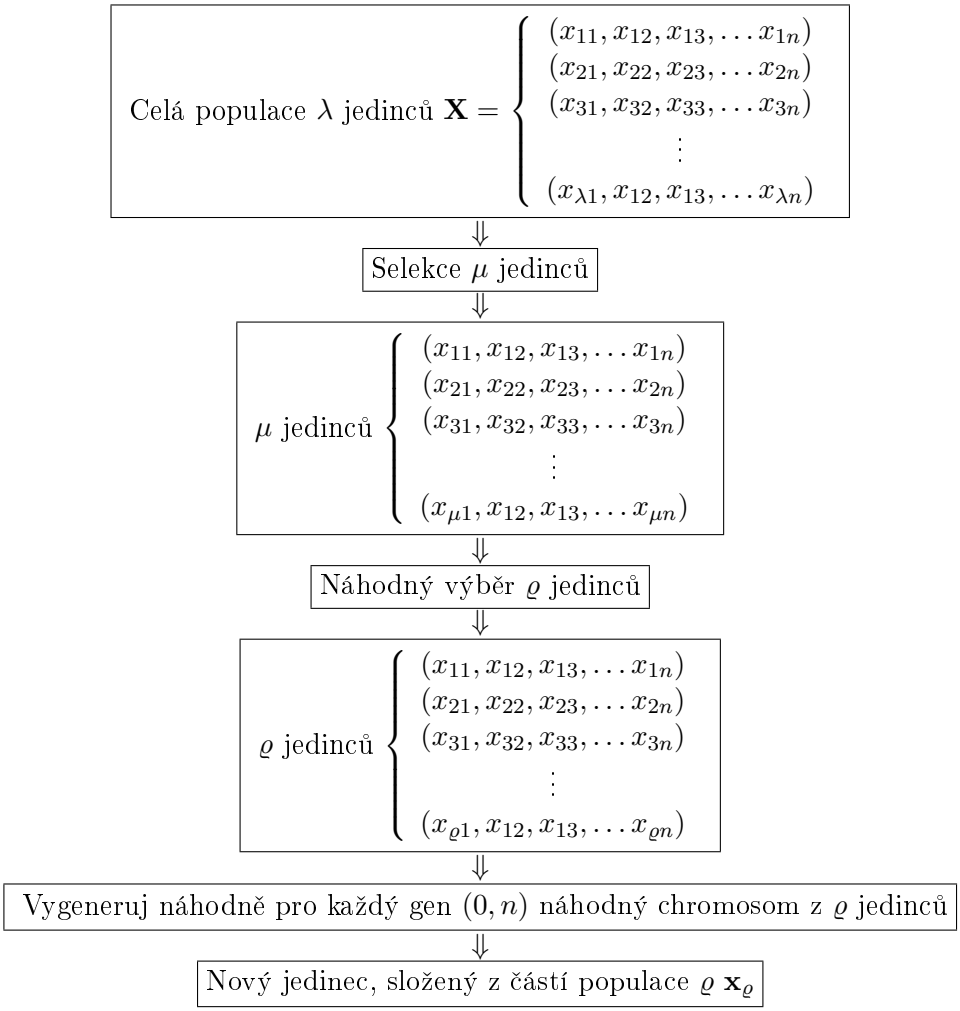
Podstatný je fakt, že do křížení vstupuje generace předchozí, která předá částečně svou informaci generaci nové.

1.1.2.3 Mutace

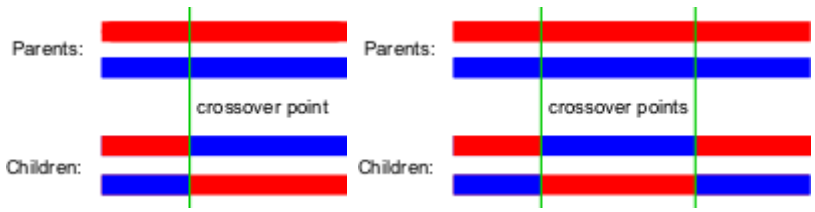
Definice Mutace: Je u GA operátor náhodné změny hodnoty genu. V případě ES je to nahrazení celého genotypu genem podobným. U mutace vzniká nová, potenciálně vhodná informace.

Příkladem mutace u GA je prohození genu/genů v daném genotypu. Geometricky si lze představit mutaci nad binárním genotypem po převodu na fenotyp podle obrázku 2. Jedinec je reprezentován chromozomem, kde každý gen může být reprezentován binárně $\{0, 1\}$. Výchozí jedinec je reprezentován genotypem $(1, 0, 0, 0; 1, 0, 0, 0)$ a nachází se na souřadnicích $(8, 8)$. Prohozením libovolného bitu docílíme ve dvou rozměrech translaci jedince do jednoho ze směrů hlavních os souřadného systému (viz obrázek 1.1.2.2).

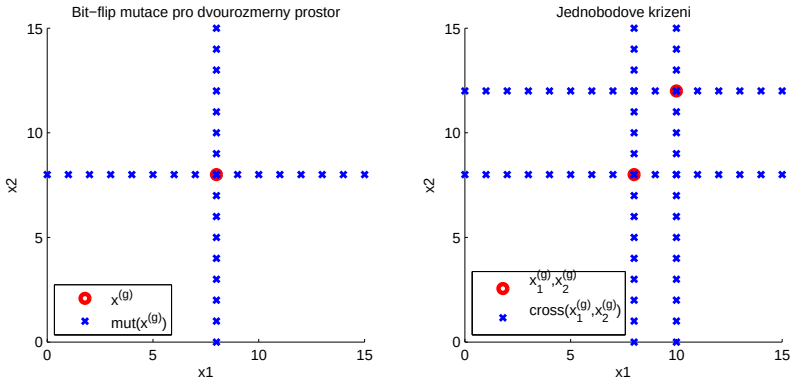
V ES je přístup poněkud jiný. ES reprezentují jedince pomocí vektorů reálných čísel a genotyp je zároveň i fenotyp. V případě mutace ES se využívá podmnožiny, případně celé



Obrázek 1.1: Průběh multirekombinace pro $(\mu/\varrho, \lambda)$ ES



Obrázek 1.2: Jedno a dvoubodové křížení v GA
[32] *Levý*:Jednobodové křížení, *Pravý*:Dvoubodové křížení. *Crossover point* označuje bod křížení, tedy náhodnou hodnotu $0 < r < N$. [32]



Obrázek 1.3: Ukázka křížení u GA
Levý: Mutace jednoho bitu u jedince reprezentovaným binárním řetězcem o délce 8 bitů, *Pravý*: Jednobodové křížení u jedinců reprezentovaných binárním řetězcem o délce 8 bitů

populace, z níž vytvořím novou populaci, částečně korelovanou s původní. Pro generování nové populace se většinou používá normální rozdělení.

V případě CMA-ES probíhá mutace tak, že stávající populace jedinců je plně nahrazena novými, normálně rozdělenými jedinci, přičemž parametry pro normální rozdělení bere ze stávající populace speciálním výpočtem kovarianční matice a její střední hodnoty (střední hodnota, jak jsem již zmínil je parametr selekce).

Obdobně jako u křížení, resp. rekombinace nelze popis zobecnit, neboť se liší pro různé algoritmy. Obecným rysem mutace je, že vnáší novou náhodnou složku do stávající populace, která může být někdy výhodná pro lepší prohledání stavového prostoru.

1.2 Evoluční strategie

Evoluční strategie, je další z kategorií evolučních výpočtů. ES se liší od GA tím, že využívá stávající reprezentaci jedinců k tomu, aby mohla vygenerovat populaci novou většinou pomocí normálního rozdělení. Populace jako celek tvoří reprezentaci nějakého stavu v prostoru řešení a vlastností celku nebo její podmnožiny. Využije při tom operátory k vytvoření nové populace. Navzájem se od sebe ES mohou lišit různými přístupy k výpočtům parametrů normálního rozdělení.

1.2.1 Dělení evolučních strategií

Jelikož práce s populací lze u ES zobecnit, byla vytvořena metodika označování kategorií ES. Označení říká co je vstupem a výstupem jednoho kroku (jedné generace) dané ES. Symbolem μ značíme populaci, jenž byla vybrána k vytvoření nových jedinců λ.

1.2.1.1 (μ, λ)-ES

Z populace se vybere μ jedinců, kteří vytvoří λ jedinců do populace nové.

$$\mu \quad \rightarrow \quad (\mu, \lambda)\text{-ES} \quad \rightarrow \quad \lambda \tag{1.1}$$

1.2.1.2 $(\mu + \lambda)$-ES

Z populace se vybere μ jedinců, která vytvoří λ jedinců a výsledkem je sloučení μ a λ. Pro udržení konstantní velikosti populace v generacích je resp. může být $\lambda^{(worst)}$ jedinců odstraněno.

$$\begin{array}{ccccc} \mu & \rightarrow & (\mu + \lambda)\text{-ES} & \rightarrow & \mu + \lambda \\ & \searrow & \mu & \nearrow & \end{array} \tag{1.2}$$

1.2.1.3 $(\mu/\varrho, \lambda)$-ES

Z populace se vybere μ jedinců, kteří vytvoří λ jedinců do populace nové. V průběhu ES je použit i operátor rekombinace, který je prováděn nad ϱ jedinci.

$$\mu \quad \rightarrow \quad (\mu/\varrho, \lambda)\text{-ES} \quad \rightarrow \quad \lambda \tag{1.3}$$

Příkladem může být základní algoritmus CMA-ES, který je označován jako $(\mu/\mu_W, \lambda)$-ES, čímž říká, že do rekombinace vstupuje μ jedinců, kteří jsou vážení jistou váhou (podle subskriptu W). V případě algoritmu CMA-ES označení říká, že křížení probíhá nad $\mu_W = \mu$ jedinci.

Znakem ϱ nemusíme značit mutirekombinaci, nicméně informace za lomítkem je nositelem informace o tom, co se děje v křížení. Což by mělo u odpovídající ES blíže specifikováno.

1.2.1.4 $(1 + \lambda)$-ES

Z populace se vybere jeden jedinec, který vytvoří λ jedinců do nové populace.

$$\begin{array}{ccccc} 1 & \rightarrow & (1 + \lambda)\text{-ES} & \rightarrow & 1 + \lambda \\ & \searrow & 1 & \nearrow & \end{array} \tag{1.4}$$

1.2.1.5 $(1 + 1)$-ES

Z populace se vybere jeden jedinec, který vytvoří jednoho jedince do nové populace.

$$\begin{array}{ccccc} 1 & \rightarrow & (1 + 1)\text{-ES} & \rightarrow & 1 + 1 \\ & \searrow & 1 & \nearrow & \end{array} \tag{1.5}$$

Ve své podstatě je tato ES tzv. hill-climbing technikou, neboť ve své jedné generaci vytvoří nového jedince. Ten se přidá do stávající populace a buď zvítězí a zůstane, nebo, je-li lepší, bude odstraněn.

1.3 Matematický aparát

1.3.1 Newtonova metoda

Newtonova metoda, jak ukáži později, má velmi blízko k algoritmu CMA-ES. Metoda je založena na vlastnostech inverzní Hessovy matice, která je právě algoritmem CMA-ES aproximována.

Je to iterativní metoda pro hledání kořenů rovnic. Lze ji ovšem použít i pro hledání inflexních bodů funkcí, které mají druhou derivaci. Takže hledáme poblíž lokální optima.

Budu předpokládat, že mám zadanou funkci $f : \mathbf{R} \to \mathbf{R}$, která má spojité druhé derivace, zadaný výchozí bod x_0 a hledám inflexní bod x^* (první derivace je v bodě x^* nulová). Budu aproximovat funkci f Taylorovým polynomem druhého stupně podle rovnice 1.6.

$$f(x + \Delta x) = f(x) + \frac{df}{dx}(x)\,\Delta x + \frac{1}{2}\frac{d^2 f}{dx^2}(x)\,\Delta x^2 \tag{1.6}$$

Δx je krok algoritmu, resp. délka kroku. Hledám takové Δx, které je co nejblíže nule. Toto získám zderivováním výrazu 1.6 podle Δx a vznikne:

$$\begin{gathered}
\frac{df}{dx}(x) + \frac{d^2 f}{dx^2}(x)\,\Delta x = 0 \\
\frac{df}{dx}(x) + \frac{d^2 f}{dx^2}(x)\,(x_n - x_{n-1}) = 0 \\
x_n - x_{n-1} = -\frac{\frac{df}{dx}(x_n)}{\frac{d^2 f}{dx^2}(x_n)} \\
x_n = x_{n-1} - \frac{\frac{df}{dx}(x_n)}{\frac{d^2 f}{dx^2}(x_n)}
\end{gathered} \tag{1.7}$$

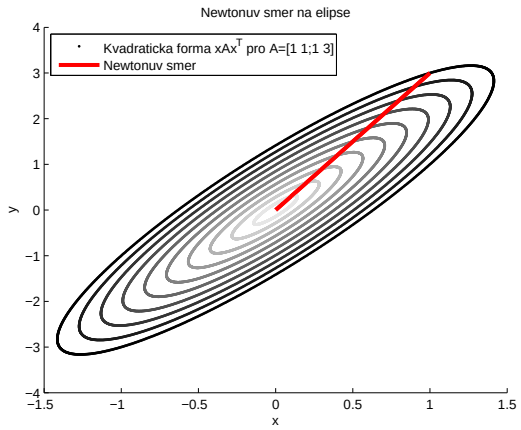
vyřeším-li tuto rovnici, odvodím rovnici pro Newtonovy metody [34]. Obyčejné gradientní metody berou v potaz pouze okamžitou hodnotu gradientu v určitém bodě, tzn. směr kolmý na tečnu vrstevnice. U Newtonovy metody, pokud se bod, ze kterého vycházíme nachází poblíž optima, směřuje Newtonův směr přímo ve směru optima. Na obrázku 1.3.1 je ukázka ideálního směru Newtonova algoritmu pro kvadratickou funkci s nesymetrickou maticí $\mathbf{A} = \begin{pmatrix} 1 & 1 \\ 1 & 3 \end{pmatrix}$ pro dvourozměrný případ.

Obrázek 1.3.1 ukazuje řezy po 0.1 krocích kvadratickou funkcí.

Uvedl jsem výpočet pro jednorozměrný případ. Pro vícerozměrný případ je nahrazena první derivace gradientem $\bigtriangledown f$ a $\left(\frac{d^2 x}{dx^2}\right)^{-1}$ inverzním Hessiánem $\mathbf{H}^{-1}$ funkce f a pro vícerozměrný případ rovnice vypadá následovně:

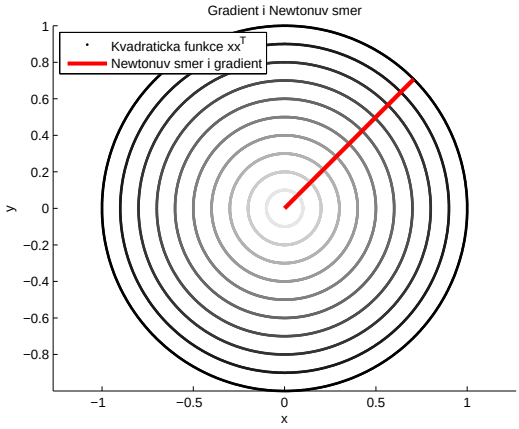
$$\mathbf{x}_n = \mathbf{x}_{n-1} - \mathbf{H}^{-1} \bigtriangledown f(\mathbf{x}_{n-1}) \tag{1.8}$$

Obecně lze samozřejmě aproximovat libovolnou funkci. Pro ukázku jsem na obrázcích použil kvadratickou funkci, která není symetrická podle hlavních os. Pokud rozložím matici $\mathbf{H}^{-1}$ na vlastní čísla a vlastní vektory, transformuje matice $\mathbf{H}^{-1}$ aproximaci na parabolu se stejnou velikostí ve všech hlavních osách, pro níž je právě gradient shodný s Newtonovým směrem (transformuji nesymetrické osy $\mathbf{x}\mathbf{A}\mathbf{x}^T$ na symetrické $\mathbf{x}\mathbf{x}^T$, pro níž je má gradient přímo směr k optimální hodnotě).



Obrázek 1.4: Newtonův směr.

Hledáme-li nejmenší hodnotu pro kvadratickou funkci $\mathbf{xAx}^T$ charakterizovanou $\mathbf{A} = \begin{pmatrix} 1 & 1 \\ 1 & 3 \end{pmatrix}$. Červenou barvou je vyznačen Newtonův směr a čím světlejší černá, tím menší hodnota.



Obrázek 1.5: Newtonův směr i záporný gradient

Hledáme-li nejmenší hodnotu pro kvadratickou funkci $\mathbf{xx}^T$ má záporný gradient a Newton stejný směr k minimu. Červenou barvou je vyznačen Newtonův směr a čím světlejší černá, tím menší hodnota.

Povinnost existence druhé derivace metodě výrazně omezuje použití, neboť ne vždy je funkce spojitá. Další věcí je samotný výpočet inverzního Hessiánu, který může být výpočetně náročný.

1.3.2 Matice

V následující tabulce vyjádřím několik podstatných vlastností, které budu používat při úpravách matic nebo se na ně budu odkazovat. Symbolem $\mathbf{A}$ budu označovat libovolnou reálnou čtvercovou matici, $\mathbf{B}$ matici vlastních vektorů k odpovídací matici, $\mathbf{D}$ diagonální matici, případě diagonální matici vlastních čísel odpovídající matice a je $\mathbf{I}$ maticí identity. Takto označených symbolů se budu držet i ve zbytku mé práce, pokud neuvedu něco jiného.

Název	Značení	Podstatná vlastnost
Diagonální matice	$diag\,(d_1, \ldots, d_n)$	Platí $\mathbf{D}^T = \mathbf{D}$, báze je $\mathbf{I}$ Vlastní vektory $\mathbf{B} = \mathbf{I}$ $\mathbf{AD} = \mathbf{DA}^T$
Symetrická matice		$\mathbf{A}^T = \mathbf{A}$
Ortonormální báze		$\mathbf{B} = \mathbf{B}^{-1}$ $\mathbf{BB}^T = \mathbf{I}$
Pozitivně definitní matice		$\forall \mathbf{x} \in \mathbf{R}^n\ \mathbf{xAx}^T > 0$ Vlastní čísla jsou > 0
Pozitivně semidefinitní matice		$\forall \mathbf{x} \in \mathbf{R}^n\ \mathbf{xAx}^T \geq 0$ Vlastní čísla jsou ≥ 0

Předchozí vlastnosti vychází z [28] a [27].

Definice Hessova matice: Matice $\mathbf{H}$ je Hessián (Hessova matice) typu (n, n) parciálních derivací skalární funkce $f\,(x_1, x_2, x_3, \ldots x_n)$ je li funkce dvakrát spojitě derivovatelná podle každé ze svých argumentů

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 x_n} \\ \frac{\partial^2 f}{\partial x_2 x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n x_1} & \frac{\partial^2 f}{\partial x_n x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix} \tag{1.9}$$

Definuji podmínku symetrie Hessovy matice, která vychází ze Schwarzovy věty (upravená definice podle [24]):

Definice Schwarzova věta: Jsou-li parciální derivace funkce f podle argumentů x_i a x_j spojité a derivovatelné na otevřené množině je parciální derivace totožná pro $\frac{\partial^2 f}{\partial x_i x_j} = \frac{\partial^2 f}{\partial x_j x_i}$

Jestliže funkce f splňuje Schwarzovu větu, je Hessova matice symetrická. Nyní ještě definuji kovarianční matici:

Definice Kovarianční matice: Matice $\mathbf{C}$ je kovarianční matice typu (n, n), kde na každém prvku i, j matice je kovariance náhodného vektoru i-tého a j-tého jevu. Symbolem $C(X_1, X_2)$ označuji kovarianci náhodných jevů X_1 a X_2 a $D(X)$ varianci jevu [31]:

$$\mathbf{C} = \begin{pmatrix} D(X_1) & C(X_1, X_2) & \cdots & C(X_1, X_n) \\ C(X_2, X_1) & D(X_2) & \cdots & C(X_2, X_n) \\ \vdots & \vdots & \ddots & \vdots \\ C(X_n, X_1) & C(X_n, X_2) & \cdots & D(X_n) \end{pmatrix} \tag{1.10}$$

Kovarianční matice udává tvar a směr funkce hustoty vícerozměrného normálního rozdělení $\mathcal{N}$, o níž se zmíním více v kapitole o normálním rozdělení. Jelikož platí $C(X_1, X_2) = C(X_2, X_1)$, je matice symetrická a pozitivně semidefinitní.

1.3.3 Vlastní čísla a vlastní vektory matic

Vlastnosti vlastních čísel jsou pro algoritmus CMA-ES podstatné, proto je zde definuji a vysvětlím jejich vlastnosti, které úzce souvisí s algoritmem. Pro zjednodušení budu předpokládat, že jsou matice reálné.

1.3.3.1 Definice vlastních čísel a vlastních vektorů

Definice Vlastní čísla a vlastní vektory: Nechť $\mathbf{C}$ je čtvercová pozitivně definitní matice typu (n, n). Číslo $d^2 \in \mathbf{R}$ se nazývá vlastním číslem matice $\mathbf{A}$, pokud existuje nenulový vektor $\mathbf{b} \in \mathbf{R}^n$ takový, že:

$$\mathbf{C}\mathbf{b} = d^2\mathbf{b} \tag{1.11}$$

Vektor $\mathbf{b}$, který splňuje uvedenou rovnost, se nazývá vlastní vektor matice $\mathbf{A}$ příslušný vlastnímu číslu d^2. [27]

Je-li matice $\mathbf{C}$ pozitivně definitní, jsou všechna vlastní čísla větší než 0.

Pro zjednodušení budu používat maticové značení, kde matice $\mathbf{B} = \left[\mathbf{b}_1^T, \ldots, \mathbf{b}_n^T\right]$ představuje matici vlastních vektorů a každý sloupec představuje vlastní vektor, seřazený v klesající posloupnosti podle jejich odpovídajících vlastních čísel. Vlastní čísla budu značit d_i^2 a pro množinu vlastních čísel budu používat diagonální matici $\mathbf{D}^2 = diag\left(d_1^2, \ldots, d_n^2\right)$, kde odpovídající pozice na diagonále odpovídá její velikosti od největšího do nejmenšího vlastního čísla.

Rozvedu vlastnosti vlastních čísel z rovnice 1.11 v definici. Podle [11] předpokládám, že $\mathbf{C}$ je symetrická a pozitivně definitní matice, která má ortonormální bázi tvořenou sloupci s vlastními vektory $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$ s odpovídajícími vlastními čísly $\mathbf{D}^2 = diag\left(d_1^2, \ldots, d_n^2\right)$. Takže matici $\mathbf{C}$ lze vyjádřit jako rozklad podle rovnosti 1.12.

$$\mathbf{C} = \mathbf{B}\mathbf{D}^2\mathbf{B}^T = \mathbf{B}diag\left(d_1^2, \ldots, d_n^2\right)\mathbf{B}^T \tag{1.12}$$

Od této chvíle budu symbolem $\mathbf{D}^2$ označovat diagonální matici vlastních čísel a $\mathbf{B}$ matici vlastních vektorů k odpovídající matici.

Rovnice 1.12 platí, protože vektory v matici $\mathbf{B}$ jsou navzájem ortonormální. Když vynásobím rovnici inverzní maticí pro níž platí $\mathbf{B} = \mathbf{B}^{-1}$ (viz. vlastnosti ortonormálních matic), dostanu rovnost z definice pro vlastní čísla a vlastní vektory. Pomocí rovnosti 1.12 lze nalézt velmi elegantně inverzní matici $\mathbf{C}$.

$$\begin{aligned} \mathbf{C}^{-1} &= \left(\mathbf{B}\mathbf{D}^2\mathbf{B}^T\right)^{-1} \\ &= \mathbf{B}^{-1}\mathbf{D}^{-2}\left(\mathbf{B}^T\right)^{-1} \\ &= \mathbf{B}\mathbf{D}^{-2}\mathbf{B}^T \end{aligned} \tag{1.13}$$

a druhá odmocina z $\mathbf{C}$ je

$$\begin{aligned} \mathbf{C}^{\frac{1}{2}} &= \left(\mathbf{B}\mathbf{D}^2\mathbf{B}^T\right)^{\frac{1}{2}} \\ &= \mathbf{B}\mathbf{D}\mathbf{B}^T \end{aligned} \tag{1.14}$$

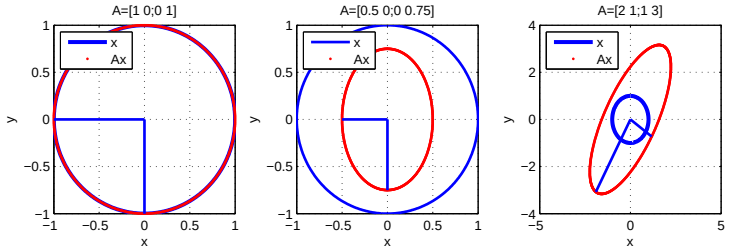
Dokáži tvrzení v 1.14. Hledám takové $\mathbf{C}^{\frac{1}{2}}$, aby platil výraz $\mathbf{C} = \mathbf{C}^{\frac{1}{2}}\left(\mathbf{C}^{\frac{1}{2}}\right)^T$. Použiji definovaný vztah 1.14 a upravím jej s vlastností komutativity násobení diagonálních matic, tedy že platí $\mathbf{B}\mathbf{D} = \mathbf{D}\mathbf{B}^T$.

$$\begin{aligned} \mathbf{C} &= \mathbf{C}^{\frac{1}{2}}\mathbf{C}^{\frac{1}{2}} \\ &= \mathbf{B}\underbrace{\mathbf{D}\mathbf{B}^T}_{\mathbf{B}\mathbf{D}}\underbrace{\mathbf{B}^T\mathbf{D}}_{\mathbf{D}\mathbf{B}}\mathbf{B} = \\ &= \mathbf{B}\underbrace{\mathbf{B}\mathbf{D}^2}_{\mathbf{D}^2\mathbf{B}^T}\mathbf{B}\mathbf{B} = \\ &= \mathbf{B}\mathbf{D}^2\underbrace{\mathbf{B}^T\mathbf{B}}_{\mathbf{I}}\mathbf{B} = \\ &= \mathbf{B}\mathbf{D}^2\mathbf{B}^T \end{aligned} \tag{1.15}$$

Výsledkem násobení $\mathbf{C}^{\frac{1}{2}}\left(\mathbf{C}^{\frac{1}{2}}\right)^T$ je opět kovarianční matice $\mathbf{C} = \mathbf{B}\mathbf{D}^2\mathbf{B}^T$, čímž jsem dokázal platnost tvrzení $\mathbf{C}^{\frac{1}{2}} = \mathbf{B}\mathbf{D}\mathbf{B}^T$. Vztah 1.14 použiji při výpočtu jednoho z parametrů algoritmu CMA-ES, proto ho zde pro úplnost uvádím.

Na obrázku 1.3.3.1 je zobrazená skupina bodů kružnice ($[\sin t, \cos t]$ pro $t = \langle 0, 2\pi \rangle$), která je zobrazena na matici $\mathbf{C}$. Z obrázku je zjevné, že ve směru vlastních vektorů s největším d_i^2 nebo nejmenším d_n^2 vlastním číslem mají hodnoty $\mathbf{x}$ největší nebo nejmenší nárůst. Další vlastností je, že vlastní vektory nejsou rotovány maticí $\mathbf{C}$, což plyne přímo z definice, neboť každý součin vlastního vektoru a jeho odpovídající matice je násobek vlastního vektoru a jeho vlastního čísla.

V případě první matice identity, jejíž vlastní čísla jsou $d_1^2 = d_2^2 = 1$ nedochází k žádné změně, jedná se o identické zobrazení, což odpovídá levému obrázku 1.3.3.1. V případě matice $\mathbf{C} = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.75 \end{pmatrix}$, která má vlastní čísla $d_1^2 = 0.75$ a $d_2^2 = 0.5$ dochází ke zmenšení ve směru odpovídajících vlastních vektorů. A protože je báze (a matice vlastních vektorů) $\mathbf{B} = \mathbf{I}$, jsou vlastní vektory ve směru hlavních os, takže zobrazení změní velikost, ale neotočí kružnici, což odpovídá 1.3.3.1. V posledním případě se celé zobrazení otočí ve směru největšího vlastního vektoru s největším vlastním číslem.



Obrázek 1.6: Geometrická interpretace rozkladu na vlastní čísla a vektory. Modrá kružnice představuje souřadnice jednotkové kružnice a červená zobrazení jednotkové na matici $\mathbf{A}$. *Levý*: Rozklad na vlastní čísla $d_1^2 = 1$ a $d_2^2 = 1$ a vlastní vektory $\mathbf{x}_1\,(1,0)$, $\mathbf{x}_2 = (0,1)$ matice $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, *Prostřední*: Rozklad na vlastní čísla $d_1^1 = 0.75$ a $d_2^2 = 0.5$ a vlastní vektory $\mathbf{x}_1\,(1,0)$, $\mathbf{x}_2 = (0,1)$ matice $\begin{pmatrix} 0.5 & 0 \\ 0 & 0.75 \end{pmatrix}$, *Pravý*: Rozklad na vlastní čísla $d_1^2 \approx 3.6180$ a $d_2^2 \approx 1.3820$ a vlastní vektory $\mathbf{x}_1 \approx (0.5257, 0.8507)$, $\mathbf{x}_2 \approx (-0.8507, 0.5257)$ matice $\begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$

1.3.4 Normální rozdělení

S normálním rozdělením pracuje algoritmus CMA-ES, vysvětlím a rozvedu několik podstatných vlastností.

Náhodný vektor s normálním rozdělením budu značit $\mathcal{N}\,(\mathbf{m}, \mathbf{C})$, abych udržel shodnou notaci s většinou literatury na problematiku adaptace kovariančních matic.

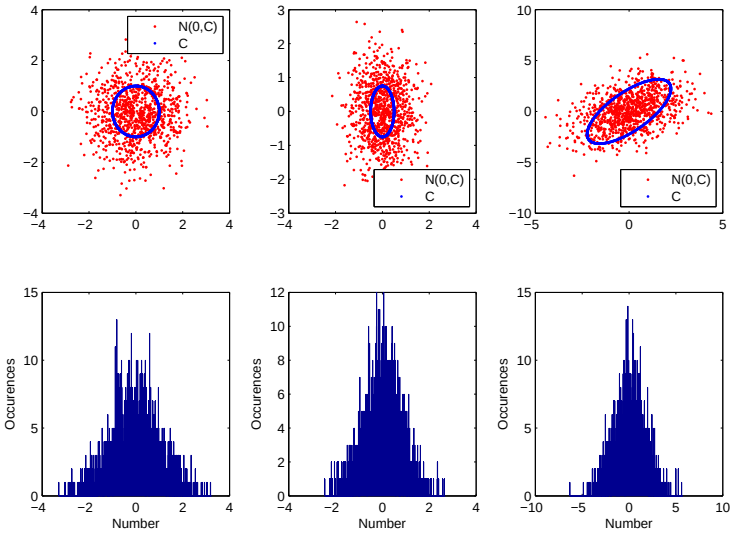
Podle [18] platí pro $\mathcal{N}\,(\mathbf{m}, \mathbf{C})$ 1.16:

$$\begin{aligned} \mathcal{N}\,(\mathbf{m}, \mathbf{C}) &\equiv \mathbf{m} + \mathcal{N}\,(\mathbf{0}, \mathbf{C}) \\ &\equiv \mathbf{m} + \mathbf{C}^{\frac{1}{2}}\mathcal{N}\,(\mathbf{0}, \mathbf{I}) \\ &\equiv \mathbf{m} + \mathbf{B}\mathbf{D}\underbrace{\mathbf{B}^T\mathcal{N}\,(\mathbf{0}, \mathbf{I})}_{\mathcal{N}(\mathbf{0},\mathbf{I})} \\ &\equiv \mathbf{m} + \mathbf{B}\underbrace{\mathbf{D}\mathcal{N}\,(\mathbf{0}, \mathbf{I})}_{\mathcal{N}(\mathbf{0},\mathbf{D}^2)} \\ &\equiv \mathbf{m} + \mathbf{B}\mathcal{N}\,(\mathbf{0}, \mathbf{D}^2) \end{aligned} \tag{1.16}$$

Symbolem $\equiv$ říkám, že vztahy ve výrazu 1.16 jsou definičně ekvivalentní.

Rovnice 1.16 ukazuje v jednotlivých řádcích několik vlastností, které mají vztah s rozkladem na vlastní čísla. Postupně je podrobně uvedu:

1. Předpokládám, že mám normální rozdělení s nulovou střední hodnotou a symetrickou, pozitivně definitní matici $\mathbf{C}$.
2. Kovarianční matice $\mathbf{C}$ je druhá odmocnina zobrazení do normálního rozdělení $\mathcal{N}\,(\mathbf{0}, \mathbf{I})$.
3. Matice $\mathbf{C}^{\frac{1}{2}}$ má odpovídající rozklad $\mathbf{B}\mathbf{D}\mathbf{B}^T$ podle rovnice 1.14.



Obrázek 1.7: Vizualizace normálních rozdělení.
Grafy jsou pro dva rozměry v případě různých kovariančních matic $\mathbf{C}$ s nulovou střední hodnotou. Na dolních grafech jsou odpovídající histogramy k danému rozdělení. *Levý*: Normální rozdělení pro kovarianční matici $\mathbf{C} = \mathbf{I}_2$, *Prostřední*: Normální rozdělení pro symetrickou pozitivně definitní kovarianční matici $\mathbf{C} = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.75 \end{pmatrix}$, *Pravý*: Normální rozdělení pro nesymetrickou pozitivně definitní kovarianční matici $\mathbf{C} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$. Z histogramů je dále vidět, že pravděpodobnost daného jevu klesá se vzdáleností od střední hodnoty $\mathbf{m} = \mathbf{0}$.

4. Protože $\mathbf{B^T}$ je ortonormální, platí pro ni $\mathbf{B}\mathbf{B}^T = \mathbf{B}^T\mathbf{B} = \mathbf{I}$. Pokud přesuneme matici $\mathbf{B}$ do normálního rozdělení (tzn. vynásobím maticí transponovanou), vznikne matice identity ($\mathbf{B}^T\mathcal{N}(\mathbf{0}, \mathbf{I}) = \mathcal{N}\left(\mathbf{0}, \mathbf{B}^T\mathbf{B}\right) = \mathcal{N}(\mathbf{0}, \mathbf{I})$).

5. Přesunu-li $\mathbf{D}$ do pozice kovarianční matice, tak stejně jako se kovarianční matice odmocnila při přesunu z pozice kovariance v normálním rozdělení, tak umocním $\mathbf{D}$ druhou mocninou ($\mathbf{D}\mathcal{N}(\mathbf{0}, \mathbf{I}) = \mathcal{N}\left(\mathbf{0}, \mathbf{D}\mathbf{D}^T\right) = \mathcal{N}(\mathbf{0}, \mathbf{D^2})$).

6. Matice $\mathbf{B}$ pootočí rozdělení podle vlastních vektorů kovarianční matice normálního rozdělení, které bude osově souměrné s hlavními osami o velikosti vlastních čísel (neboť $\mathbf{D} = diag\left(d_1^2, \ldots d_n^2\right)$).

Kapitola 2

Analýza algoritmu CMA-ES

Algoritmus CMA-ES je stochastická ES, pro optimalizaci nelineárních a nekonvexních funkcí. V této ES je populace navzorkovaná podle mnoharozměrného normálního rozdělení $\mathcal{N}(\mathbf{m}, \mathbf{C})$. Normální rozdělení určuje kovarianční matice $\mathbf{C}$ a střední hodnota $\mathbf{m}$. Tyto proměnné jsou podle kvalitní populace adaptovány. Vzájemné vlastnosti jsou modelované kovarianční maticí $\mathbf{C}$ [30].

Algoritmus prokázal velmi dobré výsledky na většině vícerozměrných funkcí oproti svým velmi silným konkurentům a má velkou podporu v komunitě která se zabývá se optimalizací i přes to, že se jedná o poměrně nový algoritmus. Více k experimentům je v [7].

Snažil jsem se, aby většina symbolů, jimiž označuji proměnné algoritmu, byly shodné s článkem [11], ze kterého jsem algoritmus zkoumal. Pro zlepšení čitelnosti popisu algoritmu použiji substituci $\mathbf{y}_{i:\lambda} = \frac{\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}$. Obecný přepis CMA-ES do obecného kódu který je schopen pracovat s maticemi je v algoritmu 2.

2.1 Podrobný popis kroků algoritmu

2.1.1 Generování

V každém krok algoritmu se vygeneruje populace podle normálního rozdělení o velikosti λ podle kovarianční matice $\mathbf{C}^{(g)}$ a vážené střední hodnoty $\mathbf{m}^{(g)}$.

$$\mathbf{x}_k^{(g+1)} \approx \sigma^{(g)} \mathcal{N}\left(\mathbf{m}^{(g)}, \mathbf{C}^{(g)}\right) \approx \mathbf{m}^{(g)} + \sigma^{(g)} \mathcal{N}\left(\mathbf{0}, \mathbf{C}^{(g)}\right) \quad \forall k = 1 \dots \lambda \tag{2.1}$$

Symbolem $\approx$ značím generování náhodných vektorů (a tedy i nové populace) podle odpovídajícího normálního rozdělení $\mathcal{N}\left(\mathbf{m}^{(g)}, \mathbf{C}^{(g)}\right)$. V případě, že se jedná o první krok, populace se vygeneruje podle kovarianční matice tak, aby bylo normální rozdělení jednotkové a násobené $\sigma^{(0)} = 0.5$. Tento případ odpovídá levému obrázku 1.3.3.1.

V dalších generacích může nastat několik situací v závislosti na kovarianční matici $\mathbf{C}^{(g)}$. Rozložím-li $\mathbf{C}^g = \mathbf{B}^{(g)}\mathbf{D}^2\mathbf{B}^{(g)}$ pak:

- Je-li $\mathbf{C}^{(g)} = \mathbf{I}$ tedy $\mathbf{B}^{(g)} = \mathbf{B}^{(g)^T} = \mathbf{D}^2 = \mathbf{I}$, pak normální rozdělení může odpovídat levému obrázku 1.16. Rozdělení má stejnou pravděpodobnost ve všech směrech se stejnou vzdáleností od střední hodnoty. Tomuto stavu odpovídá počáteční stav.

Inicializace:
$\mathbf{m} = rand\,()^n$
$\sigma = \frac{1}{2}$
$\lambda = 4 + \lfloor \log(3 \log N) \rfloor$
$\mu = \lfloor \frac{\lambda}{2} \rfloor$
$\forall i = 1 \ldots \mu : w_i = \frac{\ln\left(\frac{\lambda}{2}+0.5\right) - \ln i}{\sum_{j=1}^{\mu} \ln\left(\frac{\lambda}{2}+0.5\right) - \ln j}$
$\mu_{eff} = \frac{\sum_{i=1}^{N} w_i}{\sum_{i=1}^{N} w_i^2}$
$(c_c, c_\sigma, c_1, c_\mu) = \left(\frac{4 + \frac{\mu_{eff}}{N}}{N + 4 + 2\frac{\mu_{eff}}{N}}, \frac{\mu_{eff}+2}{N+\mu_{eff}+5}, \frac{2}{(n+1.3)^2+\mu_{eff}}, \frac{2\left(\mu_{eff}-2+\frac{1}{2}\right)}{(N+2)^2+\mu_{eff}} \right)$
$d_\sigma = 1 + 2\max\left(0, \sqrt{\frac{\mu_{eff}-1}{n+1}} - 1\right) + c_\sigma$
$\mathbf{p}_c = \mathbf{p}_\sigma\,(0, \ldots 0)^T$
$\mathbf{C}^{\frac{1}{2}} = \mathbf{C} = \mathbf{B} = diag\,(1, 1, \ldots 1)$
$\mathbf{D} = diag\,(1, \ldots, 1)$
$eigeneval = 0$
while ***not*** *zastavovací kritérium splněno* **do**
  Generování λ jedinců normálně rozdělených $\mathcal{N}(\mathbf{m}, \mathbf{C})$
  Vyhodnocení fitness každého jedince $f(\mathbf{x}_i)\ \forall 1 \leq i \leq \lambda$
  Selekce a rekombinace z μ jedinců $\mathbf{m}^{(g+1)} = \sum_{i=1}^{\mu} w_i x_{i:\lambda} s$
  Výpočet $\mathbf{p}_\sigma^{(g+1)} = (1 - c_\sigma)\,\mathbf{p}_\sigma^{(g)} + \sqrt{c_\sigma\,(2 - c_\sigma)\,\mu_{eff}}\mathbf{C}^{(g)^{-\frac{1}{2}}} \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}$
  $\mathbf{p}_c^{(g+1)} = (1 - c_c)\,\mathbf{p}_c^{(g)} + \sqrt{c_c\,(2 - c_c)\,\mu_{eff}} \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}$
  $\mathbf{C}^{(g+1)} = (1 - c_1 - c_\mu)\,\mathbf{C}^{(g)} + \underbrace{c_1 \mathbf{p}_c^{(g+1)} \mathbf{p}_c^{(g+1)^T}}_{rank-1-update} + c_\mu \underbrace{\sum_{i=1}^{\mu} w_i \mathbf{y}_{i:\lambda}^{(g+1)} \mathbf{y}_{i:\lambda}^{(g+1)^T}}_{rank-\mu-update}$
  $\sigma^{(g+1)} = \sigma^{(g)} e^{\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|p_\sigma^{(g+1)}\|}{E\|\mathcal{N}(\mathbf{0},\mathbf{I})\|} - 1\right)\right)}$
  $g \leftarrow g + 1$
end

Algorithm 2: Základní algoritmus $(\mu/\mu_W, \lambda)$ CMA-ES

- Je-li $\mathbf{B}^{(g)} = \mathbf{B}^{(g)^T} = \mathbf{I}$, pak normální rozdělení může odpovídat prostřednímu obrázku 1.16. Rozdělení má větší pravděpodobnost ve směru té hlavní osy, která má největší vlastní číslo.
- Je-li $\mathbf{B}^{(g)} \neq \mathbf{I}$, pak rozdělení může odpovídat pravému obrázku 1.16. Rozdělení má větší pravděpodobnost vzniku ve směru vlastních vektorů $\mathbf{b}_i^{(g)}$ s největším vlastním číslem d_{ii}^2.

2.1.2 Vyhodnocení

Vyhodnotí se kvalita všech λ jedinců nové populace, tak abychom měli setříděnou populaci. Takže máme setříděnou posloupnost $\mathbf{x}_{i:\lambda}^{(g)}$ takových jedinců, že platí $f\left(\mathbf{x}_{1:\lambda}^{(g)}\right) \leq f\left(\mathbf{x}_{2:\lambda}^{(g)}\right) \leq \ldots f\left(\mathbf{x}_{\lambda:\lambda}^{(g)}\right)$, hledám-li minimum fitness funkce f. Naopak hledáme-li maximum tak máme posloupnost $\mathbf{x}_{i:\lambda}^{(g)}$ jedinců takových, že platí $f\left(\mathbf{x}_{1:\lambda}^{(g)}\right) \geq f\left(\mathbf{x}_{2:\lambda}^{(g)}\right) \geq \ldots f\left(\mathbf{x}_{\lambda:\lambda}^{(g)}\right)$.

2.1.3 Selekce a rekombinace

Ze setříděné populace vybereme μ nejlepších a z nich pak spočítáme váženou střední hodnotu.

$$\mathbf{m}^{(g+1)} = \sum_{i=1}^{\mu} w_i \mathbf{x}_{i:\lambda}^{(g+1)} \tag{2.2}$$

Hodnotu vah je doporučováno volit tak, aby největší váhu měli nejlepší jedinci a postupně se snižovala. Zároveň by měl její součet být 1. Jedna z použitých funkcí pro výpočet vektoru vah je např. $w_i = \frac{\ln\left(\frac{\lambda}{2}+0.5\right)-\ln i}{\sum_{j=1}^{\mu} \ln\left(\frac{\lambda}{2}+0.5\right)-\ln j}$. Na obrázku 2.1 je graf vah pro sto-dimenzionální případ (μ je v tomto případě 8).

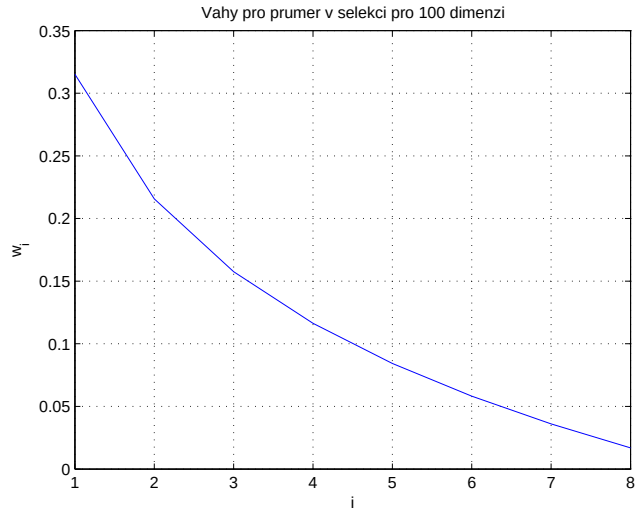
Vážená střední hodnota $\mathbf{m}^{(g+1)}$ z μ jedinců udává, kde bude největší pravděpodobnost výskytu jedinců v následující generaci. Výpočet $\mathbf{m}^{(g+1)}$ reprezentuje selekci.

2.1.4 Výpočet $\mathbf{p}_\sigma$

Hodnota $\mathbf{p}_\sigma$ slouží k výpočtu hodnoty σ, tedy velikost kroku algoritmu (tzv. step-size parametru). Při výpočtu kovarianční matice $\mathbf{C}$ se proměnná $\mathbf{p}_\sigma$ nevyskytuje, ale ovlivňuje hodnotu σ. Ta udává velikost násobku normálně vygenerovaných hodnot. Parametrem σ ovlivňujeme pouze velikost kroku (normálního rozdělení) protože parametry $\mathbf{C}$ a $\mathbf{m}$ neudávají velikost, ale pouze pozici a směr. Rovnice 2.3 počítá novou hodnotu $\mathbf{p}_\sigma^{(g+1)}$:

$$\mathbf{p}_\sigma^{(g+1)} = (1-c_\sigma)\,\mathbf{p}_\sigma^{(g)} + \sqrt{c_\sigma + (2-c_\sigma)\,\mu_{eff}}\,\mathbf{C}^{(g)^{-\frac{1}{2}}} \underbrace{\frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}}_{evolution\ path} \tag{2.3}$$

Na začátku je hodnota $\mathbf{p}_\sigma^{(0)}$ nulová. V každé další generaci se mění podle velikosti vzdálenosti hodnot mezi současnou generací $\mathbf{m}^{(g+1)}$ a předchozí $\mathbf{m}^{(g)}$. Hodnotu $\mathbf{C}^{(g)^{-1\frac{1}{2}}}$ lze přepsat dále podle 1.14 na $\mathbf{B}^{(g)}\mathbf{D}^{(g)^{-1}}\mathbf{B}^{(g)^T}$. Transformace provádí postupným násobením maticemi rozkladu následující kroky:



Obrázek 2.1: Váhy w_i pro vážení jedinců při počítání $\mathbf{m}$. Na horizontální ose je pořadí vah, přičemž první je váha pro nejsilnějšího jedince ($f(\mathbf{x}_{1:\lambda})$). Poslední pro nejslabšího jedince ($f(\mathbf{x}_{\mu:\lambda})$), který prošel selekcí.

- $\mathbf{B}^{(g)^T}$ otočí $\frac{\mathbf{m}^{(g+1)}-\mathbf{m}^{(g)}}{\sigma^{(g)}}$ podle $\mathbf{B}^{(g)}$
- $\mathbf{D}^{(g)^{-1}}$ změní velikosti v hlavních osách souřadného systému tak, aby byly stejně velké.
- $\mathbf{B}^{(g)}$ otočí $\frac{\mathbf{m}^{(g+1)}-\mathbf{m}^{(g)}}{\sigma^{(g)}}$ do původního souřadného systému (protože platí $\mathbf{B}\mathbf{B}^T = \mathbf{I}$).

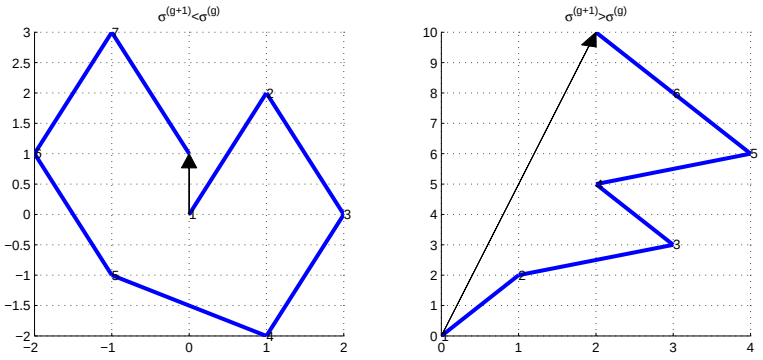
Násobením $\mathbf{C}^{-\frac{1}{2}}$ dosáhneme toho, že hodnoty $\mathbf{p}_\sigma$ budou nezávislé na velikosti kovarianční matice a to díky násobení $\mathbf{D}^{-1}$.

Vliv $\mathbf{C}^{-\frac{1}{2}}$ ukazuje obrázek 2.3, kde se hodnota $\mathbf{p}_\sigma^{(g)}$ nemění s poklesem $\mathcal{L}_2$ normy kovarianční matice $\|\mathbf{C}^{(g)}\|_2 = \sqrt{\max \mathbf{D}}$ (norma je podle [33]).

Vysvětlím, co znamená evoluční cesta [18] (překlad z angl. *evolution path*). V průběhu výpočtu algoritmu se přesouvá postupně střední hodnota z $\mathbf{m}^{(g)}$ do $\mathbf{m}^{(g+1)}$, která přibližně udává, jakou cestu algoritmus v průběhu generací prošel. A pokud se hodnoty $\mathbf{m}^{(g)}$ a $\mathbf{m}^{(g+1)}$ pohybují v generacích přibližně v okolí stejné hodnoty, tak se vyrušují. To říká, že algoritmus se pohybuje kolem nějakého bodu a je vhodné zmenšit hodnotu $\sigma^{(g+1)} < \sigma^{(g)}$ v další generaci. Příklad takové evoluční cesty je na obrázku 2.2 nalevo. Druhý opačný případ, který může nastat je, že cesty mají přibližně stejný směr, takže se nevyrušují a je naopak vhodné zvětšit hodnotu $\sigma^{(g+1)} > \sigma^{(g)}$, viz. obrázek 2.2 napravo.

Dělení $\sigma^{(g)}$ rozdílu $\frac{\mathbf{m}^{(g+1)}-\mathbf{m}^{(g)}}{\sigma^{(g)}}$ je použito při generování populace jako násobek normálního rozdělení z výrazu 2.1, takže hodnotu rozdílů normalizujeme.

Pro uchování vztahu s hodnotami $\mathbf{p}_\sigma$ v předchozích generacích je použito tzv. exponenciální vyhlazování (překlad z angl. *exponential smoothing*) s předchozími hodnotami. Za předpokladu $\mu_{eff} = 1$ (součet všech vah v selekci dává 1) je hodnota $(1-c_\sigma)^2 + \sqrt{c_\sigma(2-c_\sigma)}^2 = 1$.



Obrázek 2.2: Ukázky dvou evolučních cest

Na levém obrázku je vidět, že se střední hodnoty $\mathbf{m}^{(g+1}$ a $\mathbf{m}^{(g)}$ pohybují v svém okolí a navzájem se vyrušují, takže velikost $\sigma^{(g+1)}$ by měla být zmenšena, čímž se zmenší i šířka normálního rozdělení při generování nové populace. Zatímco na pravém obrázku mají střední hodnoty $\mathbf{m}^{(g+1)}$ a $\mathbf{m}^{(g)}$ podobný směr a navzájem se nevyrušují, takže velikost $\sigma^{(g+1)}$ by měla být zvětšena.

Pro $c_\sigma = 1$ si neuchovává žádnou historii a pro $c_\sigma < 1$ přičítá historii úměrně se zmenšováním c_σ. Pro $c_\sigma = 0$ bere v potaz pouze historii, což je nepřípustná hodnota.

Volba konstanty c_σ je doporučena jako $c_\sigma = \frac{\mu_{eff}+2}{N+\mu_{eff}+5}$ [18].

Je dokázáno, že hodnota $\mathbf{p}_\sigma^{(g+1)}$ má za předpokladu náhodné selekce hodnot $\mathbf{m}^{(g)}$ a $\mathbf{m}^{(g+1)}$ rozdělení jako $\mathcal{N}(\mathbf{0}, \mathbf{I})$ [18].

2.1.5 Výpočet $\mathbf{p}_c$

Obdobně jako $\mathbf{p}_\sigma^{(0)}$ platí, že na začátku je hodnota $\mathbf{p}_c^{(0)} = \mathbf{0}$. Hodnota $\mathbf{p}_c$ slouží k tzv. *rank-1-update* výpočtu. Výpočet se nazývá *rank-1-update*, protože součin $\mathbf{p}_c^{(g+1)}\mathbf{p}_c^{(g+1)^T}$ má hodnost 1.

$$\mathbf{p}_c^{(g+1)} = (1 - c_c)\,\mathbf{p}_c^{(g)} + \sqrt{c_c\,(2 - c_c)\,\mu_{eff}}\,\underbrace{\frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}}_{evolution\ path} \tag{2.4}$$

Ve výrazu 2.4 pro výpočet $\mathbf{p}_c^{(g+1)}$ zlomek $\frac{\mathbf{m}^{(g+1)}-\mathbf{m}^{(g)}}{\sigma^{(g)}}$ udává jeden ze směrů, v jakém se má nová kovarianční matice směrovat, resp. v jakém směru zvýšit pravděpodobnost výskytu (přičtení $\mathbf{p}_c^{(g+1)}\mathbf{p}_c^{(g+1)^T}$ přidá nový vlastní vektor ke kovarianční matici).

Rozdíl vzdáleností středních hodnot $\frac{\mathbf{m}^{(g+1)}-\mathbf{m}^{(g)}}{\sigma^{(g)}}$ přidává setrvačnost cesty ke kovarianční matici.Pomocí součinu $\mathbf{p}_c^{(g+1)}\mathbf{p}_c^{(g+1)^T}$ změní tvar (a směr) $\mathbf{C}^{(g+1)}$ tak, aby se ve směru rozdílu zvýšila pravděpodobnost těch hodnot, kam se nová střední hodnota $\mathbf{m}^{(g+1)}$ posunula.

Je dokázáno, že velikost vektoru $\mathbf{p}_c$ má, za předpokladu $c_c = 1$ a $\mu_{eff} = 1$ stejné rozdělení, jako $\mathcal{N}(\mathbf{0}, \mathbf{C})$ [18].

Na obrázku 2.3 je vidět rozdíl vlivu normy kovarianční matice $\|\mathbf{C}^{(g)}\|_2$ na hodnoty $\|\mathbf{p}_c^{(g)}\|$ a $\|\mathbf{p}_\sigma^{(g)}\|$. Zatímco pokles normy $\|\mathbf{C}^{(g)}\|$ ovlivní velikost $\|\mathbf{p}_c^{(g)}\|$, tak hodnota $\|\mathbf{p}_\sigma^{g}\|$ je na velikosti kovarianční matice nezávislá.

2.1.6 Výpočet kovarianční matice C

Výsledek výpočtu kovarianční matice $\mathbf{C}$ udává směry větších hustot pravděpodobností normálního rozdělení. $\mathbf{C}$ určuje směr kam se zvětší pravděpodobnost výskytu další generace.

Velikost normálního rozdělení je určována velikostí vlastních čísel $\mathbf{D}^2$ a směr vlastními vektory matice $\mathbf{C}^{(g)}$. Podle matice vlastních čísel $\mathbf{D}^2$ se transformuje funkce hustoty pravděpodobnosti podle odpovídajících velikostí v hlavních osách a následně se rozdělení otočí podle matice vlastních vektorů $\mathbf{B}$ (zobrazení na bázi $\mathbf{B}$). Z toho co jsem řek plyne, že nová populace bude mít zvýšenou pravděpodobnost ve směrech vlastních vektorů s většími vlastními čísly. Pro lepší čitelnost výrazu 2.5 použiji substituci $\mathbf{y}_{i:\lambda}^{(g+1)} = \frac{\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}$.

$$\mathbf{C}^{(g+1)} = (1 - c_1 - c_\mu)\,\mathbf{C}^{(g)} + \underbrace{c_1 \mathbf{p}_c^{(g+1)} \mathbf{p}_c^{(g+1)^T}}_{rank-1-update} + c_\mu \underbrace{\sum_{i=1}^{\mu} w_i \mathbf{y}_{1:\lambda}^{(g+1)} \mathbf{y}_{i:\lambda}^{(g+1)^{(g+1)}}}_{rank-\mu-update} \tag{2.5}$$

Výpočet $\mathbf{p}_c$ označený jako *rank-1-update* jsem popsal v předchozím odstavci, nyní vysvětlím interpretaci druhého členu *rank-μ-update* ve výrazu 2.5.

Obdobně jako v případě *rank-1-update*, kde *1* udává, že hodnost této operace je 1, tak *rank-μ-update* udává hodnost $\min(\mu, n)$ váženého součtu součinů výsledku z 2.5.

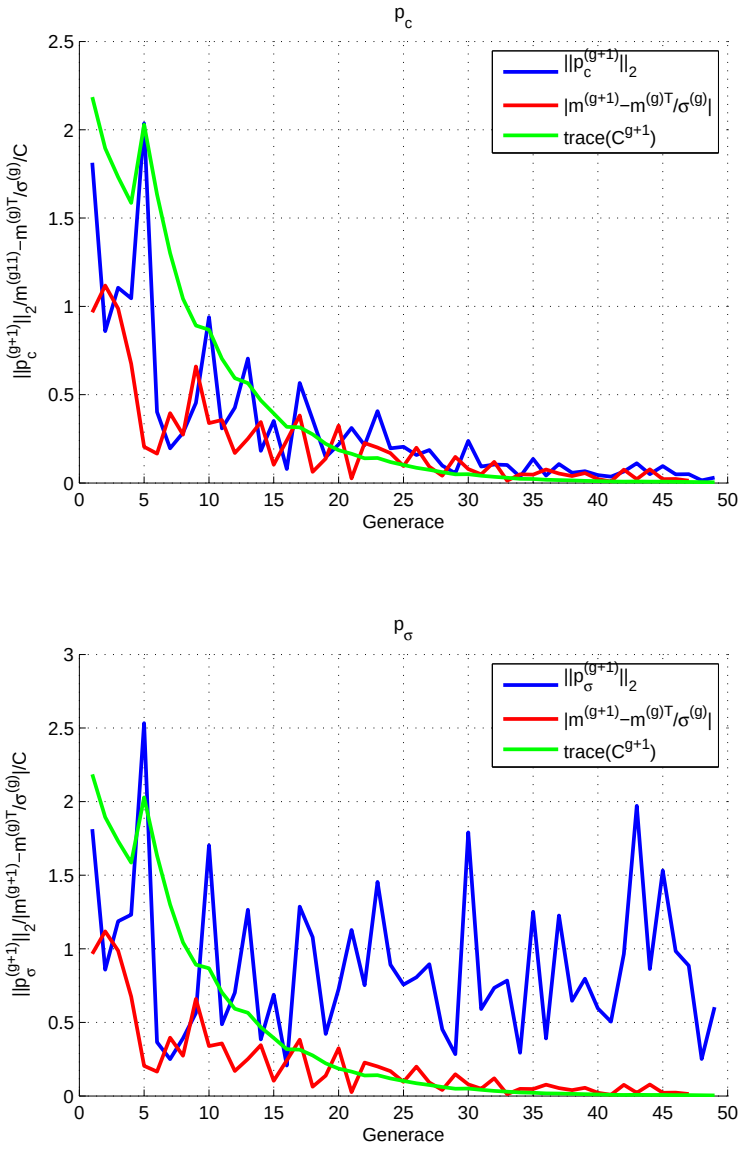
Odvodím rovnici 2.5. Budu předpokládat, že kovarianční matici počítám pouze z aktuální populace podle vzorce 2.6:

$$\mathbf{C}_\mu^{(g+1)} = \sum_{i=1}^{\mu} w_i \left(\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}\right)\left(\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}\right)^T \tag{2.6}$$

Pokud mám dostatečně velkou populaci vybraných jedinců, tak pomocí rovnice 2.6 mohu odhadovat hodnotu kovarianční matice $\mathbf{C}$. Tento odhad je na obrázku 2.4, kde populace je tvořena $\lambda = 300$ jedinci, z nichž je vybráno $\mu = 100$ jedinců podle jejich fitness funkce f. Vzniká ale problém s historií, neboť v případě rovnice 2.6 zahazuji v každé nové generaci předchozí hodnoty $\mathbf{C}^{(g)}$, což může způsobit příliš rychlou konvergenci k lokálnímu optimu, nebo naopak příliš velkou divergenci (slepé prohledávání stavového prostoru, ztráta pozitivní definitnosti $\mathbf{C}$), proto ve vzorci 2.6 zohledním historii, přičemž větší váhu bude mít ta nedávná.

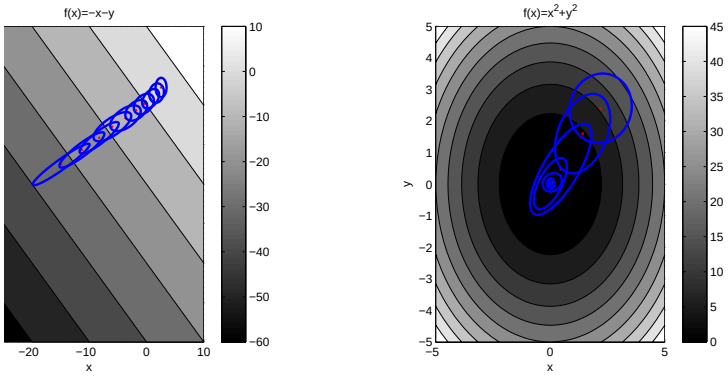
$$\begin{aligned} \mathbf{C}^{(g+1)} &= (1 - c_\mu)\,\mathbf{C}^{(g)} + c_\mu \tfrac{1}{\sigma^{(g)^2}} \mathbf{C}_\mu^{(g+1)} \\ &= (1 - c_\mu)\,\mathbf{C}^{(g)} + \underbrace{c_\mu \sum_{i=1}^{\mu} w_i \mathbf{y}_{i:\lambda}^{(g+1)} \mathbf{y}_{i:\lambda}^{(g+1)^T}}_{rank-\mu-update} \end{aligned} \tag{2.7}$$

Přidáme-li k výpočtu 2.7 výpočet *rank-μ-update* vznikne nám rovnice pro výpočet 2.5.



Obrázek 2.3: Vliv rozdílu středních hodnot

Vliv $\sum_{i=1}^{N} \frac{\mathbf{m}^{(g+1)}-\mathbf{m}^{(g)}}{\sigma}$ a $\|\mathbf{C}^{(g)}\|_2$ na hodnotu normy $\|\mathbf{p}_c\|$ a $\|\mathbf{p}_c\|$ pro funkci $f(\mathbf{x}) = \mathbf{x}\mathbf{x}^T$ při ignorování předchozích hodnot $\mathbf{p}_c^{(g)}$ a $\mathbf{p}_\sigma$ (tzn. $c_c = 1$, $c_\sigma = 1$).



Obrázek 2.4: Odhad konvergence algoritmu CMA-ES.
Pokud počítáme kovarianční matici podle vzorce 2.6. Hledáme minima dvou funkcí, které jsou na obrázku charakterizované stupněm šedi. Modrou barvou vizualizuji kovarianční matici v dané generaci a červenou odpovídací střední hodnotu $\mathbf{m}$ *Levý*: Funkce $f(x, y) = -x - y$. *Pravý*: Funkce $f(x + y) = x^2 + y^2$.

Volba c_μ je kritická, neboť v případě malých hodnot může kovarianční matice konvergovat příliš pomalu nebo naopak při velkých hodnotách c_μ může zanedbávat nově vypočtené směry. Empiricky je hodnota doporučená jako $c_\mu \frac{2\left(\mu_{eff}-2+\frac{1}{2}\right)}{(N+2)^2+\mu_{eff}}$.

Vlastnosti *rank-μ-update* přidávají tedy k výpočtu $\mathbf{C}^{(g+1)}$ vlastnosti celé μ populace.

Spojením *rank-1-update* a *rank-μ-update* získáme dobré vlastnosti ze stávající populace s uchováním části informace s historii, z čehož vznikne rovnice 2.5

2.1.7 Výpočet hodnoty σ

Hodnota σ udává šíři rozdělení pro normální rozdělení. Označuje se i step-size parametr. Primární veličina, která ovlivňuje velikost $\sigma^{(g+1)}$ je Eukleidovská norma $\|\mathbf{p}_\sigma^{(g+1)}\|$ v poměru k Eukleidovské normě očekávané hodnoty rozdělení $\|\mathcal{N}(\mathbf{0}, \mathbf{I})\|$. Norma $\|\mathcal{N}(\mathbf{0}, \mathbf{I})\|$ je určena podle [11] jako $\|\mathcal{N}(\mathbf{0}, \mathbf{I})\| = \sqrt{n} + \mathcal{O}\left(\frac{1}{n}\right) = N^{0.5}\left(1 - \frac{1}{(4N)} + \frac{1}{(21N^2)})\right)$, kde N je dimenze fitness funkce f.

- Pro $\|\mathbf{p}_\sigma^{(g+1)}\| = \|\mathcal{N}(\mathbf{0}, \mathbf{I})\|$ bude hodnota $\sigma^{(g+1)} = \sigma^{(g)}$
- Pro $\|\mathbf{p}_\sigma^{(g+1)}\| > \|\mathcal{N}(\mathbf{0}, \mathbf{I})\|$ bude hodnota $\sigma^{(g+1)} > \sigma^{(g)}$
- Pro $\|\mathbf{p}_\sigma^{(g+1)}\| < \|\mathcal{N}(\mathbf{0}, \mathbf{I})\|$ bude hodnota $\sigma^{(g+1)} < \sigma^{(g)}$

Konstanta d_σ je určena empiricky jako $d_\sigma = 1 + 2\max\left(0, \sqrt{\frac{\mu_{eff}-1}{n+1}} - 1\right) + c_\sigma$.

2.1.8 Zastavovací kritéria

Kromě klasických zastavovacích kritérií (někdy také zastavovací podmínky), jako počet vyhodnocení proměnných, počet kroků algoritmu nebo dosažení požadované fitness může algoritmus zastavit i několik jiných kritérií, které vycházejí z jeho proměnných.

Všechna zastavovací kritéria vycházející z vlastností algoritmu podle článku [18] jsem implementoval do knihovny.

Následující podmínky jsou nezávislé na problému:

- **NoEffectAxis** - Zastav jestliže 0.1 směrodatné odchylky v některé z komponent $\mathbf{C}^{(g)}$ přičtené k $\mathbf{m}^{(g)}$ nezmění jeho hodnotu.
- **NoEffectCoord** - Zastav jestliže přičtení 0.2 směrodatné odchylky v nějaké i-té osy nezmění hodnotu $\mathbf{m}^{(g+1)}$ v dané souřadnici $m_i^{(g+1)}$.
- **EqualFunValues** -Zastav jestliže $10+\lceil\frac{30n}{\lambda}\rceil$ generacích je hodnota $\sum_{g=0}^{10+\lceil\frac{30n}{\lambda}\rceil} f\left(\mathbf{x}_{1:\lambda}^{(g)}\right)$ rovna 0.
- **ConditionCov** - Zastav jestliže číslo podmíněnosti kovarianční matice $\mathbf{C}$ je větší než $\kappa = 10^{14}$.

A následující podmínky mají parametr, který se může měnit v závislosti na řešeném problému:

- **TolFun** - Zastav jestliže $10 + \lceil\frac{30n}{\lambda}\rceil$ generacích je hodnota $\sum_{g=0}^{10+\lceil\frac{30n}{\lambda}\rceil} f\left(\mathbf{x}_{1:\lambda}^{(g)}\right)$ menší nebo rovna **TolFun**.
- **TolX** - Zastav jestliže směrodatná odchylka normálního rozdělení je ve všech osách menší než hodnota **TolX** a jestliže hodnota $\sigma^{(g)}\mathbf{p}_c^{(g)}$ je ve všech osách menší než **TolX**.

Zastavovací podmínky **NoEffectAxis**, **NoEffectCoord**, **ConditionCov** a **TolX** přepíši do formy, v níž jsem s proměnnými algoritmu schopen vyhodnotit zda jsou platné. Podrobněji se o zastavovacích podmínkách zmíním v části věnované implementaci zastavovacích podmínek.

2.2 Vlastnosti algoritmu CMA-ES

Algoritmus CMA-ES je vhodný pro optimalizační problémy, kde klasické metody druhého řádu (quasi-Newton metoda, metoda sdružených gradientů) mohou selhat v některých funkcích [18].

2.2.1 Invariance

Mezi nejzajímavější vlastnost patří invariance. V následujícím seznamu vyjmenuji tyto vlastnosti a úpravy k dosažení invariance.

- Invariance proti translaci, kdy posunu celou fitness funkci o $\mathbf{t}$, takže fitness funkce bude $f(\mathbf{x}+\mathbf{t})$. Pokud odpovídajícím způsobem upravím původní počáteční střední hodnotu $\mathbf{m}^{(0)}$ na posunutou střední hodnotu $\mathbf{m}_t^{(0)} = \mathbf{m}^{(0)} + \mathbf{t}$ stane se problém identický.

- Invariance proti pořadí znamená, že pokud algoritmu předám populaci v libovolném pořadí, výsledky zůstanou stejné, protože algoritmus CMA-ES závisí pouze na ohodnocení jedinců, nikoli na pořadí.

- Invariance proti otočení, kdy otočím celou fitness funkci o úhel θ, takže pokud bude $\mathbf{R}$ rotační matice o úhel θ, tak funkce f otočená o úhel θ bude $f\left(\mathbf{R}\mathbf{x}^T\right)$. Pokud odpovídajícím způsobem upravím původní počáteční střední hodnotu $\mathbf{m}^{(0)}$ na novou otočenou střední hodnotu $\mathbf{m}_{\mathbf{R}}^{(0)} = \mathbf{R}\mathbf{m}^{(0)^T}$, zůstane problém identický jako v případě, že mám fitness funkci $f(\mathbf{x})$.

- Invariance proti lineární transformaci, kdy vynásobím celou fitness funkci skalárem c, tak fitness funkce bude $cf(\mathbf{x})$. Pokud odpovídajícím způsobem upravím původní počáteční hodnotu $\mathbf{m}^{(0)}$ na novou vynásobenou střední hodnotu $\mathbf{m}_c^{(0)} = c\mathbf{m}^{(0)}$ a kovarianční matici $\mathbf{C}^{(0)}$ na $\mathbf{C}_c^{(0)} = c\mathbf{C}^{(0)}$, tak zůstane problém identický jako v případě, že mám fitness funkci $f(\mathbf{x})$.

- Invariance proti různým poměrům hlavních os fitness funkce f, kdy každý vektor $\mathbf{x}$ složím se nějakým skalárním součinem $\mathbf{c}$, takže fitness funkce bude $f(\mathbf{c}\cdot\mathbf{x})$. Pokud odpovídajícím způsobem upravím původní počáteční hodnotu $\mathbf{m}^{(0)}$ na novou vynásobenou střední hodnotu $\mathbf{m}_{\mathbf{c}}^{(0)} = \mathbf{c}\mathbf{m}^{(0)}$ a každý prvek $C_i i$ na diagonále kovarianční matice $\mathbf{C}^{(0)}$ vynásobím odpovídající složkou c_i vektoru $\mathbf{c}$, pak zůstane problém identický jako v případě, že mám fitness funkci $f(\mathbf{x})$.

- Invariance proti libovolné lineární transformaci $\mathbf{A}$, která má inverzní matici. Pokud odpovídajícím způsobem upravím původní kovarianční matici $\mathbf{C}_{\mathbf{A}}^{(0)} = \mathbf{A}^{-1}\left(\mathbf{A}^{-1}\right)^T$, stejně jako $\mathbf{m}_{\mathbf{A}}^0 = \mathbf{A}\mathbf{m}^{(0)}$ tak zůstane problém identický jako v případě, že mám fitness funkci $f(\mathbf{x})$.

2.2.2 Stabilita

Za předpokladu, že fitness funkce bude náhodná funkce nezávislá na $\mathbf{x}$ ($f(\mathbf{x})$ =`rand()` náhodná selekce) bude se střední hodnota $\mathbf{m}^{(g)}$ náhodně pohybovat, ale parametry algoritmu $\mathbf{C}^{(g)}$ a $\sigma^{(g)}$ budou konvergovat k hodnotě 0. [18]

2.2.3 Vztah s Hessovou maticí

Algoritmus CMA-ES je úzce svázán s aproximací inverzní Hessovy matice. Algoritmus CMA-ES aproximuje inverzní Hessovu matici $\mathbf{H}^{-1}$, která je optimální pro problémy, jejíž tvar funkce je vhodně aproximovatelný Taylorovým polynomem druhého stupně. V tomto případě dosahuje podobných výsledků, jako již zmíněná Newtonova metoda.

Budu předpokládat, že hledám minimum funkce $f(\mathbf{x}) = \mathbf{x}\mathbf{x}^T$. Podle [4] je matice identity optimální kovarianční matice pro hledání optima funkce f a konverguje v $(1,\lambda)$-ES. Dále

hledám minimum funkce která má tvar paraboly $f_E(\mathbf{x}) = \mathbf{x}\mathbf{H}\mathbf{x}^T$ a budu předpokládat, že $\mathbf{H}$ je symetrická a pozitivně definitní matice.

Rozložím-li matici $\mathbf{H} = \mathbf{B}\mathbf{D}^2\mathbf{B}^T$ na vlastní čísla a vlastní vektory podle 1.12, budu mít ortonormální matici vlastních vektorů $\mathbf{B}$ (tzn. splňující $\mathbf{B} = \mathbf{B}^{-1}$) a diagonální matici vlastních čísel $\mathbf{D}^2$.

Dále pak před dosazením hodnot budu do f_E transformovat hodnoty $\mathbf{x}$ jako

$$\mathbf{x}_E = \mathbf{D}^{-1}\mathbf{B}\mathbf{x} \tag{2.8}$$

Výrazem 2.8 nejdříve otočím hodnoty $\mathbf{x}$ ve směru $\mathbf{B}$ a následně velikosti hlavních os zmenším tak, že si budou všechny rovny, takže bude platit rovnost 2.9.

$$f_E(\mathbf{x}) = f(\mathbf{x}_E) \tag{2.9}$$

Tvrzení mohu dokázat 2.9. Dosadím-li do f_E místo $\mathbf{x}$ transformaci $\mathbf{x}_E$ a upravím podle 2.10.

$$\begin{aligned}
f_E(\mathbf{x}_E) &= \mathbf{x}_E\mathbf{H}\mathbf{x}_E^T \\
&= \underbrace{\mathbf{D}^{-1}\mathbf{B}\mathbf{x}}_{\mathbf{x}_E}\underbrace{\mathbf{B}\mathbf{D}^2\mathbf{B}^T}_{\mathbf{H}}\underbrace{\mathbf{D}^{-1}\mathbf{B}^T\mathbf{x}^T}_{\mathbf{x}^T} \\
&= \mathbf{D}^{-1}\mathbf{B}\mathbf{x}\mathbf{B}\mathbf{D}^2\underbrace{\mathbf{B}^T\mathbf{D}^{-1}}_{\mathbf{D}^{-1}\mathbf{B}}\mathbf{B}^T\mathbf{x}^T \\
&= \underbrace{\mathbf{D}^{-1}\mathbf{B}\mathbf{x}}_{\mathbf{x}\mathbf{D}^{-1}\mathbf{B}^T}\mathbf{B}\mathbf{D}^2\mathbf{D}^{-1}\underbrace{\mathbf{B}\mathbf{B}^T}_{\mathbf{I}}\mathbf{x}^T \\
&= \mathbf{x}\mathbf{D}^{-1}\underbrace{\mathbf{B}^T\mathbf{B}}_{\mathbf{I}}\underbrace{\mathbf{D}^2\mathbf{D}^{-1}}_{\mathbf{D}}\mathbf{x}^T \\
&= \mathbf{x}\underbrace{\mathbf{D}^{-1}\mathbf{D}}_{\mathbf{I}}\mathbf{x}^T \\
&= \mathbf{x}\mathbf{x}^T
\end{aligned} \tag{2.10}$$

Budu-li vycházet z předpokladu, že pro $\mathbf{x}\mathbf{x}^T$ je ideální rozdělení $\mathcal{N}(\mathbf{0}, \mathbf{I})$, tak pro $\mathbf{x}\mathbf{H}\mathbf{x}^T$ bude ideální rozdělení $\mathcal{N}(\mathbf{0}, \mathbf{H}^{-1})$.

Uvedená transformace $\mathbf{x}_E$ provádí to samé, co Newtonova metoda, která aproximovanou funkci nahradí polynomem druhého stupně, tedy kvadratickou funkcí. Newtonova metoda transformuje funkci na symetrickou parabolu, pro níž je směr gradientu směr přímý k optimu. Algoritmus CMA-ES provádí stejnou transformaci, jako jsem zmínil v předchozím odstavci. Proto se obě metody chovají podobně a CMA-ES aproximuje kovarianční matici $\mathbf{C}$, což je hodnota inverzního Hessovy matice $\mathbf{H}^{-1}$.

2.3 Varianty CMA-ES

K čistému algoritmu CMA-ES existuje řada variant rozšiřujících jeho vlastnosti. Standardní varianta se označuje jako L-CMA-ES, kde počáteční písmeno značí *Local search*, takže od algoritmu očekávám méně robustní prohledávání. Algoritmus L-CMA-ES dosahuje rychlé konvergence poblíž nějakého optima.

Další obecná varianta algoritmu je G-CMA-ES, kde počáteční písmeno značí *Global search*. Algoritmus G-CMA-ES je robustnější při prohledávání ale na úkor rychlosti (a mnohdy i výpočetní náročnosti).

2.3.1 IPOP-CMA-ES

IPOP-CMA-ES [2] spadá do třídy G-CMA-ES. Je to jedna z variant CMA-ES, která nahrazuje klasický algoritmus. Při dosažení určitých podmínek dojde k restartování všech parametrů algoritmu do počátečního stavu. Výsledkem toho může být robustnější prohledávání prostoru, neboť s novým restartem algoritmus znovu inicializuje všechny parametry na počáteční hodnoty s novou náhodnou střední hodnotou. Další charakteristikou IPOP-CMA-ES je, že populace po provedení restartu naroste o násobek $\Delta\lambda$.

Podmínky pro restart vycházejí ze zastavovacích podmínek algoritmu CMA-ES. Z toho zároveň plyne, že zastavení IPOP-CMA-ES se nemůže řídit zastavovacími kritérii klasického CMA-ES.

Výraznou výhodou oproti klasickému CMA-ES je, že IPOP-CMA-ES má mnohem lepší výsledky pro multimodální funkce. Ve své podstatě se jedná o velmi sofistikované náhodné prohledávání, neboť při každém restartu se inicializují parametry na počáteční hodnoty a některé na náhodné. Jakmile je algoritmus poblíž nějakého optima, přibližuje se k němu stejně jako CMA-ES. Když příliš moc rychle konverguje, restartuje se, neboť toto optimum prozkoumal a zkusí nalézt nové.

Implementace algoritmu IPOP-CMA-ES je součástí mé práce.

2.3.2 σ-m-IPOP-CMA-ES

σ-**m**-IPOP-CMA-ES je algoritmus IPOP-CMA-ES rozšířený o nastavení počáteční střední hodnoty a step-size parametru po restartu. Střední hodnota se nastaví jako průměr nejhoršího a nejlepšího jedince podle rovnice 2.11

$$\mathbf{m}^{(0)} = \frac{\mathbf{x}_{best} + \mathbf{x}_{worst}}{2} \tag{2.11}$$

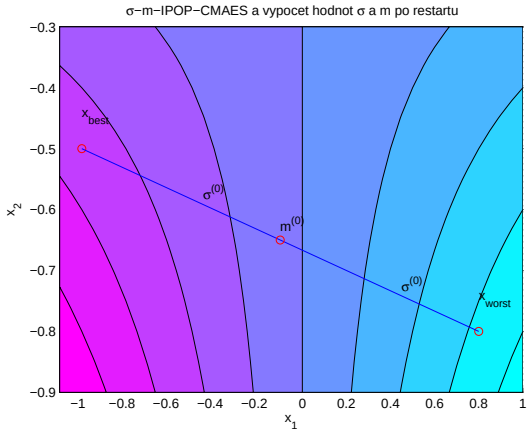
A hodnotu $\sigma^{(0)}$ po restartu vypočítám jako polovinu Eukleidovské vzdálenosti mezi nejlepším a nejhorším jedincem, tedy podle rovnice 2.12.

$$\sigma^{(0)} = \frac{\|\mathbf{x}_{best} - \mathbf{x}_{worst}\|}{2} \tag{2.12}$$

Algoritmus se ukázal jako rychlejší, než jeho původní varianta IPOP-CMA-ES. V případě, že má funkce mnoho stejných lokálních optim, uvázne a osciluje mezi nimi a opakovaně provádí restarty čímž zvyšuje populaci a roste tím i výpočetní náročnost.

Výhodou je, že si pamatuje oproti IPOP-CMA-ES i ta dobrá řešení, proto neustále konverguje k těm lepším i po restartu, a tuto vlastnosti IPOP-CMA-ES nemá. Dále bych doporučil dávat menší počet kroků, neboť tato strategie má lepší výsledky při prohledávání. Bohužel má ale malé úspěchy při konvergenci k optimům. A poslední výsledek není, oproti IPOP-CMA-ES ten nejlepší, ale v průběhu může nalézt překvapivé a velmi atraktivní řešení.

Implementace algoritmu σ-**m**-IPOP-CMA-ES je součástí mé práce.



Obrázek 2.5: Ukázka výpočtu parametrů algoritmu σ-**m**-IPOP-CMA-ES pro funkci $f(\mathbf{x}) = \mathbf{x}\mathbf{x}^T$. Polovina Eukleidovské vzdálenosti (polovina modré čáry) mezi nejlepší $\mathbf{x}_{best}$ a nejhorší $\mathbf{x}_{worst}$ hodnotu slouží k výpočtu $\sigma^{(0)}$ a nová střední hodnota $\mathbf{m}^{(0)}$ po restartu průměr nejlepší a nejhorší hodnoty.

2.3.3 LS-CMA-ES

Další ze známějších variant je LS-CMA-ES [3]. Tato varianta vylepšuje stávající implementaci CMA-ES o přímý výpočet inverzní Hessovy matice ve vhodný okamžik. Algoritmus lze zařadit do třídy L-CMA-ES.

Algoritmus je vhodný pro funkce, které mají eliptický tvar (obecněji jsou unimodální), kde algoritmus LS-CMA-ES konverguje k nejlepším řešením v méně iteracích, než CMA-ES. Rychlost konvergence ke globálnímu optimu je přímočará.

2.3.4 MO-CMA-ES

Varianta MO-CMA-ES [21] spadá do jiné kategorie než všechny ostatní varianty, neboť se jedná o multikriteriální variantu čistého algoritmu CMA-ES pro optimalizaci vícerozměrných vektorových funkcí.

2.3.5 ACTIVE-CMA-ES

Poslední varianta je ACTIVE-CMA-ES, která vychází z článku [22]. Všechny doposud zmíněné implementace zohledňují ve výpočtu parametrů algoritmu pouze μ nejlepších jedinců. Existují dvě možnosti, jak zohledňovat nejhorší jedince. Jedna je zjednodušená a může způsobit problémy u některých funkcí. Nejhorší jedince zohlední tak, že nastaví záporné váhy $\lambda - \mu$ jedincům. Druhá možnost odečítá u výpočtu *rank-μ-update* těchto $\lambda - \mu$ záporně vážené jedince. Výpočet *rank-μ-update* v tomto případě vypadá podle rovnice 2.13.

$$\underbrace{\sum_{i=1}^{\mu} \mathbf{y}_{1:\lambda}^{(g+1)} \mathbf{y}_{i:\lambda}^{(g+1)(g+1)} - \frac{1}{\mu} \sum_{k=\lambda-\mu+1}^{\lambda} \mathbf{y}_{k:\lambda} \mathbf{y}_{k:\lambda}{}^T}_{rank-\mu-update} \tag{2.13}$$

Výhodou algoritmu ACTIVE-CMA-ES je, že využívá k předcházení špatným směrům a vybírá i nejhorší jedince. Experimenty ukázaly, že ACTIVE-CMA-ES má menší počet iterací. Je cca. o 10% rychlejší. Množství generací nutných k dosažení požadované hodnoty fitness je menší pro všechny testované funkce.

Algoritmus má blízko k tzv. tabu search prohledávání.

2.4 Existující implementace algoritmu CMA-ES

Na stránkách autora algoritmu [17] jsou k dispozici všechny známé implementace kódu v různých jazycích. Algoritmus CMA-ES v základní variantě podle článku[11] je k dispozici v následujících jazycích:

- C
- C++
- Fortran
- Java
- Matlab
- Octave
- Python
- Scilab

Značnou část implementací napsal sám autor algoritmu Nikolaus Hansen a součástí všech Hansenových kódů jsou i skripty v Matalbu pro vyhodnocení výsledků algoritmu.

2.4.1 C a C++

V jazyce C je k dispozici pouze čistý algoritmus CMA-ES [14]. Kód je vytvořen tak, aby byl snadno integrovatelný do jiného kódu vč. jazyka C++. Autorem kódu je Nikolaus Hansen. Autor nevytvářel přímou implementaci v jazyce C++, ale psal kód s ohledem na ANSI normu, takže kód lze přeložit i na kompilátorech C++. Poslední revize je ze září 2010.

V jazyce C++ je algoritmus CMA-ES implementován v následujících knihovnách :

- Knihovna **Shark** [20] implementuje $(\mu/\mu_W, \lambda)$-CMA-ES,$(\mu/\mu_W + \lambda)$-CMA-ES, MO-CMA-ES.
- Knihovna **EO - Evolving Objects evolutionary computation framework** [23] implementuje $(\mu/\mu_W, \lambda)$-CMA-ES.
- Knihovna **Open BEAGLE** [6] implementuje $(\mu/\mu_W, \lambda)$-CMA-ES.
- Knihovna **PACLib** implementuje $(\mu/\mu_W, \lambda)$-CMA-ES.

2.4.2 Fortran

V jazyce Fortran je k dispozici pouze čistý algoritmus CMA-ES. Kód [26] je v knihovně **pCMALib**.

2.4.3 Java

V jazyce Java je k existuje implementace čistého algoritmu CMA-ES přímo od autora knihovny Nikolause Hansena, která má i podrobně zapracovanou dokumentaci. Poslední revize je z roku 2007. Dále je pak čistý CMA-ES k dispozici v knihovně **OpenOpal OPtimization And Learning environment**.

2.4.4 Matlab a Octave

Pro Matalb existuje algoritmus v několika vydáních přímo od autora algoritmu:

- **purecmaes.m** [13] je minimální implementace pro Matlab a Octave. Skript je vytvořen tak aby bylo co nejvíce čitelný, algoritmus nemá kromě zastavovací podmínky **ConditionCov** žádné zastavovací podmínky. Tato implementace nemusí být vždy tak efektivní.
- **cmaes.m** [12] je plná implementace algoritmů CMA-ES, IPOP-CMA-ES a ACTIVE-CMA-ES pro Matalb a Octave. Skript je vytvořen s ohledem na efektivnost kódu a má všechny zastavovací podmínky podle článku [11].

Varianta **cmaes.m** je po 12 úpravách ve verzi 3. První verze [9] vznikla v únoru 2003 a je téměř shodná s **purecmaes.m**. V květnu 2006 byla přidána varianta IPOP-CMA-ES [10] a v poslední verzi z června 2008 byla evoluční strategie [12] rozšířena o aktivní výpočet kovarianční matice ACTIVE-CMA-ES podle článku [22].

Poslední verze čistého CMA-ES je 2.55, která je bez rozšíření aktivního výpočtů kovarianční matice a restartů [8].

2.4.5 Python

Pro jazyk Python existuje opět ve dvou variantách od Nikolause Hansena:

- Minimalistická verze [16], skript je vytvořen tak, aby byl co nejvíce čitelný. Skript má jedinou zastavovací podmínku **ConditionCov**.
- Plná verze [15], skript je vytvořen tak, aby byl co nejvíce efektivní. Skript má všechny zastavovací podmínky [11] a implementuje restartovací variantu IPOP-CMA-ES [2].

Kapitola 3

Implementace algoritmu CMA-ES do knihovny JCool

3.1 Knihovna JCool

JCool je open-source knihovna pro jazyk Java na optimalizaci spojitých funkcí. Implementuje jak numerické (Gradient, Sdružený gradient, Quasi-Newton, ...) a přírodou inspirované (Evoluční, PSO, ACO, ...) algoritmy. Součástí knihovny je sada testovacích funkcí.

Knihovna JCool je rozdělena na 5 následujících modulů:

benchmark - Modul obsahuje všechny testovací funkce a algoritmy na optimalizaci, které jsou v knihovně k dispozici.

core - Modul obsahuje rozhraní pro ostatní kód (rozhraní pro funkce, bod, telemetrii...).

experiment - Modul obsahuje třídy pro experimenty, tedy třídy, které spojují algoritmy s jejich testovacími funkcemi a solvery.

solver - Modul obsahuje třídy solverů, tedy tříd, které pracují s algoritmy a předávají jim odpovídající testovací funkce.

ui - Modul obsahuje třídy pro práci s grafickým rozhraním knihovny.

3.2 Knihovna Commons Math

Commons Math je matematická knihovna, implementující statistické a algebraické operace, které nejsou k dispozici v jazyce Java. Základní rysy jsou [1]:

- Knihovna je vyvíjena s ohledem na jednoduchou použitelnost.
- Knihovna zdůrazňuje malé, jednoduše integrovatelné komponenty spíše, než velké knihovny s množstvím závislostí.
- Všechny algoritmy v knihovně jsou důkladně dokumentovány podle obecně uznávaných nejlepších postupů.

- V případě, že existuje více obecně uznávaných algoritmů pro nějaký problém, návrhový vzor Strategy podporuje více možností implementace.
- Knihovna je bez závislostí na další balíčcích.

V metodě je značné množství maticových operací i z pokročilejší algebry, proto jsem pro maticové operace použil knihovnu **Commons-Math** verze 2.1, která plně vyhovovala algebraickým výpočtům. Z knihovny využívám velké množství funkcí pro operace nad maticemi a vektory, rozklad na vlastní čísla a vlastní vektory kovarianční matice $\mathbf{C}$ a implementaci generátoru vektorů s normálním rozdělením podle kovarianční matice $\mathcal{N}(\mathbf{m}, \mathbf{C})$.

Instance třídy **RealMatrix** implementují matice. Třídu jsem použil pro implementaci algebraických operací pro násobení s maticemi (metoda **multiply**), skalární součin matice a vektoru (metoda **operate**), násobení a přičítání skaláru k prvkům matice (metoda **scalarAdd** a **scalarMultiply**), pro transpozici matice (metoda **transpose**) a pro počítání maximální normy matice ($\mathcal{L}_\infty$ norma podle [29]) pomocí metody **getNorm**.

Instance třídy **EigenDecomposition** implementují rozklad na vlastní čísla a vlastní vektory, kde instancí třídy **EigenDecompositionImpl** metodou **getRealEigenvalues** získáme pole seřazených vlastních čísel (stejně jako metoda **getD**, která vrací diagonální matici vlastních čísel) a metodou **getEigenvector(int i)** získáme i-tý vlastní vektor k odpovídajícímu vlastnímu číslu na indexu $i - 1$ (rovněž lze použít metodu **getV**, která vrací matici se sloupci vlastních vektorů).

Knihovna je dlouho vyvíjena a dlouho ještě bude, proto jsem volil zrovna Commons-Math, protože je rychlá, velmi dobře odlaďovaná a stále vyvíjená a podporovaná. Díky použití maticové knihovny jsem dosáhl efektivní a především přehledné implementace, neboť předpokládám, že ve vývoji se ještě bude pokračovat i po obhajobě.

3.2.1 Základní rozhraní knihovny JCool

Předávání výsledků a práce s výsledky ošetřují rozhraní **Consumer** a **Producer**. **Producer** má na starosti vytváření výsledků, zatímco **Consumer** informace přijímá. Volání a předávání výsledků do třídy konzumenta má na starosti odpovídající potomek třídy **OptimizationMethod**.

Třídy **Consumer** a **Producer** může obsluhovat jeden z potomků rozhraní **Solver**. **Solver** pracuje s producentem i konzumentem a stará se o řádné volání producenta a předávání výsledků konzumentovi. K dispozici jsou tři třídy:

- **BasicSolver**.
- **TimeoutSolver** .
- **StepSolver**.

Výsledky lze ze třídy **Solver** získat přes metodu **getResults**, která vrací instanci **OptimizationMethod**. Potomci třídy **Solver** dávají výsledky z výpočtu algoritmu, které předávají pomocí instance třídy **Synchronization**.

Třída **OptimizationMethod** zapouzdřuje informace o zastavovacích podmínkách, počet iterací algoritmu, výsledné řešení (potomek třídy **Telemetry**) a statistické informace o průběhu třídy **Statistics**.

3.2.2 Implementace algoritmů v knihovně JCool

Veškerý kód o který jsem knihovnu **JCool** rozšířil je v modulu **benchmark**. Kód je rozdělen do několika tříd v různých balíčcích. Algoritmy CMA-ES, IPOP-CMA-ES a σ-**m**-IPOP-CMA-ES jsou uloženy v balíčku **src.main.java.cz.cvut.felk.cig.jcool.benchmark.method.cmaes**. Následující výčet ukazuje v jakém souboru je jaká metoda uložena:

- **CMA-ES** v souboru **PureCMAESMethod.java**.
- **IPOP-CMA-ES** v souboru **IPOPCMAESMethod.java**.
- σ-**m-IPOP-CMA-ES** V souboru **SigmaMeanIPOPCMAESMethod.java**.

Zastavovací podmínky jsou uloženy v modulu **benchmark** v balíčku **src.main.java.cz.cvut.felk.cig.jcool.benchmark.stopcondition**. Následující výčet ukazuje v jakém souboru je jaká zastavovací podmínka uložena:

- **NoEffectAxis** V souboru **NoEffectAxisStopCondition.java**.
- **NoEffectCoord** V souboru **NoEffectCoordStopCondition.java**.
- **EqualFunValues** V souboru **EqualFunValuesStopCondition.java**.
- **ConditionCov** V souboru **ConditionCovStopCondition.java**.
- **TolFun** V souboru **TolFunStopCondition.java**.
- **TolX** V souboru **TolXStopCondition.java**.

Předek generické třídy **OptimizationMethod** třída **Producer** slouží k předávání dosavadních výsledků daného algoritmu. Všechny instance této třídy jsou producenty. Třída má dvě abstraktní metody a to:

```
void addConsumer(Consumer<? super T> consumer);
T getValue();
```

První metoda přidává nové instance příjemců informací o stavu algoritmu, které jsou instancemi abstraktní třídy **Consumer**. Pro tyto účely všichni potomci musí implementovat metody, **addConsumer**, které slouží k nastavení objektu konzumenta a zároveň tento objekt slouží k předávání výsledků daného algoritmu konzumentovi. K získávání výsledků slouží **getValue**. Výsledky mohou být instance potomků abstraktní generické třídy Telemetry, tedy:

- **ValueTelemetry** - uchovává stávající hodnotu *value* fitness funkce f.
- **ValuePointTelemetry** - uchovává stávající hodnotu *value* fitness funkce f a pozici $\mathbf{x}$ hodnoty $value = f(\mathbf{x})$.
- **ValuePointTelemetryList** - uchovává stávající hodnoty $value_i$ fitness funkce f a jejich odpovídající pozice $\mathbf{x}_i$ hodnoty $value_i = f(\mathbf{x}_i)$.

- **ValuePointTelemetryList** - uchovává stávající hodnoty $value_i$ fitness funkce f a jejich odpovídající pozice $\mathbf{x}_i$ hodnoty $value_i = f(\mathbf{x}_i)$ a barvu $color_i$(hodnocení, zda je odpovídající bod i nejlepší z celého seznamu).

Na obrázku E.3 je UML diagram třídy **Telemetry** a jeho potomků.

Dále má knihovna JCool rozhraní **OptimizationMethod**. Od ní jsou odvozené všechny třídy implementující algoritmy pro optimalizaci. Rozhraní má tři metody, které musí implementovat potomek:

```
void init(ObjectiveFunction function);
StopCondition[] getStopConditions();
void optimize();
```

První metoda **init**, slouží k inicializaci dané optimalizační metody v potomkovi a předání dané fitness funkce f. Ta je potomkem třídy **ObjectiveFunction**.

Druhá metoda **getStopConditions** vrací pole instancí **StopConditions**, což je seznam zastavovacích podmínek dané metody. Třída, která pracuje s optimalizačními algoritmy má potom na starosti obsluhu zastavení algoritmu, tj. volá cyklicky všechny zastavovací podmínky, které vrátí v poli a metoda **isConditionMet** třídy **StopCondition** a vrátí v závislosti na stavu, zda se má algoritmus zastavit, či ne.

A poslední abstraktní metoda **optimize** zapouzdřuje a slouží svým potomkům k implementaci jednoho kroku optimalizačního algoritmu. Jedno volání metody **optimize** odpovídá jedné iteraci daného algoritmu.

To jak by měla vypadat obsluha obecného optimalizačního algoritmu v knihovně JCool pomocí abstraktního předka **OptimizationMethod** je podrobněji popsána v algoritmu 3.

```
   input : Instance optimalizační metody method
1  while true do
2  |  method.optimize()
3  |  stopConditions = method.getStopConditions()
4  |  foreach stopCondition in stopConditions do
5  |  |  if stopCondition. isConditionMet() is true then
6  |  |  |  Break;
7  |  |  end
8  |  end
9  end
```

Algorithm 3: Obsluha optimalizačních metod pomocí abstraktní třídy Optimization-Method.

Algoritmy vyhodnocují svou kvalitu na základě fitness funkce f, která je reprezentována v knihovně rozhraním **Function**. Pokud má fitness funkce určený gradient, druhou derivaci (Hessián) nebo je nějak omezená, je možné odvodit implementaci funkce kvality od tříd **FunctionGradient**, **FunctionHessian** nebo **FunctionBounds**. Všechna tato rozhraní v sobě sjednocuje rozhraní **ObjectiveFunction**, které je předáváno jako parametr při volání funkce **init** odvozených tříd od **OptimizationMethod**.

Ne všechny fitness funkce mají známý gradient, Hessián nebo omezení. Pro tento účel existuje třída **BaseObjectiveFunction**, která v sobě zastřešuje obecnou funkci a zároveň je potomkem **ObjectiveFunction**. Pokud má funkce gradient, Hessián nebo omezení vyvolá odpovídající metodu předka, jinak vyvolá výjimku. Instance třídy **BaseObjectiveFunction** slouží k předávání instance fitness funkce v metodě **init**.

3.3 Implementace do knihovny JCool

Na začátku jsem předpokládal, že budu implementovat pouze základní algoritmus CMA-ES, takže základní myšlenka byla koncipována tak, že nebudu dědit žádnou další třídu z hlavní třídy **CMAESMethod**. Před vlastním kódováním jsem se rozhodl, že přidám i další variantu IPOP-CMA-ES, takže celá koncepce se rozšířila o další třídu, která se mírně odchylovala od výpočtu klasického algoritmu CMA-ES. Návrh jsem tedy rozšířil o další čtyři třídy, které mají hierarchii znázorněnou na obrázku E.1.

3.3.1 Základní třída CMAESMethod

Třída **CMAESMethod** slouží jako abstraktní předek všech mnou implementovaných variant a tříd algoritmu CMA-ES. Třída je abstraktní, takže implementace čistého algoritmu CMA-ES je přenechána třídě **PureCMAESMethod**.

3.3.1.1 Obecně k implementaci třídy CMAESMethod

CMAESMethod zahrnuje základní výpočty, které jsou obecné pro všechny metody jako je výpočet hodnot $\mathbf{p}_c$, $\mathbf{p}_\sigma$, σ a $\mathbf{C}$. Tyto základní výpočty jsou rozděleny do separátních metod, tak aby se každá členská metoda mohla jmenovat podle toho, jakou proměnnou počítá a případně bylo možné od ní odvodit patřičnou třídu, která provádí výpočet jinak. Dále pak třída implementuje základní podmínky pro zastavení, jako konvergence k řešení. Zbývající zastavovací podmínky jsem přenechal na jejich potomcích, protože potomci **RestartCMAESMethod** využívají tyto zastavovací podmínky jako podmínky pro restart algoritmu. Celý algoritmus potom provádí metoda **optimize**, která sjednocuje základní výpočty hodnot jedné metody.

Předchůdce, generická třída **OptimizationMethod**, a jako generický parametr předka používám typ **ValuePointListTelemetry**, protože mám populaci více jedinců.

K implementaci jsem použil jako předlohu článek [18] a především předlohovou variantu algoritmu v jazyce Matlab [13]. K porovnání jsem použil referenční kód v jazyce Matlab ze zdroje [13]. Pokud jsem potřeboval podrobnější a především aktuálnější informaci, použil jsem plnou implementaci [12].

Výsledný kód jsem ověřoval porovnáváním hodnot v jednotlivých krocích při generování stejných náhodných hodnot vůči implementaci [13] jejíž kód je zjednodušen a nemá většinu zastavovacích podmínek, kromě **ConditionCov**.

Seznam všech atributů třídy **CMAESMethod** s popisem a jeho ekvivalentem v analýze a popisu algoritmu je uveden v tabulce A.1.

3.3.1.2 Integrace třídy CMAESMethod do knihovny JCool

Abstraktní předchůdci třídy **CMAESMethod**, třída **OptimizationMethod** a **Producer** mají 5 abstraktních metod, které jsou:

- **addConsumer** pouze nastavuje „konzumenta“ výsledků algoritmu. Tělo metody tedy nastavuje pouze členský **consumer** atribut.
- **getValue**, jenž vrací typ ValuePointListTelemetry, který jsem definoval jako generický parametr předchodce **OptimizationMethod**.
- **init**, v níž volám metodu **initializeDefaultParameters**, která nastavuje všechny hodnoty do iniciálního stavu, nastavím v parametru předanou fitness funkci a některé její parametry.
- **getStopConditions**, vrací prázdné pole instancí StopCondition.
- **optimize**, je metoda která implementuje v několika metodách základní kroky algoritmu CMA-ES.

Třída má jedinou abstraktní metodu, **setStopConditionParameters**, v níž její potomci nastavují parametry zastavovacích podmínek.

3.3.1.3 Inicializační krok ve tříde CMAESMethod

O inicializaci algortimu se stará metoda **init**, které ten kdo jí volá předá parametrem instanci třídy **ObjectiveFunction**. V inicializační části metody se nastavují atributy daného problému, inicializace proměnných algoritmu je implementovaná v metodě **initializeDefaultParameters**.

Důvodem proč mám rozdělenou inicializační část algoritmu na dvě části je kvůli implementacím založených na restartech. Implementace CMA-ES založené na restartech vyžadují inicializaci většiny atributů na počáteční hodnoty. Metoda **init** je zde především pro předání informací o řešeném problému z jiného kódu. **init** sice volá **initializeDefaultParameters**, ale rovněž jí volají potomci třídy **RestartCMAESMethod**.

Stejný důvod, jako je rozdělení **init** na dvě metody má i rozdělení proměnné σ na dva atributy **sigma** a **initialSigma**. Pokud totiž obsluha algoritmu nastaví, že chce nějakou počáteční hodnotu atributu **sigma**, tak jej algoritmus v průběhu výpočtu modifikuje a původní počáteční hodnotu přepíše. V případě strategie založené na restartech nemáme nikde uloženou počáteční hodnotu a k tomu slouží atribut **initalizeSigma**

Metoda **init** nastavuje atribut fitness funkce **function**, minimální a maximální meze dané funkce a hodnotu nejlepšího a nejhoršího jedince ve všech doposud provedených generacích. Atribut **theBestPoint** je instancí třídy **ValuePoint** a na začátku je nastavena jeho pozice na hodnotu **Point.getDefault()** a fitness na **Double.MAX_VALUE**, takže v první generaci algoritmu musí existovat jedinec s lepší fitness, který hodnotu **theBestPoint** nahradí. Další atribut **theWorstPoint** je instancí třídy **ValuePoint** a na začátku je jeho pozice stejná jako v případě **theBestPoint**, ale fitness je nastavena na **-Double.MAX_VALUE**, takže v první generaci algoritmu musí existovat jedinec s horší fitness, který hodnotu **theWorstPoint** nahradí.

Nyní popíši zobecněnou metodu **initializeDefaultParameters** podle algoritmu 4, řádek po řádku:

1-5 Inicializuji vektor **xmean** náhodnými hodnotami s rovnoměrným rozdělením, tak aby hodnota **xmean** nepřesáhla maximální i minimální možnou hodnotu funkce **function** ve všech souřadnicích.

6 Nastavím počáteční hodnoty step-size parametru **sigma**, tedy $\sigma^{(0)}$ na předem nastavenou počáteční hodnotu **initialSigma**.

8 Nastavím velikost populace, proměnnou λ, tedy atribut **lambda**.

9 Nastavím velikost selektované populace, proměnnou μ, tedy atribut **mu**. Atribut by měl být přibližně polovina celkového množství populace.

11-19 Nastavení vah selekce, podle vzorce z teorie. Nejdříve inicializuji hodnoty jako rostoucí posloupnost od 0 do μ, potom váhy logaritmuji dekadickým logaritmem.

21 Vypočítám normu váhového vektoru **weight** pro výpočet atributu **muEff**.

22 Vypočítám efektivní hodnoty selekce μ_{eff} jako poměr druhé mocniny normy k normě druhé mocniny.

24 Vypočítám kumulační konstantu c_c, tedy atribut **cCumulation**. Počítá se podle určeného vzorce podle z [18].

25 Vypočítám konstantu pro historii σ proměnnou c_σ, tedy atribut **cSigma**. Počítá se podle určeného vzorce z [18].

26 Vypočítám konstantu pro *rank-1-update* c_1, tedy atribut **cRank1**. Počítá se podle určeného vzorce z [18].

27 Vypočítám konstantu pro *rank-μ-update* c_μ, tedy atribut **cRankMu**. Počítá se podle určeného vzorce z [18].

28 Vypočítám tlumící konstantu σ proměnnou d_σ, tedy atribut **dampingForSigma**. Počítá se podle určeného vzorce z [18].

30 Nastavím vektor kumulace cesty $\mathbf{p}_c^{(0)}$ pro výpočet *rank-1-update* na nulový vektor, tedy atribut **pCumulation** z [18].

31 Nastavím vektor normalizované kumulace cesty $\mathbf{p}_c^{(0)}$ pro výpočet step-size σ na nulový vektor, tedy atribut **pSigma** z [18].

33-34 Nastavím souřadné systémy pro normální rozdělení. **B** nastavím na matici identity, stejně jako vektor **D** nastavím na jednotkový. Takto vygenerované body budou mít stejnou pravděpodobnost výskytu od střední hodnoty.

35 Nastavím matici $\mathbf{C}^{(0)}$, tedy atribut **C** na matici identity. Hodnoty **C** počítám podle **D** a **B**, abych předešel problémům s nejednoznačností rozkladu, pro případ volby jiné počáteční hodnoty $\mathbf{D}^{(0)}$ a $\mathbf{B}^{(0)}$.

36 Nastavím matici $\mathbf{C}^{-\frac{1}{2}^{(0)}}$, tedy atribut **inverseAndSqrtC** na matici identity. Hodnoty **invserseAndSqrtC** počítám podle rozkladu ze stejného důvodu, jako počítám atribut **C**.

38 Vypočítám odhad $\mathcal{L}_2$ normy normálního rozdělení $\|\mathcal{N}(0, \mathbf{I})\|$ podle [18]

3.3.1.4 Optimalizační krok ve třídě CMAESMethod

Metoda je implementována tak, aby měla minimum lokáních proměnných a většinu proměnných ukládala do svých atributů, aby je případně mohla sdílet se svými potomky. Nyní popíši zobecněnou metodu **optimize** podle algoritmu 5, řádek po řádku.

1 Vytvořím novou populaci pomocí normálního rozdělení z parametrů vypočtených z původní generace (případně inicializačních parametrů). Metoda **generatePopulation** vrací pole instancí **ValuePoint**, tedy novou ohodnocenou množinu jedinců.

2 Uložím si střední hodnotu z původní generace, protože bude potřeba při výpočtu některých dalších parametrů.

3 Setřídím celou populaci. Protože je třída **ValuePoint** potomkem rozhraní **Comparable**, setřídí se mi populace vzestupně.

4 Uložím nejlepšího a nejhoršího jedince pro všechny generace, pokud takový existuje v nově vytvořené generaci.

5 Pomocí metody **createMatrixFromPopulation** vytvořím instanci nejlepších vybraných (μ) jedinců třídy **RealMatrix** z instancí **ValuePoint** tak, že každý sloupec představuje jednoho jedince. Instance třídy **RealMatrix** má N řádků a μ sloupců.

7 Každý sloupec vynásobím hodnotou na odpovídajícím indexu vektoru vah **weight**. Touto operací získám ze střední hodnoty vybrané populace váženou, což odpovídá výpočtu proměnné $\mathbf{m}^{(g+1)}$

8 Výpočet hodnoty atributu **pSigma** $\mathbf{p}_\sigma^{(g+1)}$ pomocí metody **calculatePSigma**.

9 Výpočet hodnoty h_σ pomocí metody **calculateHSigma**. Hodnota h_σ slouží k udržení $\mathbf{p}_c^{(g)}$, jestliže $\|\mathbf{p}_\sigma^{(g)}\|$ nabývá velkých hodnot. Tato hodnota se využívá při výpočtu proměnných $\mathbf{p}_c^{(g+1)}$ a $\mathbf{C}^{(g+1)}$, tedy v metodách **calculatePCumulation** a **calculateCovariance**.

10 Vypočtu hodnoty proměnné $\mathbf{p}_c^{(g+1)}$, tedy atributu **pCumulation** pomocí metody **caculatePCumulation**. Parametrem předáme starou střední hodnotu **xold** a hodnotu **hSigma**, která reguluje zda bude hodnota změněna, nebo ne. Tato hodnota později poslouží k výpočtu kovarianční matice jako *rank-1-update*.

11-12 Vypočtu vzdálenost nejlepších **mu** jedinců z populace od střední hodnoty dělené hodnotou **sigma**. Výsledek bude opět instance třídy **RealMatrix** tak, že každý sloupec představuje jednoho jedince. Každý sloupec odpovídá výpočtu $\mathbf{y}_{i:\lambda}^T$.

```
1  this.xmean =RealVector (this.N);
2  for n =0 to N-1 do
3  |  this.xmean.setEntry (n, (Max- Min)*(Math.random () - 0.5));
4  |  n = n + 1
5  end
6  this.attributeSigma =initialSigma
7  // inicializace parametrů ES pro selekci, λ, μ
8  this.attributeLambda = 4+Math.floor (3*Math.logarithm (this.N));
9  this.Mu = Math.floor (attributeLambda/2);
10 // inicializace vah pro selekci w
11 this.weights =RealVector (N);
12 for n =0 to this.N-1 do
13 |  this.weights.setEntry (n,n +1)
14 |  n =n +1
15 end
16 this.weights = this.weights.mapLog ().mapMultiply (-1).mapAdd (Math.logarithm
   (this.Mu +0.5))
17 Norm = this.weights.getNorm ();
18 // normalizace vah
19 this.weights.mapDivideToSelf (Norm);
20 // součet normalizovaných vah pro výpočet μ_eff
21 Norm = this.weights.getNorm ();
22 this.muEff = Math.pow (Norm, 2) / this.weights.mapPow (2).getNorm ();
23 // inicializace koeficientů adaptace c_c, c_σ, c_1, c_μ, d_σ
24 this.cCumulation = (4 + this.muEff/ this.N) / (this.N + 4 + 2 * this.muEff/ 2);
25 this.cSigma = (this.muEff + 2) / (this N + this.muEff + 5);
26 this.cRank1 = 2 / (Math.pow (this.N + 1.3, 2) + this.muEff);
27 this.cRankMu = 2 * (this.muEff- 2 + 1 / this.muEff) / (Math.pow (this.N + 2, 2) +
   this.muEff);
28 this.dampingForSigma = 1 + 2 * Math.maximum (0, Math.sqrt ((this.muEff- 1) / (this.N
   + 1)) - 1) + this.cSigma;
29 // dynamické parametry strategie p_c^(0), p_σ^(0), B^(0), D^(0), C^(0) a C^(-1/2)^(0)
30 this.pCumulation =RealVector (this.N);
31 this.pSigma =RealVector (this.N);
32 // definuje souřadný systém
33 this.B = MatrixUtils.createRealIdentityMatrix (this.N);
34 this.D = RealVector (this.N, 1);
35 this.Cov = B.mapMultiply (MatrixUtils.createRealDiagonalMatrix (this.D.mapPow
   (2).toArray ())).mapMultiply (this.B.transpose ());
36 this.inverseAndSqrtC = this.B.mapMultiply (MatrixUtils.createRealDiagonalMatrix
   (this.D.toArray ())).mapMultiply (this.B.transpose ());
37 // odhad normy ||N(0, I)|| pro výpočet d_σ this.chiN = Math.sqrt (N) * (1 - 1 / (4 *
   N)) + (1 / (21 * Math.pow (N, 2))));
```

Algorithm 4: Jeden krok metody initializeDefaultParametes třídy CMAESMethod. V kódu značím **Cov** kovarianční maticí $\mathbf{C}^{(g)}$ a **attributeSigma** step-size parametr $\sigma^{(g)}$.

15 Vypočtu novou hodnotu atributu **C**, tedy kovarianční matice $\mathbf{C}^{(g+1)}$ pomocí metody **calculateCovariance**. Všechny proměnné, které jsou nutné k výpočtu, kromě **hSigma** a **muDifferenceFromOldMean** (které představují ve výpočtu *rank-1-update*), jsou atributy třídy **CMAESMethod**.

17 Vypočtu novou hodnotu **sigma**, tedy step-size parametr $\sigma^{(g+1)}$.

18 Rozklad podle 1.12 matice $\mathbf{C}^{(g)}$ pomocí metody **calculateEigendecomposition**, tedy na ortonormální matici $\mathbf{B}^{(g+1)}$ a diagonální matici vlastních čísel $\mathbf{D}^{(g+1)}$. Metoda pak dále počítá hodnotu atributu **inverseAndSqrtC**, tedy proměnné $\mathbf{C}^{-\frac{1}{2}}$, která slouží k odstranění závislosti **pSigma**, tedy $\mathbf{p}_{\sigma}^{(g+1)}$ na kovarianční matici.

19 Inkrementace čítače počtu generací

20 Uložení nejlepšího jedince do atributu **bestValueInCurrentGeneration**, tedy hodnoty $\mathbf{x}_{1:\lambda}^{(g)}$.

21 Nastavení atributu **telemetry** pro předání výsledků. Jedná se o instanci třídy **ValuePointListTelemetry**, proto předáváme celou populaci.

22,23,24 Předání výsledků provedené generace instanci **consumer**.

Proměnná **hSigma** je lokální, protože neuchovává svou historii, ani není použitelný v rámci jiných implementací.

Výpočet hodnoty h_{σ} je 1 jestliže platí nerovnost 3.1, jinak je rovna 0.

$$\frac{\|p_{\sigma}^{(}g)\|}{\sqrt{1-(1-c_{\sigma})^{2(g+1)}}} < \sqrt{n}\left(1-\frac{1}{4N}+\frac{1}{21N^2}\right) \tag{3.1}$$

Metody, které počítají proměnné algoritmu:

- **initializeDefaultParameters** - nastavuje všechny proměnné algoritmu na počáteční hodnotu.
- **generatePopulation** - generuje novou populaci o λ jedincích s normálním rozdělením. Generování je prováděno pomocí třídy **CorrelatedRandomVectorGenerator** z knihovny **Commons-Math**.
- **calculateEigendecomposition** - počítá rozklad na vlastní čísla a vlastní vektory z kovarianční matice $\mathbf{C}^{(g+1)}$. Rozklad je prováděn pomocí třídy **EigenDecomposition** z knihovny **Commons-math**.
- **calculateCovariance** - počítá novou kovarianční matici $\mathbf{C}^{(g+1)}$.
- **calculatePSigma** - počítá novou hodnotu $\mathbf{p}_{\sigma}^{(g+1)}$.
- **calculatePCumulation** - počítá novou hodnotu $\mathbf{p}_{c}^{(g+1)}$.
- **calculateHSigma** - počítá hodnotu h_{σ}.

```
1  population = generatePopulation ()
2  xold = xmean
3  Arrays.sort (population)
4  storeTheBestAndWorst (population)
5  muBestIndividuals = createMatrixFromPopulation (population, this.Mu, this.N)
6  // výpočet nové vážené střední hodnoty m^(g+1)
7  this. xmean = muBestIndividuals.operate (weights)
8  calculatePSigma (xold)
9  hSigma = calculateHSigma ()
10 calculatePCumulation (hSigma,xold)
11 // výpočet y pro rank-μ-update
12 muDifferenceVectorsFromOldMean =
13 calculateMuDifferenceFromOldMean (muBestIndividuals,this.xold, this.N, this.Mu,
   this.stepSize)
14 // výpočet nové kovarianční matice C^(g+1)
15 calculateCovariance (hSigma,muDifferenceVectorsFromOldMean)
16 // výpočet nové hodnoty σ^(g+1)
17 calculateSigma ()
18 calculateEigendecomposition ()
19 this.currentGeneration =this.currentGeneration +1
20 bestValueInCurrentGeneration = population [0].getValue ();
21 telemetry = Arrays.asList (population);
22 if consumer!= null then
23 |  consumer.notifyOf (this)
24 end
```

Algorithm 5: Jeden krok v kódu metody optimize třídy CMAESMethod

- **calculateSigma** - počítá novou hodnotu $\sigma^{(g)}$.

Pomocné metody, které slouží k výpočtům proměnných algoritmu:

- **calculateMuDifferenceFromOldMean** - počítá vzdálenost každého sloupce matice (jedince) od střední hodnoty $\mathbf{m}^{(g)}$ dělené *step-size* atributem $\sigma^{(g)}$. Statická metoda podporuje výpočet *rank-μ-update* kovarianční matice $\mathbf{C}^{(g+1)}$.
- **createMatrixFromPopulation** - vytváří z μ vybraných jedinců matici. Každý sloupec představuje reprezentaci jednoho jedince. Statická metoda zjednodušuje výpočet.
- **triu** - vrací horní trojúhelníkovou matici ze zadané matice **toTriu**. Statická metoda se používá při výpočtu v metodě **computeEigendecomposition** v níž zaručuje symetrii kovarianční matice do formy 1.12.

Zbývající nezmíněné metody jsou již popsané a slouží k nastavování parametrů z GUI, nebo k informování o výsledcích a již jsem je dříve popsal.

3.3.2 Třída RestartCMAESMethod

Třída **RestartCMAESMethod** je koncipována jako abstraktní předek všech mnou implementovaných variant a tříd algoritmu CMA-ES založených na restartech. Uvedená třída obsahuje implementace obsluh restartů v metodě **optimize** algoritmem 6 a pokud je nurné vyvolat restart, tak metodou **restart** algoritmem 7. Odvozená třída musí implementovat podmínky pro restart metodou **isRestartConditionsMet** a novou hodnotu výpočtu proměnné po restartu λ metodou **calculateLambda**, protože právě tyto věci jsou podstatou restartovacího algoritmu.

3.3.2.1 Obecně k implementaci třídy RestartCMAESMethod

Třída **RestartCMAES** rozšiřuje abstraktního předka **CMAESMethod** o následující metody:

```
abstract void calculateLambda();
abstract boolean isRestartConditionsMet();
void restart();
public void optimize();
int getRestartCounter()
```

Třída rozšiřuje chování metody **optimize** o kontrolu, zda nebyla v právě proběhlé iteraci splněna některá z restartovacích podmínek (viz. popis **optimizeOfRestartStrategyAlg**). Pokud byla splněna alespoň jedna z restartovacích podmínek sponěna vyvolá metodu **restart**.

Vyvolání restartu vyvolá metodu **initializeDefaultParameters** předka **CMAESMethod** (viz. algoritmus 4). **initializeDefaultParameters** inicializuje všechny proměnné pro výpočet algoritmu na počáteční hodnotu. Dále pak volá metodu přepočtu parametru λ po

restartu (IPOP-CMA-ES po restartu zvyšuje množství populace o násobek **increasePopulationMultiplier**) **calculateLambda**. Postup je na 7

To, zda jsou restartovací podmínky splněny je plně v kompetenci potomků, protože metoda **isRestartConditionsMet** ve třídě **RestartCMAESMethod** je abstraktní. Stejně jako **calculateLambda**.

Poslední metoda třídy je **getRestartCounter**. Metoda vrací počet již provedených restartů (tedy počet volání metody **restart**).

```
1 super.optimize()
2 if isRestartConditionsMet() then
3 |  restart()
4 end
```

Algorithm 6: Obsluha jednoho optimalizačního kroku instance restartovací strategie

```
1 restartCounter +=restartCounter +1
2 initializeDefaultParameters()
3 calculateLambda()
```

Algorithm 7: Obsluha jednoho restartu restartovací strategie

3.3.3 Třída IPOPCMAESMethod

Třída **IPOPCMAESMethod** implementuje abstraktní třídu **CMAESMethod**, kde předkovi implementuje restartovací podmínky, které jsou shodné s podmínkami pro zastavení třídy **PureCMAESMethod**. Jedná se o tyto podmínky:

- **NoEffectAxis** v instanci **noEffectToAxisStopCondition** třídy **NoEffectAxisStopCondition**.
- **NoEffectCoord** v instanci **noEffectToAxisStopCondition** třídy **NoEffectAxisStopCondition**.
- **EqualFunValues** v instanci **equalFunValuesStopCondition** třídy **EqualFunValuesStopCondition**.
- **ConditionCov** v instanci **conditionCovStopCondition** třídy **ConditionCovStopCondition**.
- **TolFun** v instanci **tolFunStopCondition** třídy **TolFunStopCondition**.
- **TolX** v instanci **tolXStopCondition** třídy **TolXStopCondition**.
- instance třídy **SimpleStopCondition**, což je obecná zastavovací podmínka která detekuje konvergenci fitness.

Třída má jednu proměnnou **increasePopulationMultiplier**, která udává násobek $\Delta\lambda$ po restartu. Tento atribut je použit při volání metody **calculateLambda**, která je volána při každém restartu metodou **restart**.

Metody jsou následující:

```
void init(ObjectiveFunction function)
void setStopConditionParameters()
void optimize()
boolean isRestartConditionsMet()
void restart()
void calculateLambda()
```

Metoda **init** se stará pouze o správné počáteční nastavení parametrů všech restartovacích podmínek a volání metody **init** svého předka.

Metoda **setStopConditionParameters** nedělá nic, protože třída **IPOPCMAESMethod** nemá žádné zastavovací podmínky, kromě jedné nadřazené (omezení času, nebo počtu iterací).

Metoda **optimize** se stará o volání jednoho kroku algoritmu a předání stavových proměnných pro restartovací podmínky. O kontrolu zda, je restartovací podmínka splněna se stará metoda **optimize** v **RestartCMAESMethod** pomocí volání **isRestartConditionsMet**.

Metoda **isRestartConditionsMet** buď vrací **true**, je-li alespoň jedna ze zmíněných podmínek splněna, jinak je **false**.

Metoda **restart** nastavuje stejně jako **init** všechny restartovací podmínky do počátečního stavu.

3.3.4 Třída PureCMAESMethod

Třída **PureCMAESMethod** implementuje abstraktní třídu **CMAESMethod**, kde svému předkovi přidává zastavovací podmínky, které nemůže **CMAESMethod** implementovat, kvůli restartovacím podmínkám (jakmile by byla splněna restartovací podmínka, byla by splněna i podmínka zastavovací a předek **CMAESMethod** by mohl výpočet zastavit). Zbytek je přenechán na předkovi. Třída **PureCMAESMethod** implementuje následující metody:

```
void init(ObjectiveFunction function)
void optimize()
void setStopConditionParameters()
CMAESStopCondition[] getStopConditions()
```

Třídu mohou zastavit následující zastavovací podmínky:

- **NoEffectAxis** v instanci **noEffectToAxisStopCondition** třídy **NoEffectAxisStopCondition**.
- **NoEffectCoord** v instanci **noEffectToAxisStopCondition** třídy **NoEffectAxisStopCondition**.

- **EqualFunValues** v instanci **equalFunValuesStopCondition** třídy **EqualFunValuesStopCondition**.
- **ConditionCov** v instanci **conditionCovStopCondition** třídy **ConditionCovStopCondition**.
- **TolFun** v instanci **tolFunStopCondition** třídy **TolFunStopCondition**.
- **TolX** v instanci **tolXStopCondition** třídy **TolXStopCondition**.
- instance třídy **SimpleStopCondition**, což je obecná zastavovací podmínka která detekuje konvergenci fitness.

Metoda **init** zajišťuje správné počáteční nastavení parametrů všech zastavovacích podmínek a dále pak volá metodu **init** svého předka.

Volání metody **optimize** je jeden krok algoritmu. Metoda pracuje se předkem ve třídě **CMAESMethod** a následně nastavuje parametry zastavovacích podmínek po výpočtu voláním metody **setStopConditionParameters**.

Metoda **setStopConditionParameters** nastavuje všechny zastavovací podmínky po provedení jednoho kroku algoritmu (volání metody **optimize**).

Metoda **getStopConditions** vrací pole všech zastavovacích podmínek uvedené na předchozím výčtu, tak aby ten kdo algoritmus obsluhuje mohl ověřit, zda nějaká z nich nebyla splněna (metoda **isConditionMet**).

3.3.5 Třída SigmaMeanCMAESMethod

Třída **SigmaMeanCMAESMethod** je potomek třídy **IPOPCMAESMethod**, kterou rozšiřuje o nastavení hodnot proměnných $\sigma^{(0)}$ a $\mathbf{m}^{(0)}$ po restartu. Jedná se tedy o rozšířenou metodu založenou na restartech a nárůstu populace σ-**m**-IPOP-CMA-ES.

Je to třídě která rozšiřuje svého předka **IPOPCMAESMethod** o nastavení jiné hodnoty proměnné $\sigma^{(0)}$ po restartu, než je inicializační hodnota (atribut **initialSigma** třídy CMAESMethod), ale na hodnotu podle rovnice 2.12. Dále pak $\mathbf{m}^{(0)}$ po restartu na hodnotu rovnice 2.11 (viz. popis v algoritmu 8).

Pro docílení tohoto efektu ve třídě **SigmaMeanIPOPCMAES** napíšu vlastní metodu **restart**, která bude volat metodu **restart** třídy **IPOPCMAESMethod** a bude nastavovat atribty po restaru podle algoritmu 8.

```
1 super.restart ()
2 bestWorstDist = MatrixUtils.createRealVector (this.theBestPoint.getPoint
  ().toArray ()).getDistance (this.theWorstPoint.getPoint ().toArray ());
3 super.xmean = MatrixUtils.createRealVector (super.theBestPoint.getPoint
  ().toArray ()).subtract (super.theWorstPoint.getPoint ().toArray ()).mapDivide (2);
4 this.attributeSigma = bestWorstDist/2;
```

Algorithm 8: Tělo metody **restart** ve třídě **SigmaMeanIPOPCMAESMethod**

Po nastavení všech atributů na počáteční hodnotu pomocí metody **restrat** třídou **IPOPCMAESMethod**, kde přenastavím hodnoty pomocí metody **restart** ve třídě **SigmaMeanIPOPCMAESMethod**.

3.3.6 Implementace zastavovacích podmínek

Při tvorbě zastavovacích podmínek jsem vycházel především z článku [18], který podrobně vysvětluje každou zastavovací podmínku. U podmínek **NoEffectAxis** a **NoEffectCoord** jsem použil kód z [16], kde jsou podmínky dobře popsané.

Všechny zastavovací podmínky jsou potomci rozhraní **CMAESStopCondition**, aby nebyli zaměněny s jinými zastavovacími podmínkami.

Všechny tyto podmínky mají atribut **use**, kterým můžu danou zastavovací podmínku vypnout, nebo zapnout. Při volání metody **isConditionMet** se ověřuje před vyhodnocením kritéria, zda není hodnota **use** nastavena na **true**.

3.3.6.1 Podmínka NoEffectAxis

Zastavovací podmínku NoEffectAxis jsem implementoval třídou **NoEffectAxisStopCondition**. Podmínka pracuje s vlastnostmi proměnných $\mathbf{D}^{(g)}, \mathbf{B}^{(g)}, \mathbf{m}^{(g)}$ a g. Všechny tyto proměnné předávám v metodě **setValues**.

Definice Zastavovací podmínka NoEffectAxis: Zastav jestliže 0.1 směrodatné odchylky v některé z komponent $\mathbf{C}^{(g)}$ přičtené k $\mathbf{m}^{(g)}$ nezmění jeho hodnotu

---

Je nutné, aby se postupně střídali všechny komponenty, čehož lze docílit tím, že spočítám zbytek po dělení z této generace po vydělení počtu možných komponent.

Podmínku lze vyjádřit tak, že porovnáváme aktuální střední hodnotu se součtem střední hodnoty s vybraným vlastním vektorem po tom, co vynásobíme 0.1 odmocniny z vlastního čísla. Matematicky lze vyjádřit výpočet směrodatné odchylky podle rovnice 3.2.

$$\mathbf{m}_{stddev}^{(g)} = \mathbf{m}^{(g)} + 0.1\mathbf{b}_i^{(g)}\sqrt{d_{ii}^{(g)}} \tag{3.2}$$

Z rovnice 3.2 lze vyjádřit podmínku **NoEffectAxis** kódem jako v algoritmu 9.

```
1 eigenIndex = currentGeneration mod N;
2 xmeanAfterAddingStdDev = xmean.add ( B.getColumnVector
  (eigenIndex).mapMultiply ( standardDeviationRatio * Math.sqrt ( D.getEntry
  (eigenIndex))));
3 if xmean.equals (xmeanAfterAddingStdDev) then
4 |   return true;
5 end
6 return false;
```

Algorithm 9: Implementace zastavovací podmínky NoEffectAxis

3.3.6.2 Podmínka NoEffectCoord

Zastavovací podmínku NoEffectCoord jsem implementoval třídou **NoEffectCoordStopCondition**. Podmínka pracuje s vlastnostmi proměnných $\mathbf{D}^{(g)}, \mathbf{m}^{(g)}$, a $\sigma^{(g)}$. Všechny tyto proměnné předávám v metodě **setValues**.

Definice Zastavovací podmínka NoEffectCoord: Zastav jestliže přičtení 0.2 směrodatné odchylky v nějaké i-té osy nezmění hodnotu $\mathbf{m}^{(g)}$ v dané souřadnici $m_i^{(g)}$

Podmínku lze vyjádřit podobně jako v případě NoEffectAxis porovnáním výpočtu odmocniny vektoru vlastních čísel násobených $0.2\sigma^{(g)}$. Matematicky lze výpočet přičtení směrodatné odchylky podle rovnice 3.3.

$$\mathbf{m}_{stddev}^{(g)}\mathbf{m}^{(g)} + 0.2\sigma^{(g)}\sqrt{\mathbf{D}^{(g)}} \tag{3.3}$$

Z rovnice 3.3 lze vyjádřit podmínku **NoEffectCoord** kódem jako v algoritmu 10.

```
1 if D.mapSqrt().mapMultiply (atrributeSigma * 0.2).add (xmean).equals (xmean)
  then
2 |   return true;
3 end
4 return false;
```

Algorithm 10: Implementace zastavovací podmínky NoEffectCoord

3.3.6.3 Podmínka EqualFunValues

Zastavovací podmínku EqualFunValues jsem implementoval třídou **EqualFunValuesStopCondition**. Podmínka pracuje hodnotami $f\left(\mathbf{x}_{1:\lambda}^{(g)}\right)$ v $1 \ldots 10 + \lceil \frac{30n}{\lambda} \rceil$ generacích. Fitness hodnoty nejlepších jedinců v generacích ukládám do instance **ArrayList<Double>** a zároveň přičítám přidávanou hodnotu fitness k proměnné, tak abych zabránil nutnosti sčítat celý seznam uložených $10 + \lceil \frac{30n}{\lambda} \rceil$ fitness při každém vyhodnocení.

Definice Zastavovací podmínka EqualFunValuesStopCondition: Zastav jestliže $10 + \lceil \frac{30n}{\lambda} \rceil$ generacích je hodnota $\sum_{g=0}^{10+\lceil \frac{30n}{\lambda} \rceil} f\left(\mathbf{x}_{1:\lambda}^{(g)}\right)$

V atributu **currentGeneration** uchovávám právě probíhající generaci a jeho hodnotu inkrementuji pokaždé, když volám metodu **setValues**. Dále mám pak konstantu **historyDepth**, v níž uchovávám hodnotu počtu $10 + \lceil \frac{30n}{\lambda} \rceil$ uchovávaných generací. Atribut **currentSumOfHisotry** ukládá součet $10 + \lceil \frac{30n}{\lambda} \rceil$ posledních generací, nebo menší počet generací. Seznam **history** používám k uchovávání všech hodnot předaných metodou **setValues** a používám tento atribut k odečítání historie pokud počet generací přeskočí $10 + \lceil \frac{30n}{\lambda} \rceil$.

Při volání metody **setValues** předá metoda **setStopConditionParameters** fitness nejlepšího jedince $f\left(\mathbf{x}_{1:\lambda}^{(g)}\right)$ v generaci g. Pokud je $g < 10 + \lceil \frac{30n}{\lambda} \rceil$ (tzn. podmínka **currentGeneration** $\leq$ **historyDepth**) stačí pouze přidat hodnotu na seznam **history** a přičíst k

atributu **currentSumOfHistory** hodnotu $f\left(\mathbf{x}_{1:\lambda}^{(g)}\right)$. Pokud je ovšem počet generací větší, odečtu od hodnoty **currentSumOfHistory** první hodnotu v seznamu **history** a přičtu hodnotu $f\left(\mathbf{x}_{1:\lambda}^{(g)}\right)$.

Podmínku **EqualFunValues** lze vyjádřit kódem jako v algoritmu 11:

```
1 if currentGeneration <= historyDepth then
2 |   history.add (currentBestFitness);
3 |   currentSumOfHisotry += currentBestFitness;
4 end
5 else
6 |   currentSumOfHisotry += currentBestFitness- history.get ((currentGeneration- 1) mod
  |   historyDepth);
7 |   history.set (currentGeneration mod historyDepth, currentBestFitness);
8 end
```

Algorithm 11: Implementace zastavovací podmínky EqualFunValues

3.3.6.4 Podmínka ConditionCov

Poslední ze statických zastavovacích podmínek je **ConditionCov**, jejíž implementace se skrývá ve třídě **ConditionCovStopCondition**.

Definice Zastavovací podmínka ConditionCov: Zastav jestliže číslo podmíněnosti kovariační matice $\mathbf{C}$ je větší než $\kappa = 10^{14}$.

Pokud známe nejmenší a největší vlastní číslo matice $\mathbf{C}^{(g)}$, nemusíme počítat číslo podmíněnosti např. podle [28], ale stačí porovnat největší vlastní číslo s nejmenším vlastním číslem vynásobeným hodnotou κ. Pokud platí $d_{11} > \kappa d_{nn}$ je číslo podmíněnosti matice $\mathbf{C}^{(g)}$ také větší, než κ, neplatí-li podmíněnost je menší než κ

Podmínku **ConditionCov** lze vyjádřit kódem jako v algoritmu 12.

```
1 Arrays.sort (eigenvalues);
2 if eigenvalues [0] > maxMultiplyOfMinEigenvalue * eigenvalues [eigenvalues.length- 1]
  then
3 |   return true;
4 end
5 return false;
```

Algorithm 12: Implementace zastavovací podmínky ConditionCov

3.3.6.5 Podmínka TolX

Podmínka **TolX** není obecně platná pro všechny funkce, ale může být závislá na problému. Její atribut **tolX** je proto nutné nastavit na požadovanou hodnotu. Podmínka je ve třídě

TolXStopCondition.

Definice Zastavovací podmínka TolX: Zastav jestliže směrodatná odchylka normálního rozdělení je ve všech osách menší než hodnota **TolX** a jestliže hodnota $\sigma^{(g)}\mathbf{p}_c^{(g)}$ je ve všech osách menší než **TolX**.

První kritérium je splněno, je-li směrodatná odchylka ve všech osách menší než hodnota **TolX**. Toto lze matematicky vyjádřit tak, že jsou-li všechny prvky na hlavní diagonále kovarianční matice $\mathbf{C}^{(g)}$ vynásobené *step-size* v dané generaci $\sigma^{(g)}$ menší než **TolX**, podmínka je splněna.

Podmínku **TolX** lze vyjádřit kódem v jako v algoritmu 13.

```
1  pcAndSigma = pc.mapMultiply (attributeSigma).getData();
2  for i=1 to N do
3  |  if pcAndSigma [i] > tolX then
4  |  |  return false;
5  |  end
6  end
7  stdDevDistribution = diagC.mapMultiply (attributeSigma).getData(); for i=0 to N
   do
8  |  if stdDevDistribution [i] > tolX then
9  |  |  return false;
10 |  end
11 |  return true;
12 end
```

Algorithm 13: Implementace zastavovací podmínky TolX

3.3.6.6 Podmínka TolFunHistory

Podmínka **TolFunHistory** je stejně jako **TolX** závislá na řešeném problému. Podmínka je identická s **EqualFunValues** s rozdílem, že součet $10 + \lceil \frac{30n}{\lambda} \rceil$ generacích může být různý od 0 a podmínka je splněna, pokud je součet méně než atribut **tolFun**.

Kapitola 4

Experimenty

4.1 Informace k experimentům

Na algoritmu jsem provedl řadu experimentů. První z nich spočíval v porovnání algoritmu CMA-ES pro všechny dostupné metody v knihovně JCool, které ke dni 11. května 2011 k dispozici a totéž platí pro 35 tehdy dostupných testovacích funkcí, uvedených v prvním sloupci tabulky 4.1. Druhá série experimentů spočívala v zjištění průběhu u vybraných funkcí algoritmu CMA-ES.

4.1.1 Získávání informací o průběhu při porovnávání s ostatními metodami

Pro získávání informací o průběhu jednotlivých metod nebyla v knihovně původně žádná třída. Pro účely této práce jsem vytvořil vlastní solver ve třídě **StepSolver**, který implementoval rozhraní **Solver**. **StepSolver** dědí od třídy **StepStorage**. Třída **StepStorage** slouží k uchovávání mezivýsledků o nejlepších a nejhorších jedincích v jednotlivých generacích. V závislosti na typu telemetrie lze uchovávat dva druhy výsledků:

- Telemetrie **ValuePointListTelemetry**, nebo **ValuePointListColoredTelemetry** pak uchovávám nejlepšího i nejhoršího jedince jako výsledek z jedné generace.
- Telemetrie **ValuePoint**, pak uchovávám pouze jediného jedince (algoritmus pracuje pouze s jednou hodnotou v každé generaci) a to jako nejlepšího i jako nejhoršího jako výsledek z dané generace.

Ve třídě **StepSolver** jsem implementoval metodu **solve** tak, aby si v každém kroku ukládala průběh metody a zároveň ověřovala, zda nebyl překročen maximální počet 1000 iterací. Všechny algoritmy a testovací funkce volala metoda **ComparsionReporting**.

Po vypočtení výsledků každé z testovaných funkcí uvedených ve 4. sloupci tabulky 4.1 se vytvoří csv soubor a na každém řádku tohoto souboru je uložen průběh funkce pro jednotlivé metody. Pro vykreslení výsledků třída generuje Matlab skript v němž vykreslí všechny výsledky.

Při výpočtech funkce Hump, která selhala na výpočtu optimalizační metody **LevenbergMarquardtMethod**, proto příslušný graf ve své práci neuvádím.

Nárůst populace po restartu $\Delta\lambda$ je pro IPOP-CMA-ES algoritmus nastavena na $\Delta\lambda = 2$.

Výsledky z těchto experimentů jsou na přiloženém CD v */experiments/comparsion/*. Třídy které jsem vytvořil pro tento problém (**StepSolver**, **StepStorage** a **ComparsionReporting**) nejsou součástí přiloženého kódu knihovny a jsou rovněž na přiloženém CD ve složce */reporting_ code/*.

4.1.2 Diskuze o výsledcích experimentů

Ve většině testovacích funkcí dosáhl algoritmus CMA-ES lepších výsledků než konkurenční metody. Místy se sice stalo, že algoritmus uvázl v lokálním minimu, jako například v případě Levyho 3 a Levyho 5 funkce, kde algoritmus CMA-ES skončil výpočet po 20 iteracích, protože byla splněna zastavovací podmínka **EqualFunValues**. Algoritmus IPOP-CMA-ES vykompenzoval horší výsledky CMA-ES hned po prvním restartu, kdy se dostal do globálního optima, v němž proběhlo pro Levyho 3 funkci 7 restartů a pro Levyho 5 funkci 6 restartů.

Na většině průběhů fitness funkcí je vidět, kdy se algoritmus IPOP-CMA-ES restartoval. Restart se projevuje náhlým nárůstem, resp. poklesem průběhu fitness funkce. Velmi dobře je to vidět například na Langermannově funkci, kde proběhlo velmi rychle za sebou po cca. 10 generacích 7 restartů. Po restartu se algoritmus vrátil k původnímu globálnímu optimu. Jediná metoda, která podává stejně dobré výsledky je API. Naopak u některých metod měl IPOP-CMA-ES problémy s velkým počtem restartů, což ukazuje Shekelova funkce, kde je po 20. generacích zaznamenáno značné množství restartů a algoritmus osciluje.

I přes to, že metoda je metoda vhodná spíše na funkce unimodální, výsledky ukazují, že metoda nemá problémy ani s multimodálními funkcemi, jako je Ackleyova nebo Rastriginova funkce.

4.1.3 Získávání informací o průběhu výpočtu algoritmu CMA-ES

Pro získávání výsledků jsem vytvořil speciální třídu odvozenou od třídy **PureCMAESMethod**, protože atributy, které jsem chtěl získat ($\mathbf{m}^{(g)}$ a $\mathbf{C}^{(g)}$ jsou přístupné pouze potomkům).

O výstupy z kroků metody **PureCMAESMethod** se stará třída **PureCMAESMethodReporting**, která ve své členské metodě **measure** volá 1000 krát metodu předka **PureCMAESMethod optitmize** a po každé generaci uloží hodnoty **C** a **xmean**.

Po volání každé fitness funkce z testovacích funkcí uvedených ve 3 sloupci tabulky 4.1 se výsledky uloží do Matlab skriptu do odpovídajících vektorů.

Výsledky z těchto experimentů jsou na přiloženém CD ve složce */experiments/cmasteps/*. Třída, kterou jsem vytvořil pro tento problém (**PureCMAESMethodReporting**), není součástí přiloženého kódu knihovny. Třídy jsou uloženy na přiloženém CD ve složce */reporting_ code/*.

Současně jsem provedl experimenty nad všemi metodami a porovnal výsledné hodnoty dosažených fitness všech algoritmů a pro všechny testovací funkce pro 1000 a 10000 iterací. Data jsou ve složce */experiments/finalresults*.

Funkce $f(\mathbf{x})$	Průběh CMA-ES	Průběh min f s ostatními algoritmy
Ackley	C.1	B.1
Beale	C.2	B.2
Bohachevsky	C.3	B.3
Booth	C.4	B.4
Branin	C.5	B.5
Colville	-	B.6
DixonPrice	-	B.7
Easom	C.8	B.8
GoldsteinPrice	-	B.9
Griewangk	-	B.10
Hartmann	-	B.11
Himmelblau	C.10	B.12
Hump	C.11	-
Langermann	-	B.13
Levy	-	B.16
Levy3	-	B.14
Levy5	-	B.15
Matyas	C.12	B.17
Michalewicz	-	B.17
Perm	-	B.19
Powell	-	B.20
PowerSum	-	B.21
Rana	-	B.22
Rastrigin	C.13	B.23
Rosenbrock	C.14	B.24
Schwefel	C.15	B.25
Shekel	-	B.26
Shubert	-	B.27
Sphere	C.16	B.28
Trid	C.17	B.29
Whitley	-	B.30
Zakharov	C.18	B.31
Constant	C.6	-

Tabulka 4.1: Testovací funkce, nad nimiž byli provedeny experimenty.
V prvním sloupci je uveden název odpovídající funkce. Druhý sloupec obsahuje číslo obrázku na němž je vyznačen průběh výpočtu proměnných algoritmu CMA-ES a ve čtvrtém sloupci porovnání všech dostupných algoritmů v knihovně na odpovídající testovací funkci. V případě, že je v tabulce symbol „-“ znamená to, že výpočet byl zbytečný (konstantní funkce), nebo se průběhu vyskytla nějaká chyba (funkce Hump).

Kapitola 5

Závěr

Úkolem mé práce bylo nastudovat, implementovat a provést experimenty s algoritmem CMA-ES. Předpokladem pro vytvoření algoritmu byla důkladná analýza značně komplikovaného popisu založeného na znalostech i pokročilejších partií lineární algebry. Samotná analýza zabrala přes půl roku, což odpovídá i rozsahu teoretické části.

V první kapitole své práce se zabývám obecnými vlastnostmi evolučních algoritmů, kde zobecňuji jednotlivé skupiny evolučních algoritmů včetně jejich operátorů. Podrobněji rozebírám rovněž taxonomii evolučních strategií podle jejich práce s populací. V této části zařadím algoritmus CMA-ES do odpovídající třídy evolučních výpočtů. Dále pak definuji podstatné matematické pojmy jenž jsou nutné pro řádné pochopení algoritmu.

V analýze algoritmu podrobně rozebírám všechny operace, které algoritmus provádí. Vysvětluji jednotlivé kroky za použití pojmů, které jsem definoval v úvodní kapitole. Dále pak zmiňuji známé varianty, které vylepšují vlastnosti čistého CMA-ES, jsou to: IPOP-CMA-ES, LS-CMA-ES a ACTIVE-CMA-ES. Existující variantu IPOP-CMA-ES jsem rozšířil o uchovávání některých vlastností. Variantu IPOP-CMA-ES jsem rozšířil o historii nejlepšího a nejhoršího jedince, jejichž hodnoty využívám pro nastavení některých proměnný algoritmu IPOP-CMA-ES po restartu. Tento algoritmus jsem nazval σ-**m**-IPOP-CMA-ES. Rovněž podrobně zanalyzuji zastavovací podmínky, invarianci, vztah s inverzním Hesisánem a Newtonovou metodou.

Do knihovny JCool jsem implementoval čistý algoritmus CMA-ES, IPOP-CMA-ES a svojí variantu σ-**m**-IPOP-CMA-ES. Protože jsou algoritmy provázané společnými výpočty, vytvořil jsem třídu zobecněného algoritmu CMA-ES k němuž jsem postupně přidával další třídy implementující další algoritmy. Koncepce třídní hierarchie je řešena tak, aby v případě nutnosti bylo možné knihovnu snadno rozšířit o další varianty.

Metodu jsem podrobil řadě zajímavých experimentů. Mezi nejpodstatnější považuji porovnání algoritmu se všemi ostatními algoritmy, které jsou součástí knihovny na všech testovacích funkcích. Algoritmus potvrdil svou kvalitu při hledání optim oproti ostatním metodám a na většině testovacích funkcí našel optimum v nejmenším počtu iterací. V případě, že CMA-ES nedosáhl dobrých výsledků, dosahoval jsem dobrých výsledků s rozšířením IPOP-CMA-ES. K další řadě experimentů jsem si vybral menší množinu testovacích funkcí, nad níž jsem ověřil konvergenci algoritmu na unimodálních i multimodálních funkcích. Proměnné algoritmu jsem v jednotlivých generacích vizualizoval, abych ověřil funkčnost.

Problematika algoritmu CMA-ES je mnohem komplikovanější, a to co zmiňuji v teorii, je pouze zlomek toho, co lze o práci napsat. Za nejkomplikovanější považuji vzájemné interakce mezi proměnnými a vybrat z těchto výpočtů to nejzajímavější.

Danou prací jsem položil základ algoritmu CMA-ES, včetně jeho nových variant. Doufám, že mé výsledky přinesou osobám, které se budou o problematiku CMA-ES zajímat základní poznatky, z nichž lze v budoucnu vyvíjet další velmi zajímavé algoritmy.

Literatura

[1] M.-Mikkel. et al. Andersen. Commons Math, 2011. `http://commons.apache.org/math/`, stav z 25. 5. 2011.

[2] A. Auger and N. Hansen. A restart CMA evolution strategy with increasing population size. In *The 2005 IEEE International Congress on Evolutionary Computation (CEC'05)*, volume 2, pages 1769–1776, 2005.

[3] A. Auger, M. Schoenauer, and Nicolas. Vanhaecke. LS-CMA-ES: A Second-Order Algorithm for Covariance Matrix Adaptation. In *PPSN*, pages 182–191, 2004.

[4] Th. Bäck and Schwefel H. P. Overview of Evolutionary Algorithms for Parameter Optimization. *Evolutionary Computation*, 1993. `http://192.12.12.16/events/workshops/images/7/7e/Back-ec93.pdf`.

[5] T. Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, 1996.

[6] Ch. Gagné and M. Parizeau. Genericity in Evolutionary Computation Software Tools: Principles and Case-Study. *International Journal on Artificial Intelligence Tools*, 15:173–194, 2006.

[7] N. Hansen. Compilation of Results on the 2005 CEC Benchmark Function Set. `http://www.lri.fr/~hansen/cec2005compareresults.pdf`, stav z 25. 5. 2011.

[8] N. Hansen. Poslední implementace čistého algoritmu CMA-ES verze 2.55, 2003. `http://www.lri.fr/~hansen/cmaes050316.m`, stav z 25. 5. 2011.

[9] N. Hansen. První implementace algoritmu CMA-ES, 2003. `http://www.lri.fr/~hansen/cmaes030207.m`, stav z 25. 5. 2011.

[10] N. Hansen. První implementace IPOP-CMA-ES algoritmu verze 2.35, 2003. `http://www.lri.fr/~hansen/cmaes050316.m`, stav z 25. 5. 2011.

[11] N. Hansen. The CMA Evolution Strategy: A Comparing Review. In *Towards a New Evolutionary Computation*, volume 192, chapter 4, pages 75–102. Springer Berlin Heidelberg, 2006.

[12] N. Hansen. Implementace CMA-ES, IPOP-CMA-ES a ACTIVE-CMA-ES algoritmů verze 3, 2008. `http://www.lri.fr/~hansen/cmaes.m`, stav z 25. 5. 2011.

[13] N. Hansen. Zkrácená implementace v Matalb CMA-ES algoritmu, 2008. http://www.lri.fr/~hansen/purecmaes.m, stav z 25. 5. 2011.

[14] N. Hansen. ANSI C implementace CMA-ES algoritmu, 2010. http://www.lri.fr/~hansen/cmaes_c.tar.gz, stav z 25. 5. 2011.

[15] N. Hansen. Implementace algoritmu CMA-ES v jazyce Python, 2010. http://www.lri.fr/~hansen/cma.py, stav z 25. 5. 2011.

[16] N. Hansen. Zkrácená implementace algoritmu CMA-ES v jazyce Python, 2010. http://www.lri.fr/~hansen/barecmaes.py, stav z 25. 5. 2011.

[17] N. Hansen. CMA-ES Source Code, 2011. " http://www.lri.fr/~hansen/cmaes_inmatlab.html, stav z 25. 5. 2011".

[18] Nikolaus Hansen. The CMA Evolution Strategy: A Tutorial. 2011. http://www.lri.fr/~hansen/cmatutorial.pdf, stav z 25. 5. 2011.

[19] J. Heitkötter and D. Beasley. *Hitch Hiker's Guide to Evolutionary Computation.* 1993. http://www.aip.de/~ast/EvolCompFAQ/, stav z 25. 5. 2011.

[20] Ch. Igel, T. Glasmachers, and V. Heidrich-Meisner. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008. http://sourceforge.net/projects/shark-project/, stav z 25. 5. 2011.

[21] Ch. Igel, N. Hansen, and S. Roth. Covariance Matrix Adaptation for Multi-objective Optimization. *Evolutionary Computation*, 15:1–28, květen 2007.

[22] G.A. Jastrebski and D.V. Arnold. Improving Evolution Strategies through Active Covariance Matrix Adaptation. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 2814 –2821, 2006.

[23] Maarten Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving Objects: A General Purpose Evolutionary Computation Library. *Artificial Evolution*, 2310:829–888, 2002. http://www.lri.fr/~marc/EO/EO-EA01.ps.gz, stav z 25. 5. 2011.

[24] R. Mařík. Parciální derivace, 2009. http://user.mendelu.cz/marik/prez/parc-der-cz.pdf, stav z 25. 5. 2011.

[25] V. et al. Mařík. *Umělá inteligence (3).* Academia, 2001.

[26] Ch. L. Muller, G. Ofenbeck, B. Baumgartner, and I. F. Sbalzarini. pCMALib Library, 2010. http://www.mosaic.ethz.ch/Downloads/pCMALib, stav z 25. 5. 2011.

[27] P. Olšák. *Úvod do algebry, zejména lineární.* Nakladatelství ČVUT Praha, 2007.

[28] G. Strang. *Linear Algebra and Its Applications.* Brooks Cole, 1988.

[29] Eric W. Weisstein. Maximum Absolute Row Sum Norm. http://mathworld.wolfram.com/MaximumAbsoluteRowSumNorm.html, stav z 25. 5. 2011.

[30] Wikipedia. CMA-ES — Wikipedia, The Free Encyclopedia, 2011. http://en.wikipedia.org/w/index.php?title=CMA-ES&oldid=428587755, stav z 25. 5. 2011.

[31] Wikipedia. Covariance matrix — Wikipedia, The Free Encyclopedia, 2011. http://en.wikipedia.org/wiki/Covariance_matrix, stav z 25. 5. 2011.

[32] Wikipedia. Crossover (genetic algorithm) — Wikipedia, The Free Encyclopedia, 2011. http://en.wikipedia.org/wiki/Crossover_(genetic_algorithm),.

[33] Wikipedia. Matrix norm — Wikipedia, The Free Encyclopedia, 2011. http://en.wikipedia.org/wiki/Matrix_norm, , stav z 25. 5. 2011.

[34] Wikipedia. Newton's method in optimization — Wikipedia, The Free Encyclopedia, 2011. http://en.wikipedia.org/wiki/Newton's_method_in_optimization, stav z 25. 5. 2011.

Příloha A

Popis atributů třídy CMAESMethod

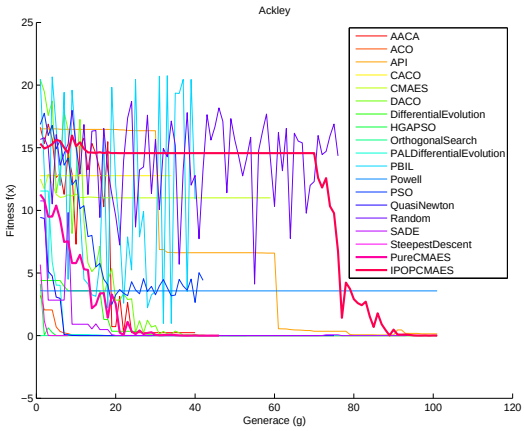
Název proměnné	Parametr	Popis
`B`	$\mathbf{B}^{(g)}$	Matice vlastních vektorů $\mathbf{C}^{(g)}$
`bestValueInCurrentGeneration`	$\mathbf{x}_{1:\lambda}^{(g)}$	Nejlepší vektor z generace g
`C`	$\mathbf{C}^{(g)}$	Kovarianční matice $\mathbf{C}^{(g)}$
`cCumulation`	c_c	Kumulační konstanta
`chiN`	$\|\mathcal{N}(\mathbf{0},\mathbf{I})\|$	Odhad $\mathcal{L}_2$ normy $\mathcal{N}(\mathbf{0},\mathbf{I})$
`consumer`	consumer	Příjemce průběhu
`countEval`	countEval	Počet vyhodnocení fitness f
`cRank1`	c_1	Konstakta *rank-1-update*
`cSigma`	c_σ	Konstanta pro výpočet $\sigma^{(g)}$
`currentGeneration`	g	Stávající generace g
`D`	$\mathbf{D}^{(g)}$	Vektor vlastních čísel $\mathbf{D}^{(g)}$
`dampingForSigma`	d_σ	Tlumící konstanta d_σ
`eigenEval`	eigenEval	Počet rozkladů $\mathbf{C}^{(g)}$
`function`	function	Fitness funkce f
`gaussianGenerator`	$\mathcal{N}(0,1)$	Generátor normálního rozdělení
`initialSigma`	$\sigma^{(0)}$	Počáteční hodnota $\sigma^{(0)}$
`inverseAndSqrtC`	$\mathbf{C}^{-\frac{1}{2}(g)}$	$\mathbf{C}^{-\frac{1}{2}(g)}$ pro výpočet $\mathbf{p}_\sigma^{(g)}$
`lambda`	λ	Velikost populace λ
`max`	max	Maximální $\forall i \in N\ x_i^{(O)} \leq \max$
`min`	min	Maximální $\forall i \in N\ x_i^{(O)} \geq \max$
`mu`	μ	Počet vybraných jedinců μ
`muEff`	μ_{eff}	Efektivní hodnota selekce
`N`	N, n	Vstupní rozměr funkce f
`pCumulation`	$\mathbf{p}_c^{(g)}$	evoluční cesta pro *rank-1-update*
`pSigma`	$\mathbf{p}_\sigma^{(g)}$	evoluční cesta pro $\sigma^{(g)}$
`sigma`	$\sigma^{(g)}$	Velikost kroku
`telementry`	telemetry	Všichni jedinci z g
`theBestPoint`	theBestPoint	Nejlepší jedinec ze všech generací
`theWorstPoint`	theWorstPoint	Nejhorší jedinec ze všech generací
`weights`	$\mathbf{w}$	Váhy
`xmean`	$\mathbf{m}^{(g)}$	Vážená střední hodnota selekce

Tabulka A.1: Popis atributů třídy CMAESMethod jejich případné odpovídající proměnné algoritmu CMA-ES

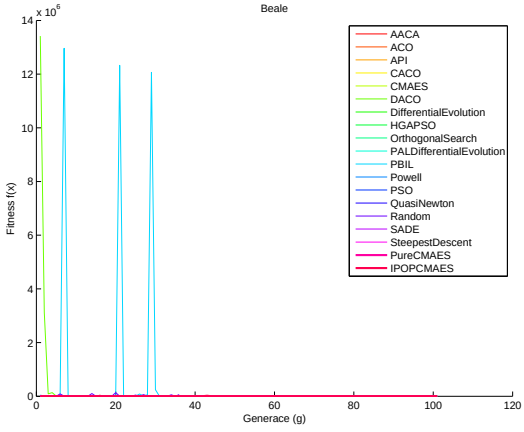
Příloha B

Grafy průběhů fitness

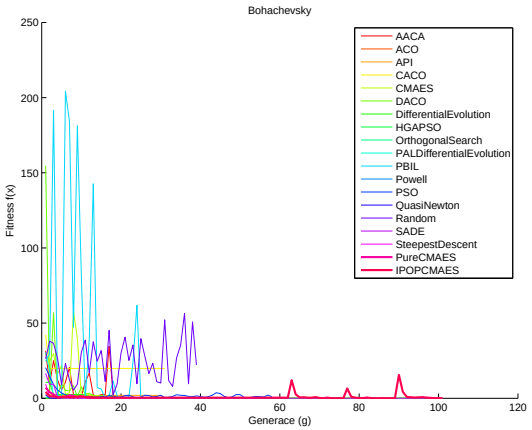
V následující příloze jsou grafy průběhu fitness pro všechny testovací funkce, které byli v knihovně k dispozici. V každém grafu je vyznačen odpovídající barvou algoritmus hledající minimum.



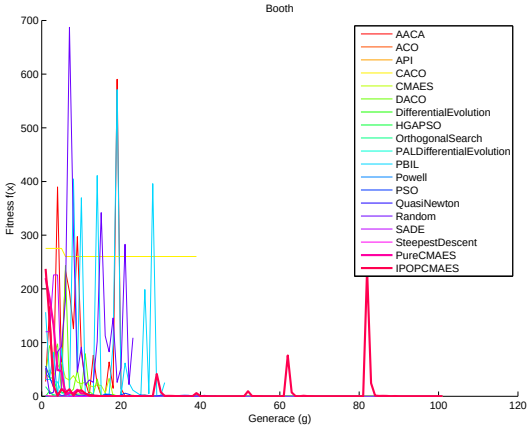
Obrázek B.1: Průběh fitness pro Ackleyovu funkci všech algoritmů.



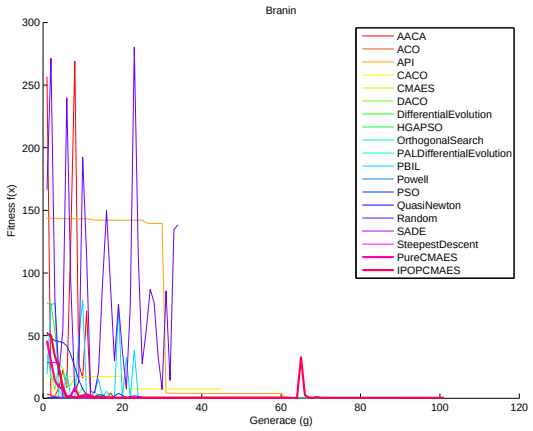
Obrázek B.2: Průběh fitness pro Bealeho funkci všech algoritmů.



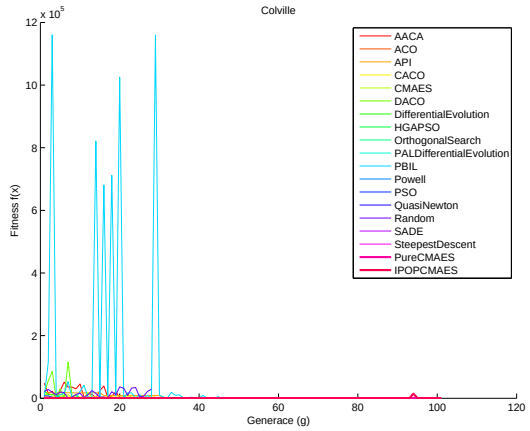
Obrázek B.3: Průběh fitness pro Bohachevskyho funkci všech algoritmů.



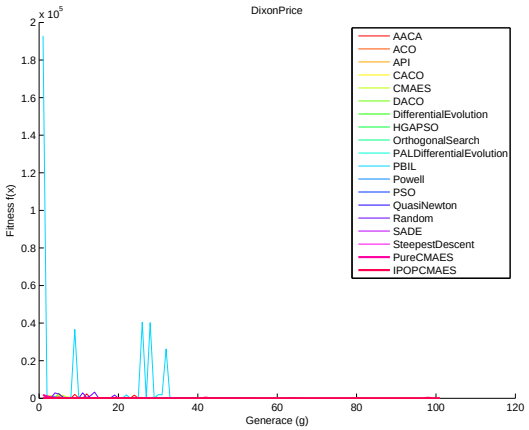
Obrázek B.4: Průběh fitness pro Booth funkci všech algoritmů.



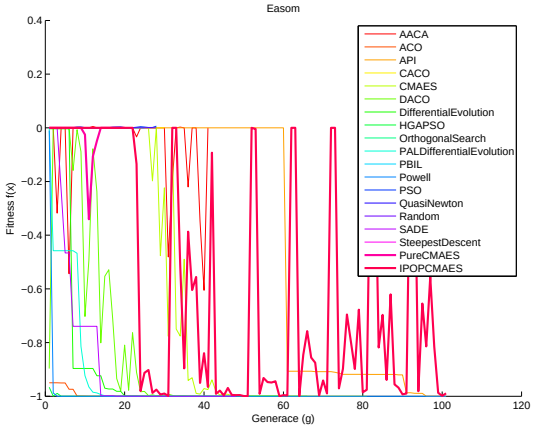
Obrázek B.5: Průběh fitness pro Branin funkci všech algoritmů.



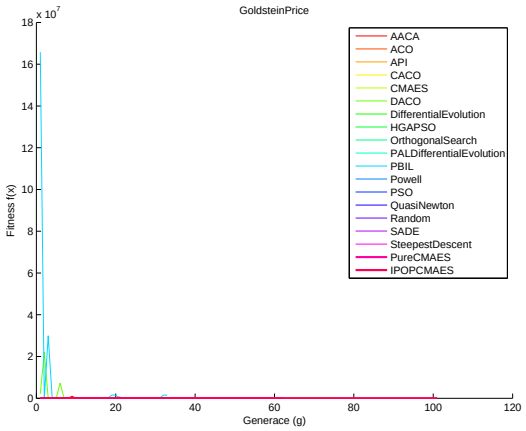
Obrázek B.6: Průběh fitness pro Colville funkci všech algoritmů.



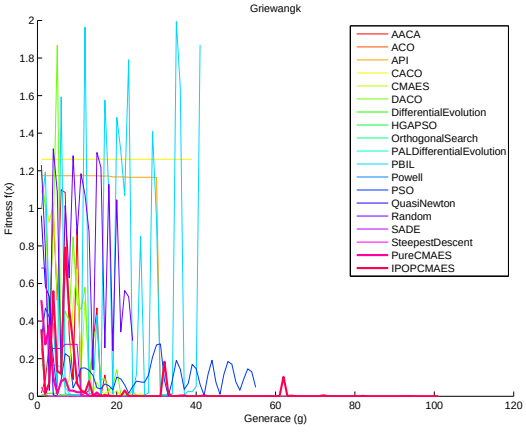
Obrázek B.7: Průběh fitness pro Dixon-Priceovu funkci všech algoritmů.



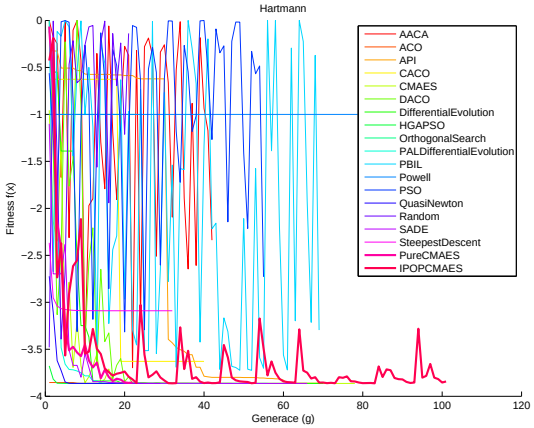
Obrázek B.8: Průběh fitness pro Easomovu funkci všech algoritmů.



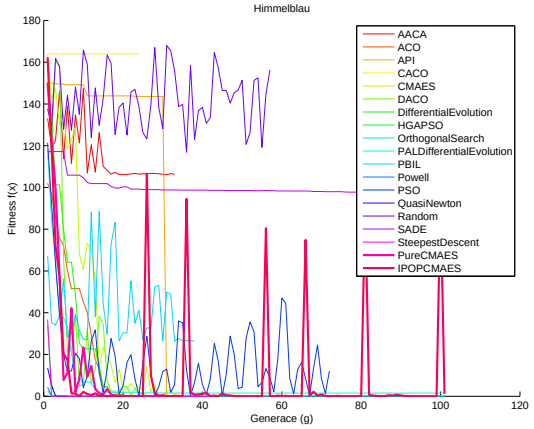
Obrázek B.9: Průběh fitness pro Goldstein-Price funkci všech algoritmů.



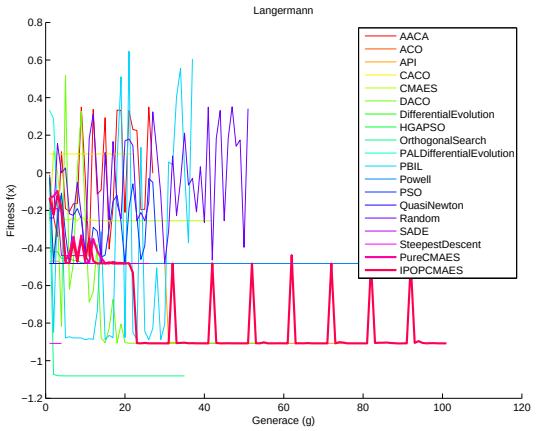
Obrázek B.10: Průběh fitness pro Griewangkovu funkci všech algoritmů.



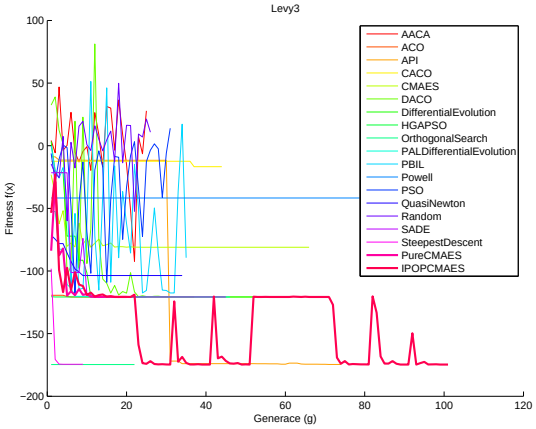
Obrázek B.11: Průběh fitness pro Hartmannovu funkci všech algoritmů.



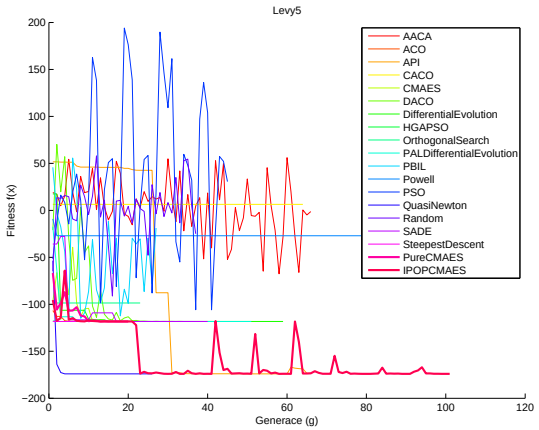
Obrázek B.12: Průběh fitness pro Himmelblauovu funkci všech algoritmů.



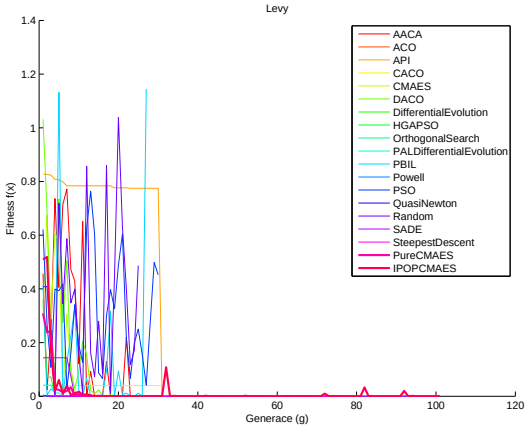
Obrázek B.13: Průběh fitness pro Langermannovu funkci všech algoritmů.



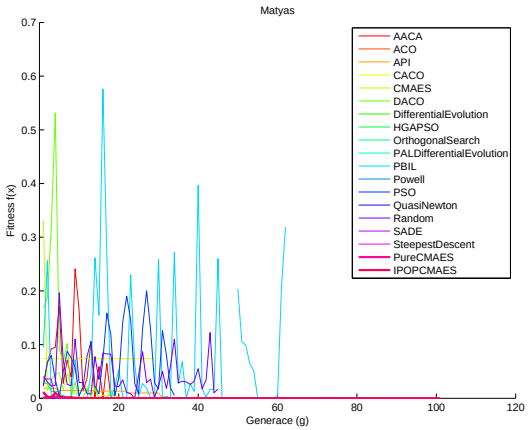
Obrázek B.14: Průběh fitness pro Levyho 3 funkci všech algoritmů.



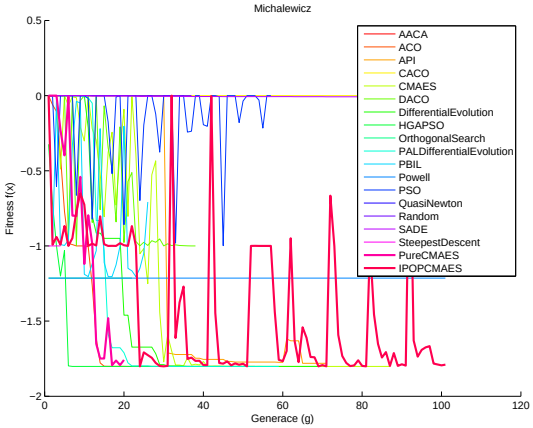
Obrázek B.15: Průběh fitness pro Levyho 5 funkci všech algoritmů.



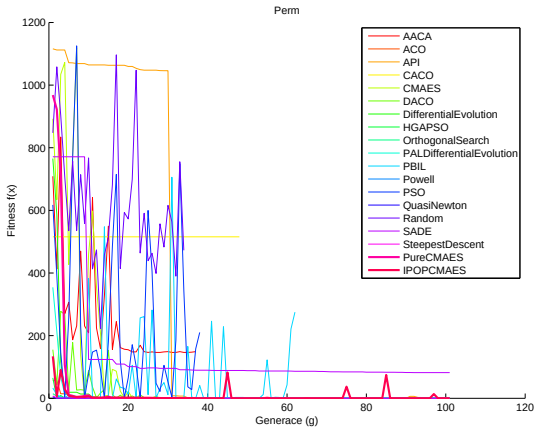
Obrázek B.16: Průběh fitness pro Levy funkci všech algoritmů.



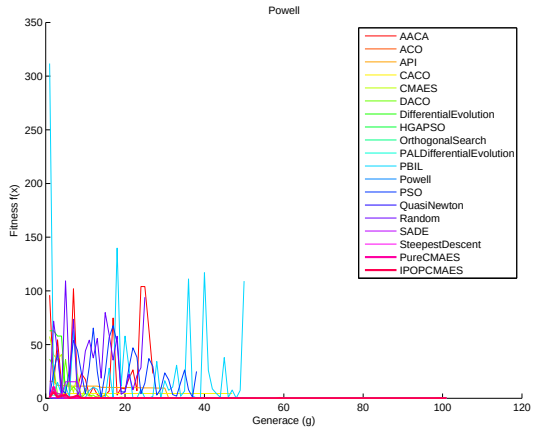
Obrázek B.17: Průběh fitness pro Matyasovu funkci všech algoritmů.



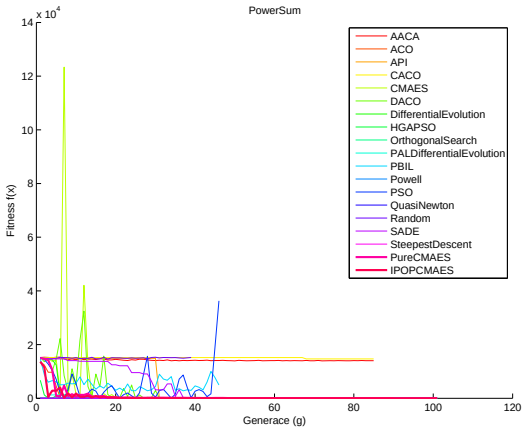
Obrázek B.18: Průběh fitness pro Michalewiczovu funkci všech algoritmů.



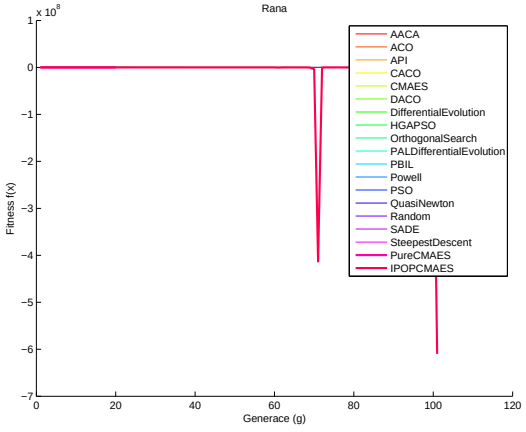
Obrázek B.19: Průběh fitness pro Perm funkci všech algoritmů.



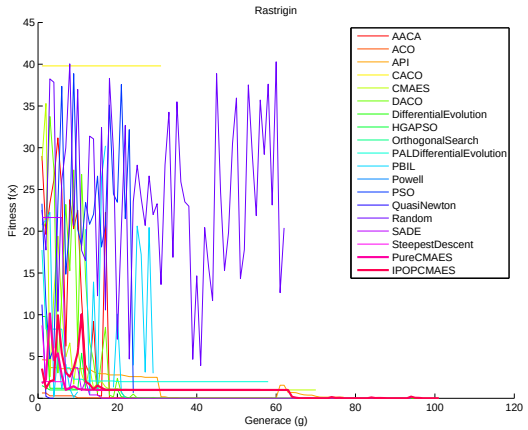
Obrázek B.20: Průběh fitness pro Powellovu funkci všech algoritmů.



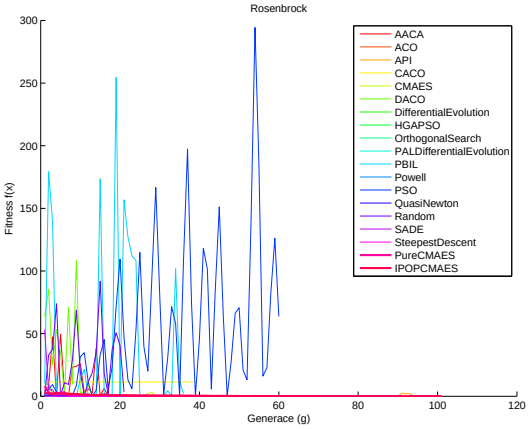
Obrázek B.21: Průběh fitness pro kvadratickou funkci všech algoritmů.



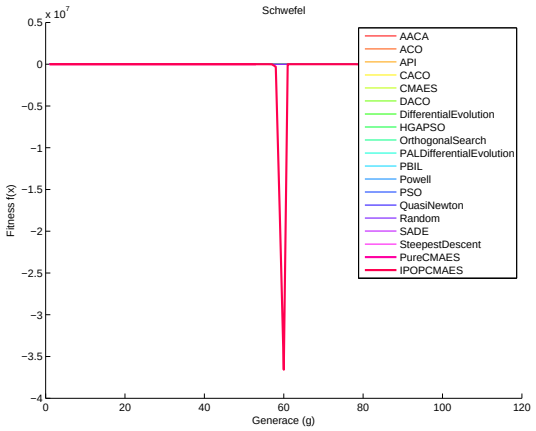
Obrázek B.22: Průběh fitness pro Ranaovu funkci všech algoritmů.



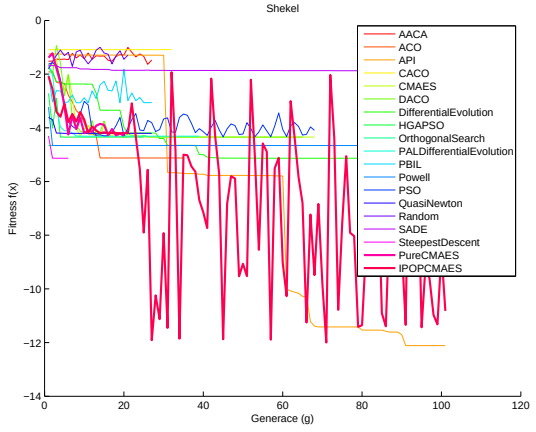
Obrázek B.23: Průběh fitness pro Rastrigin funkci všech algoritmů.



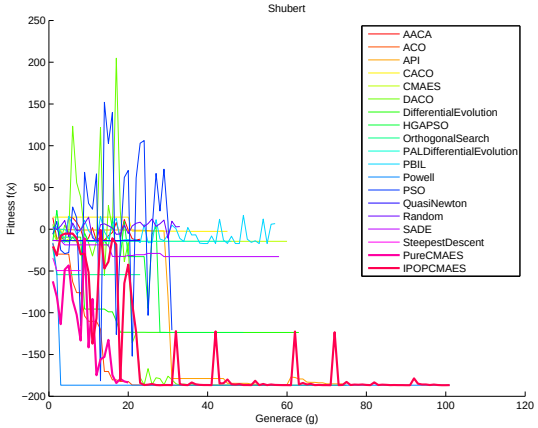
Obrázek B.24: Průběh fitness pro Rosenbrockovu funkci všech algoritmů.



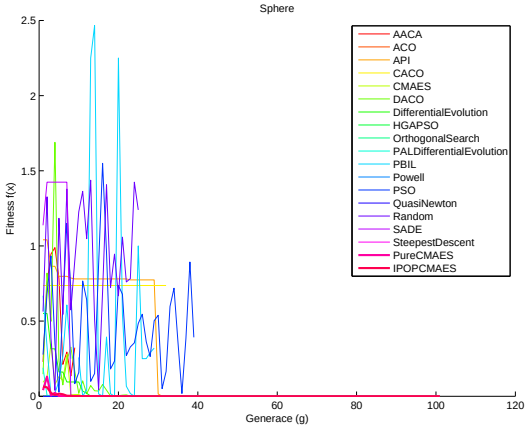
Obrázek B.25: Průběh fitness pro Schwefelovu funkci všech algoritmů.



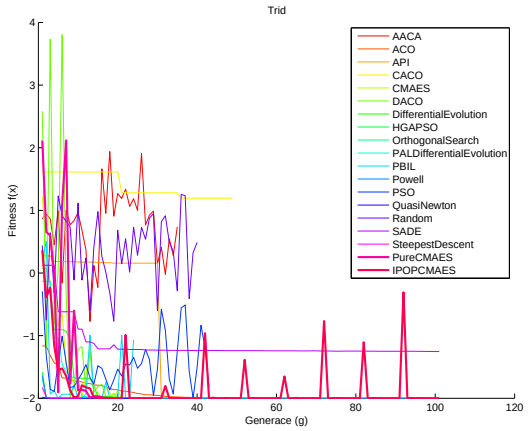
Obrázek B.26: Průběh fitness pro Shekelovu funkci všech algoritmů.



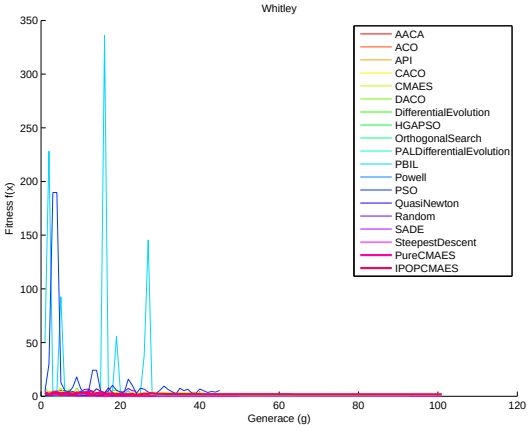
Obrázek B.27: Průběh fitness pro Shubertovu funkci všech algoritmů.



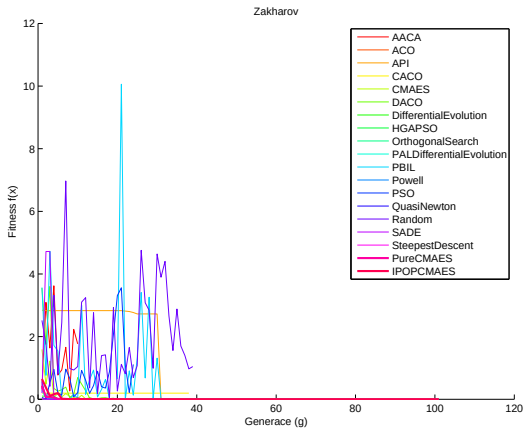
Obrázek B.28: Průběh fitness pro Sphere funkci všech algoritmů.



Obrázek B.29: Průběh fitness pro Tridovu funkci všech algoritmů.



Obrázek B.30: Průběh fitness pro Whitleyovu funkci všech algoritmů.

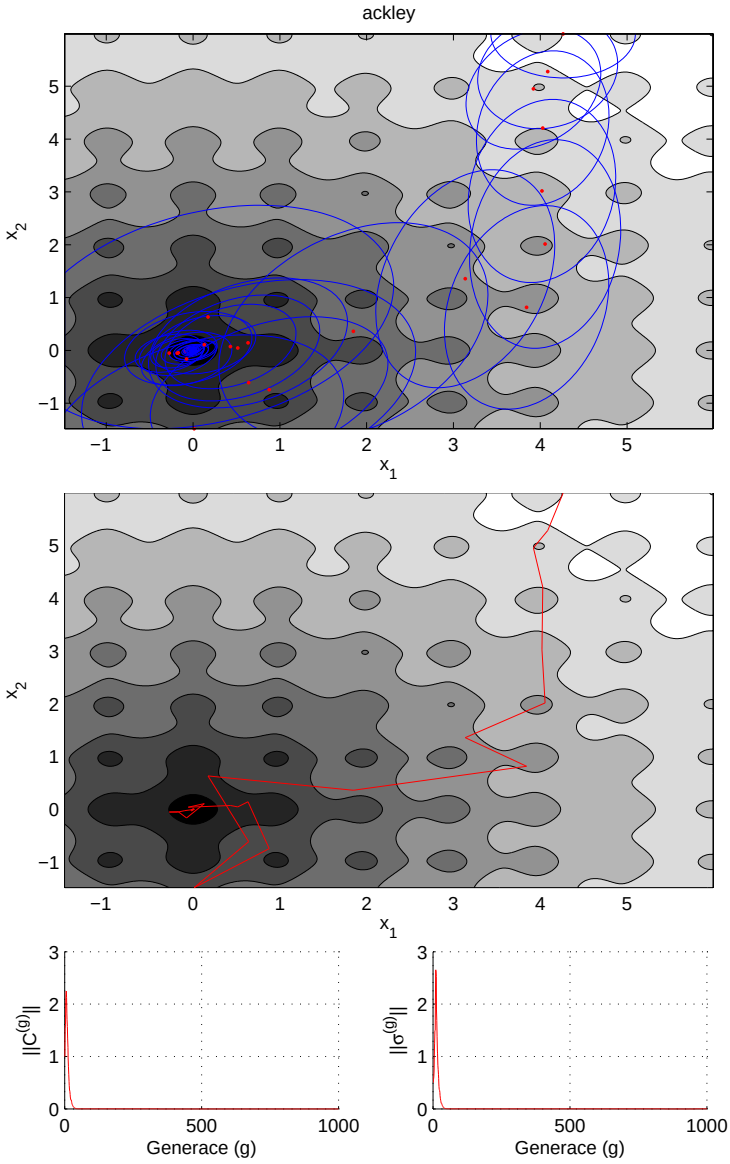


Obrázek B.31: Průběh fitness pro Zakharov funkci všech algoritmů.

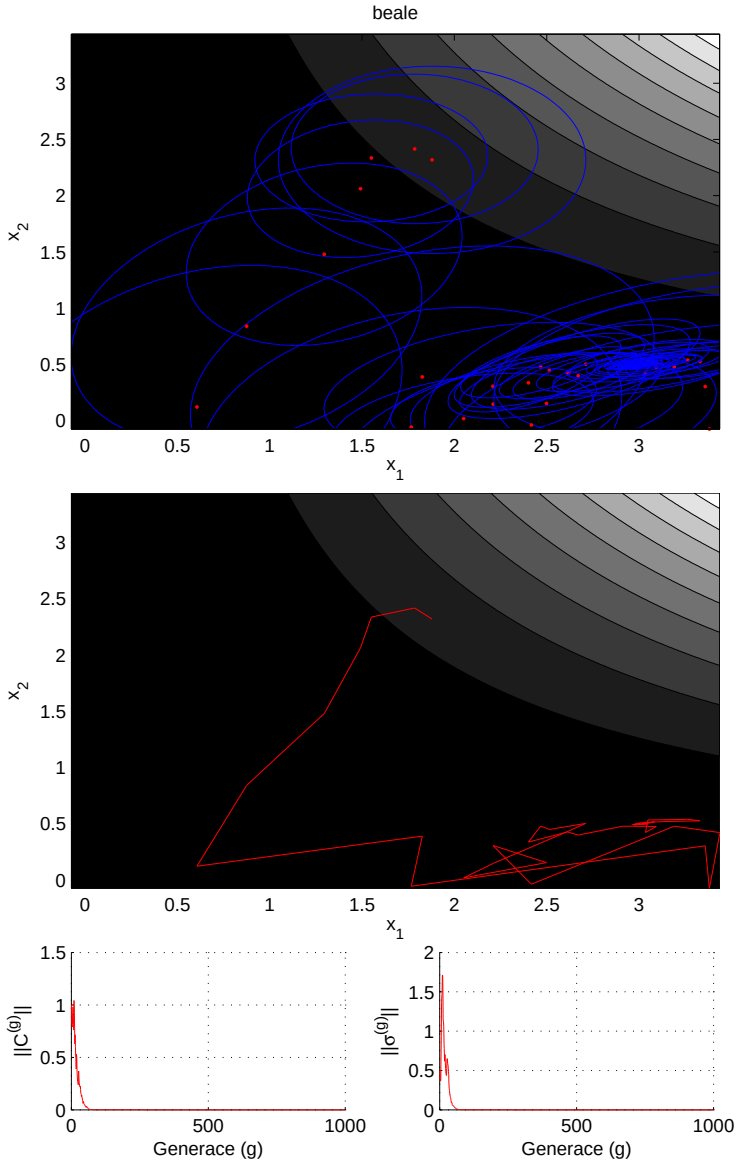
Příloha C

Kroky algoritmu CMA-ES

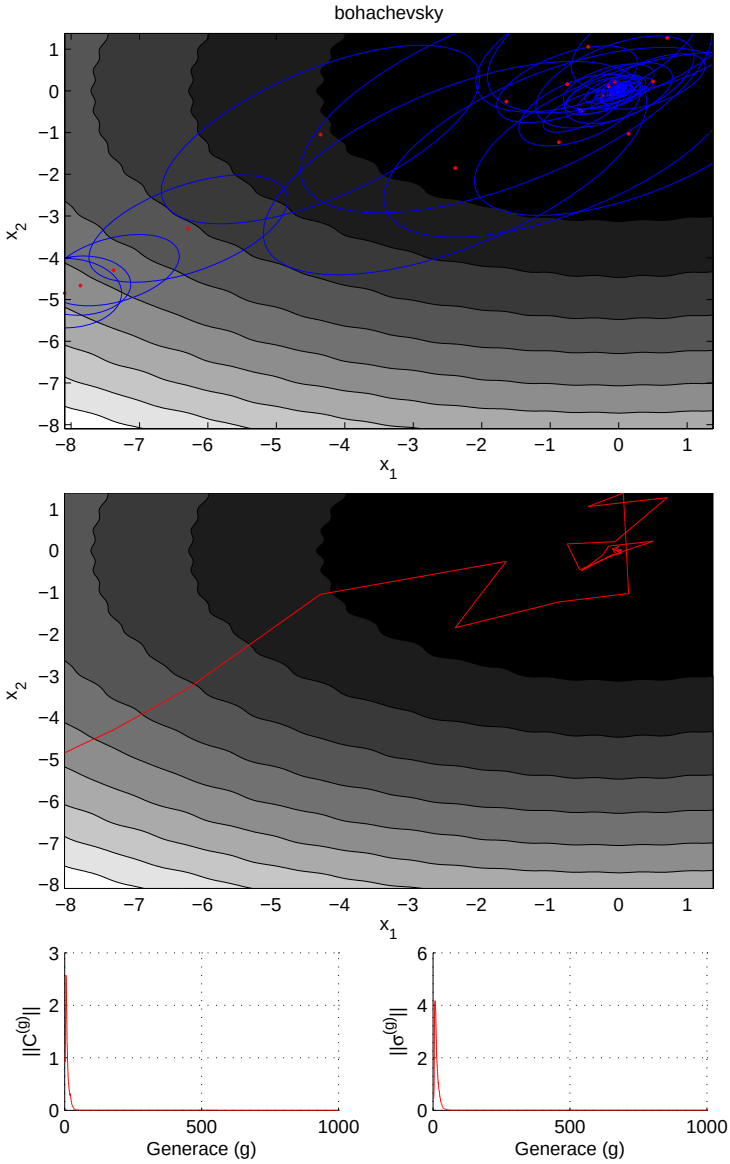
Na následujících obrázcích jsou uvedeny grafy průběhu vybraných testovacích funkcí algoritmu CMA-ES. Grafy zobrazují pohyb střední hodnoty $\mathbf{m}^{(g)}$ a $\sigma^{(g)}\mathbf{C}^{(g)}$.



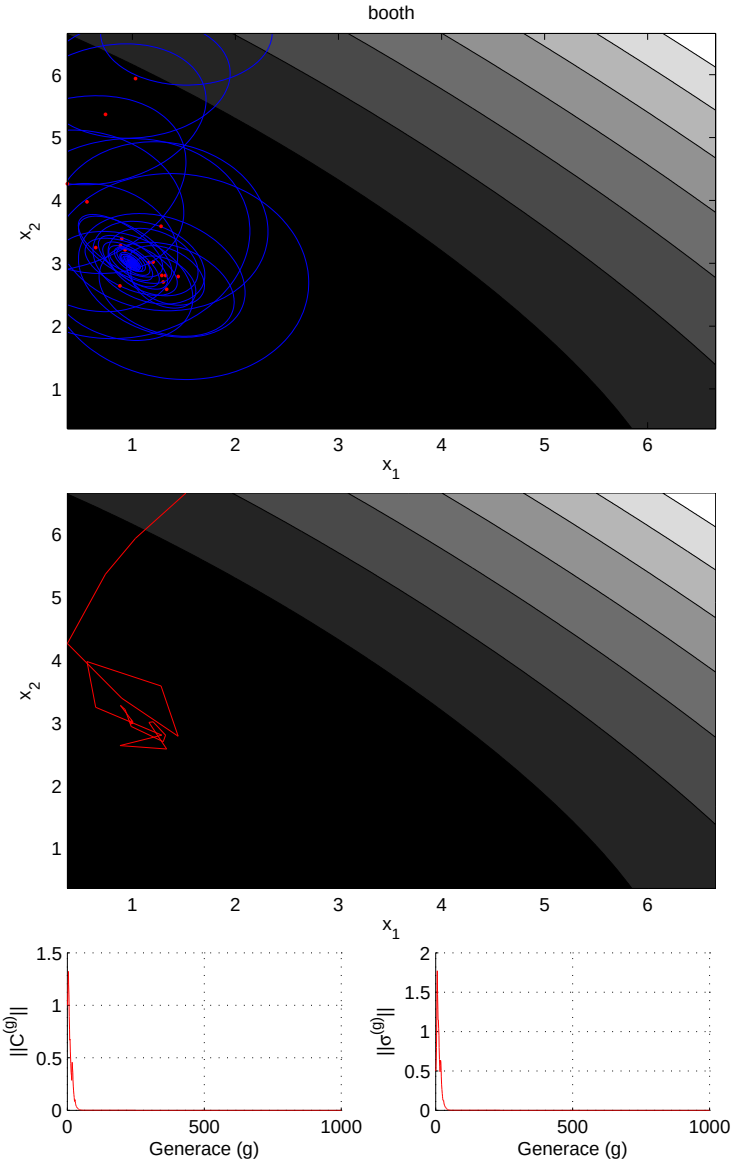
Obrázek C.1: Průběh algoritmu pro Ackleyovu funkci ve dvou rozměrech
Globální minimum je $\mathbf{x}^* = (0, 0)$, $f(\mathbf{x}^*) = 0$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



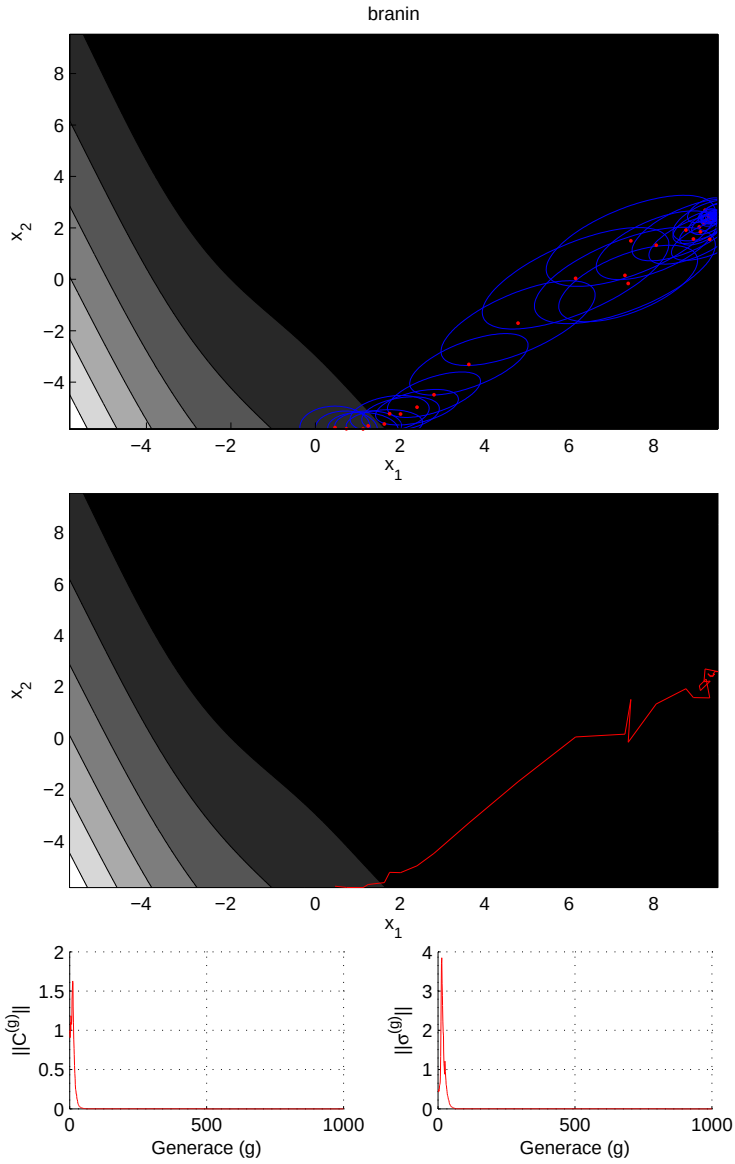
Obrázek C.2: Průběh algoritmu pro Bealeho funkci ve dvou rozměrech
Globální minimum je $\mathbf{x}^* = (3, 0.5)$, $f(\mathbf{x}^*) = 0$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



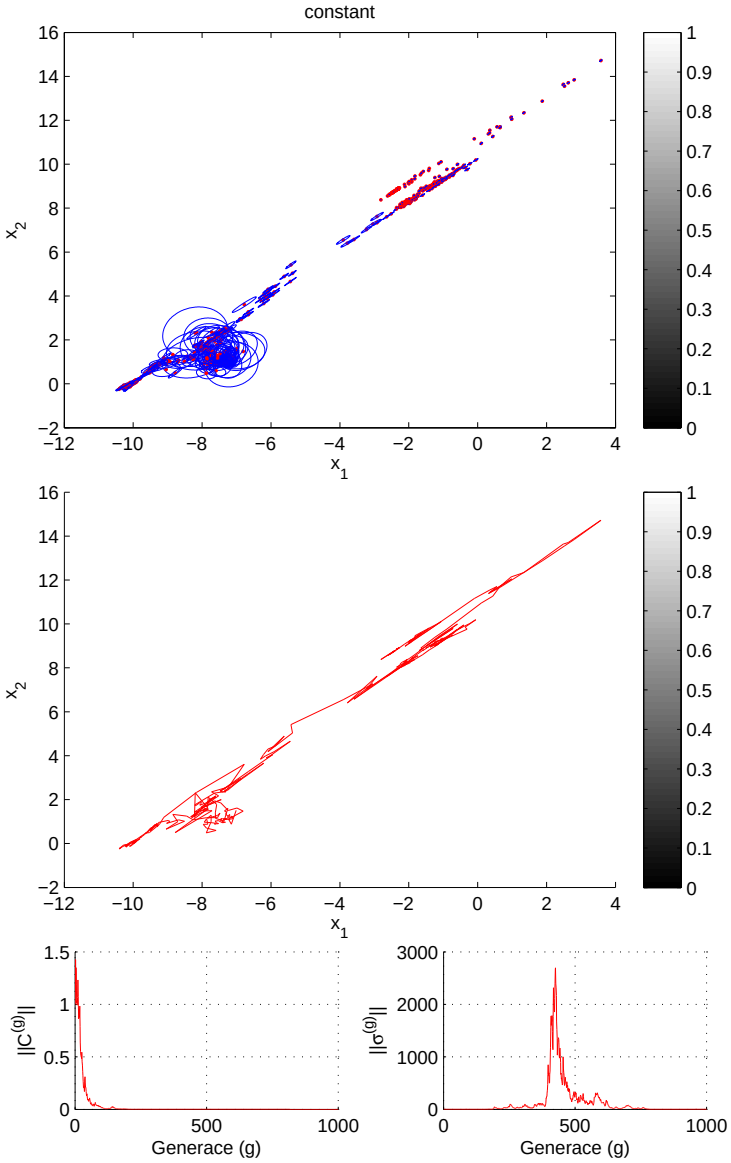
Obrázek C.3: Průběh algoritmu pro Bohachevskyho funkci prvního druhu ve dvou rozměrech Globální minimum je $\mathbf{x}^* = (0, 0)$, $f(\mathbf{x}) = 1$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



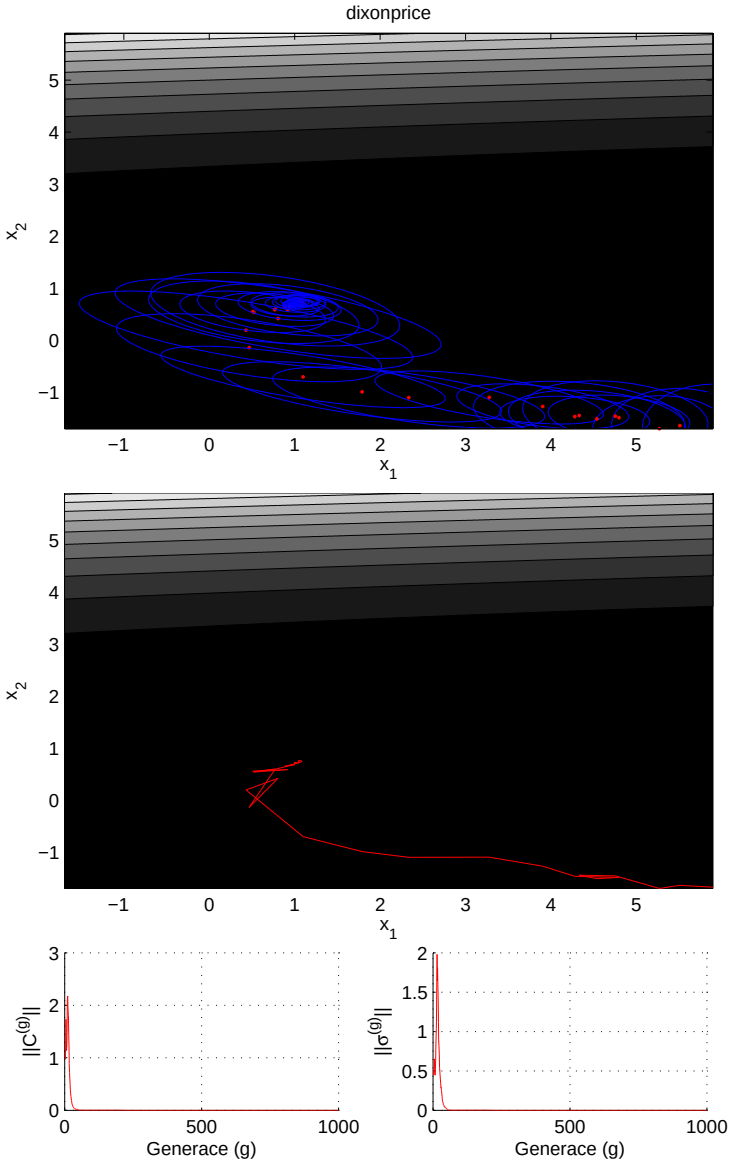
Obrázek C.4: Průběh algoritmu pro Booth funkci ve dvou rozměrech
Globální minimum je $\mathbf{x}^* = (1, 3)$, $f(\mathbf{x}^*) = 0$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



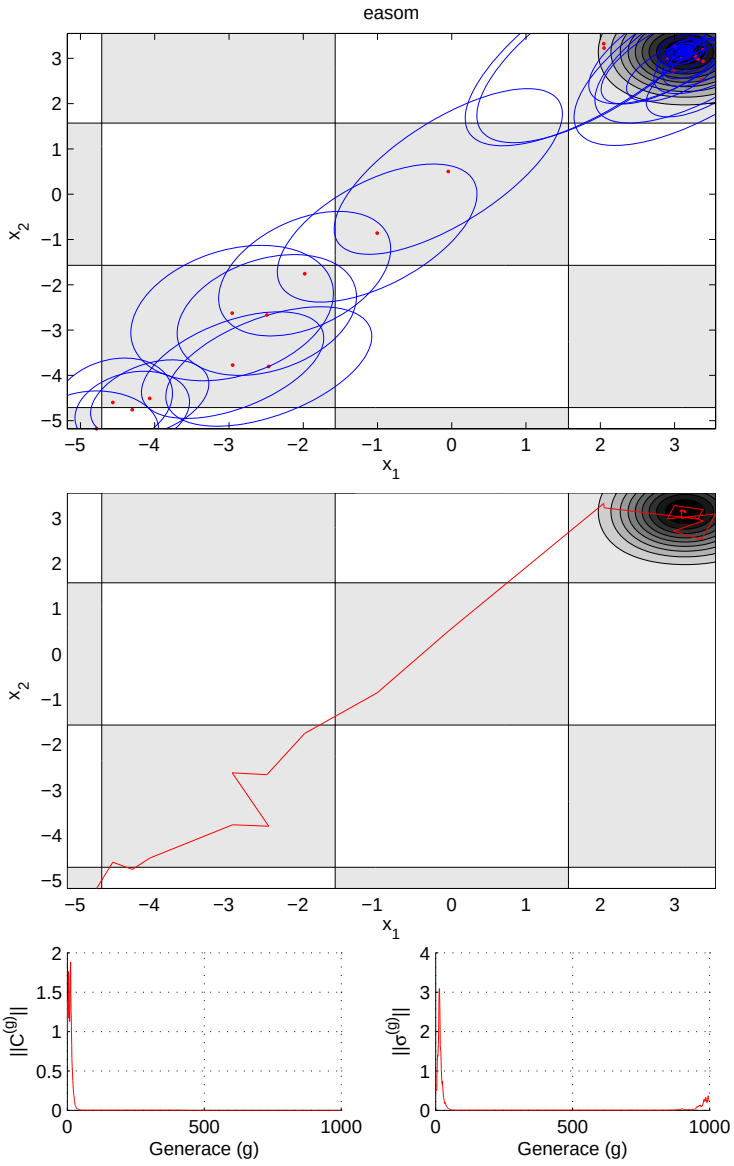
Obrázek C.5: Průběh algoritmu pro Braninovu funkci ve dvou rozměrech
Globální minimum je $\mathbf{x}_1^* = (-\pi, 12.275)$, $\mathbf{x}_2^* = (\pi, 2.275)$, $\mathbf{x}_3^* = (9.42478, 2.475)$, $f\left(\mathbf{x}_{1,2,3}^*\right) = 0.397887$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



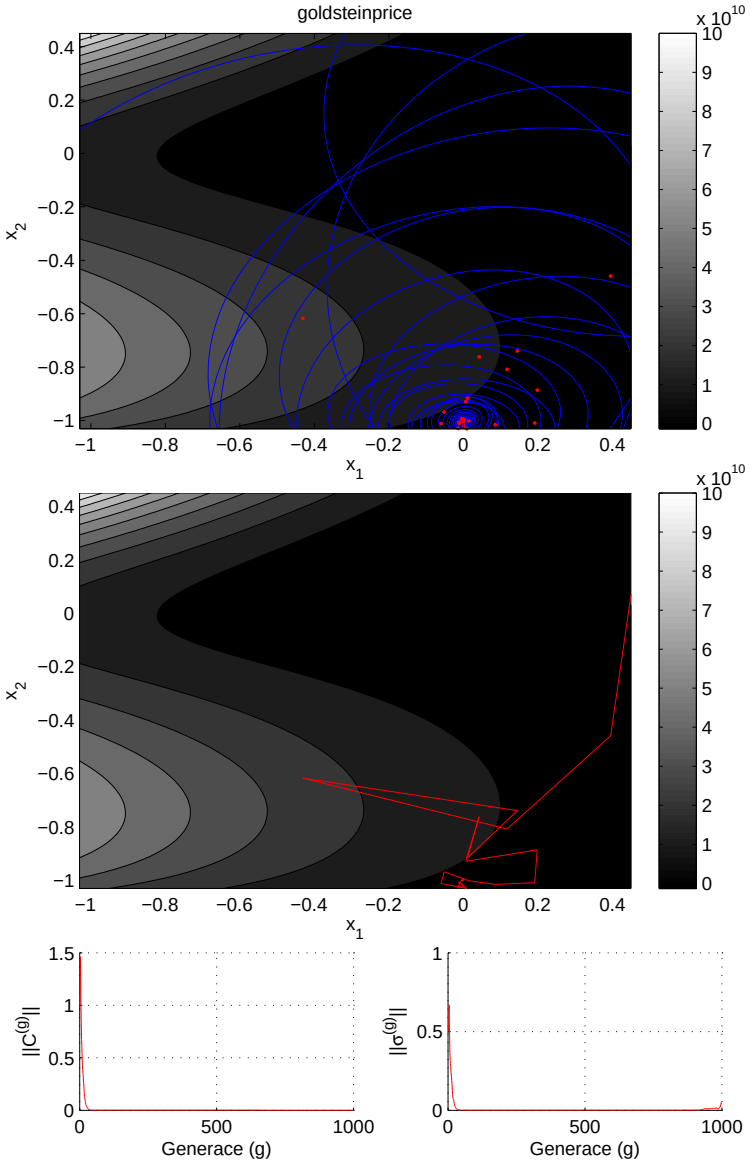
Obrázek C.6: Průběh algoritmu pro konstantní funkci ve dvou rozměrech
Globální minimum je $\mathbf{x} \in \mathbf{R}^2$, $\forall x \in \mathbf{R}^2$ $f(\mathbf{x}) = 0$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$. Protože je vypnutná zastavovací podmínka EqualFunValues algoritmus se zastaví až po určeném počtu iterací 1000.



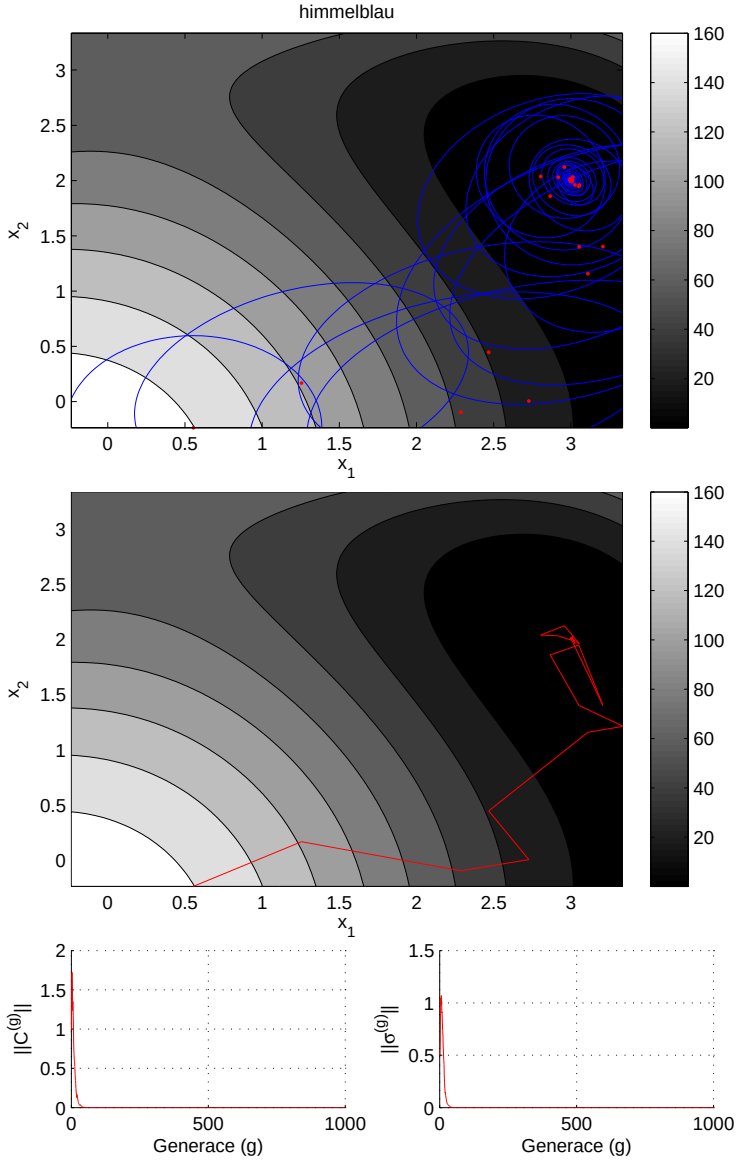
Obrázek C.7: Průběh algoritmu pro Dixon-Priceovu funkci ve dvou rozměrech Globální minimum je $\mathbf{x}^* = (0, 0)$, $f(\mathbf{x}^*) = 0$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



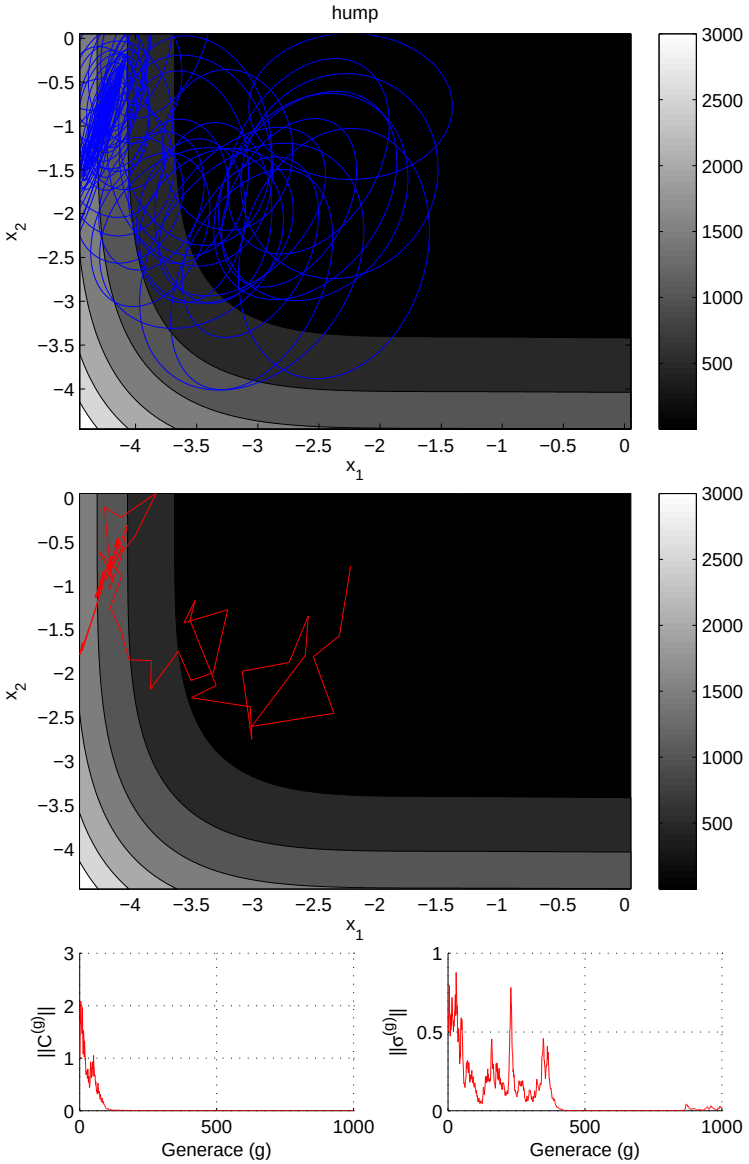
Obrázek C.8: Průběh algoritmu pro Easomovu funkci ve dvou rozměrech
Globální minimum je $\mathbf{x}^* = (\pi, \pi)$, $f(\mathbf{x}) = -1$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



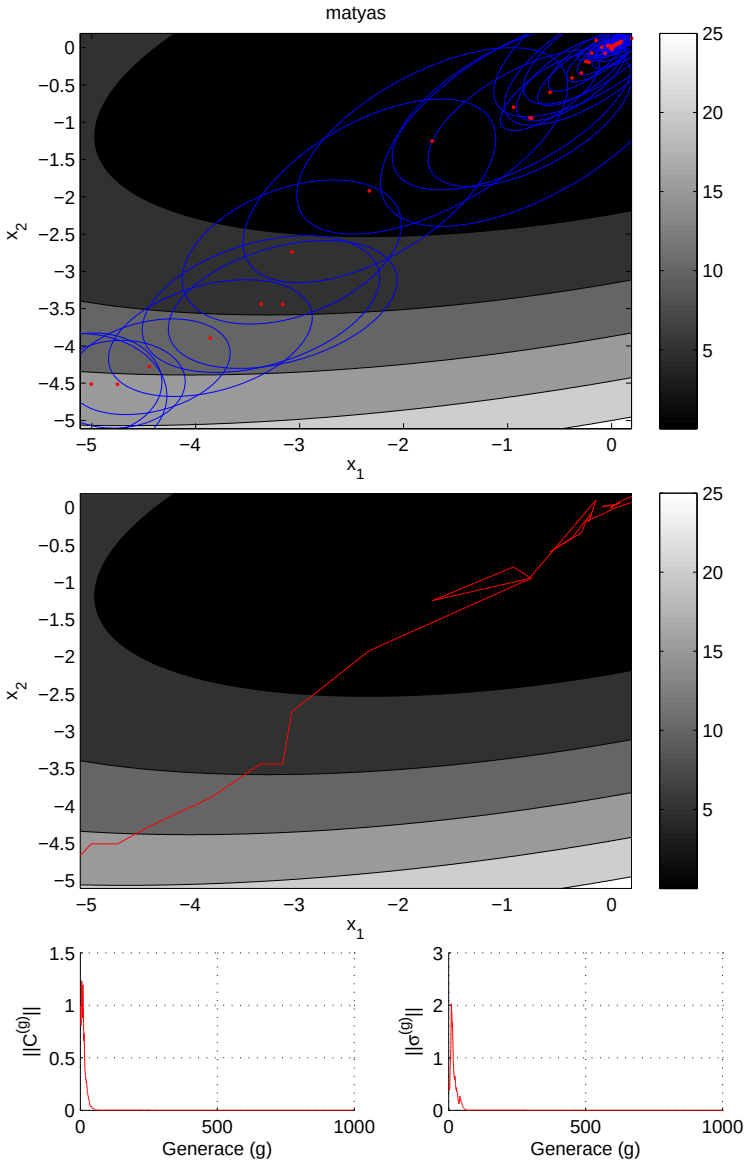
Obrázek C.9: Průběh algoritmu pro Goldstein-Priceovu funkci ve dvou rozměrech Globální minimum je $\mathbf{x}^* = (0, 1)$, $f(\mathbf{x}) = 3$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



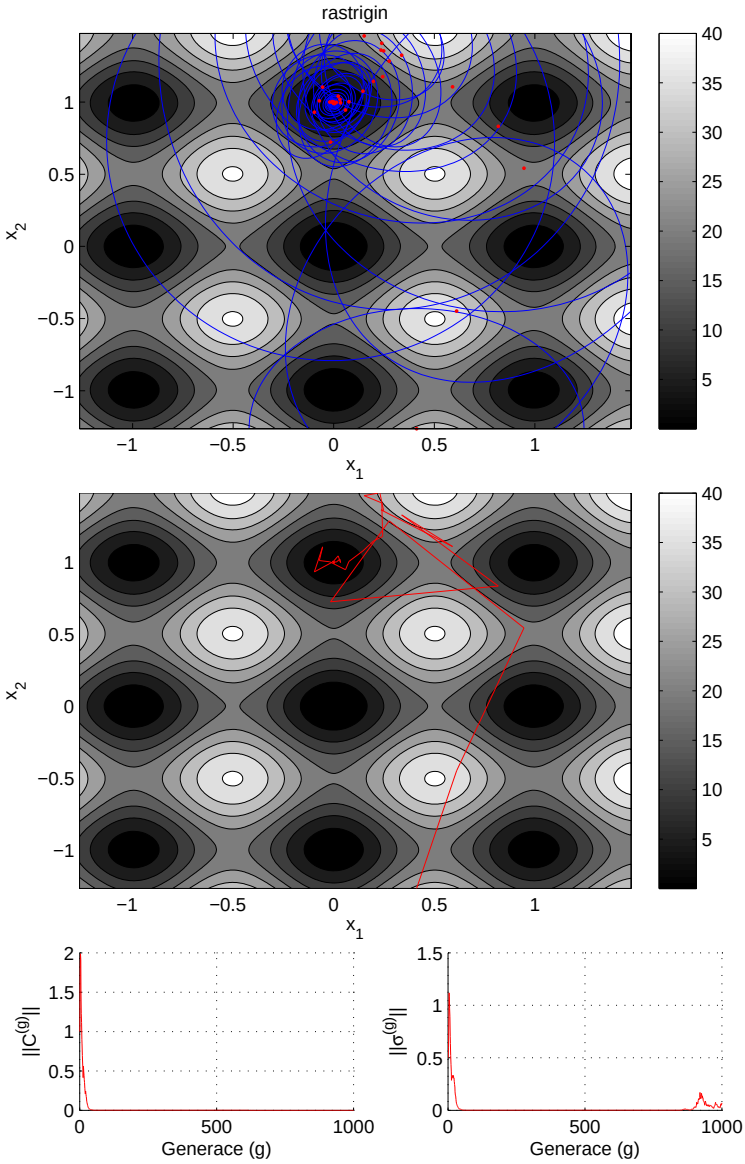
Obrázek C.10: Průběh algoritmu pro Himmelblauovu funkci ve dvou rozměrech Globální minimum je $\mathbf{x}^* = (-0.270844, -0.923038)$, $f(\mathbf{x}) = 181.616$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



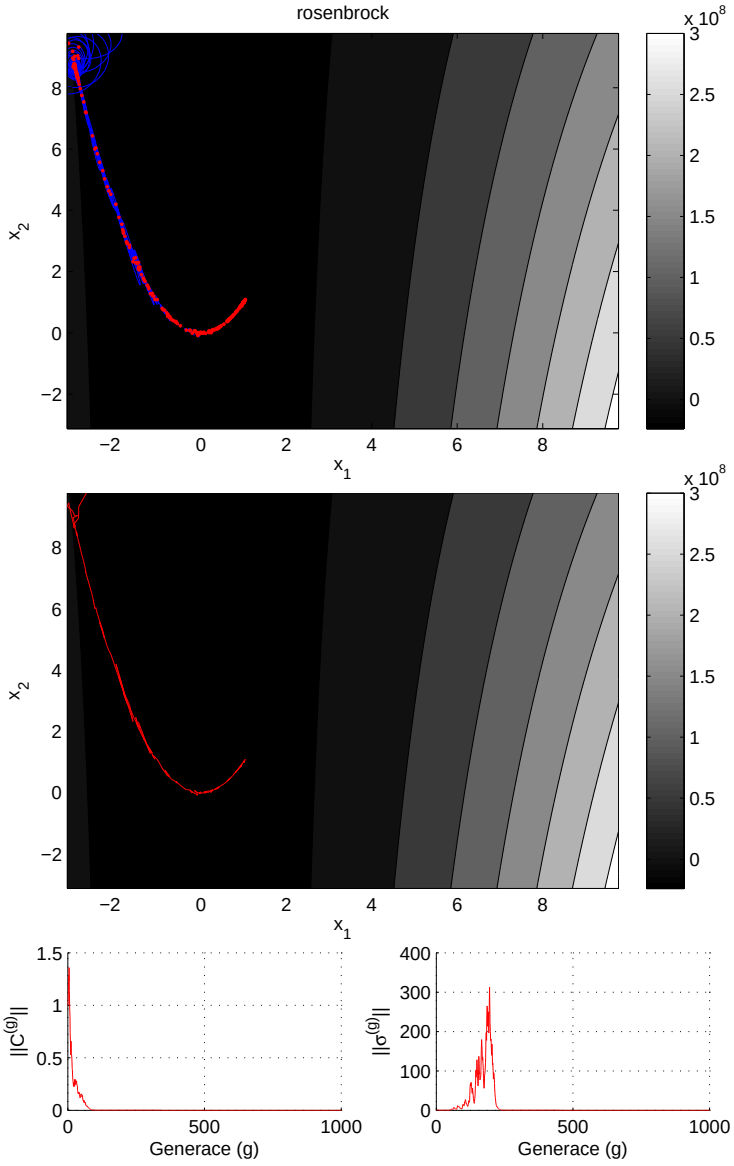
Obrázek C.11: Průběh algoritmu pro Humpovu funkci ve dvou rozměrech
Globální minimum je $\mathbf{x}^* = (0.0898, -0.7126)$, $\mathbf{x}_2^* = (-0.0898, 0.7126)$, $f\left(\mathbf{x}_{1,2}^*\right) = 0$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



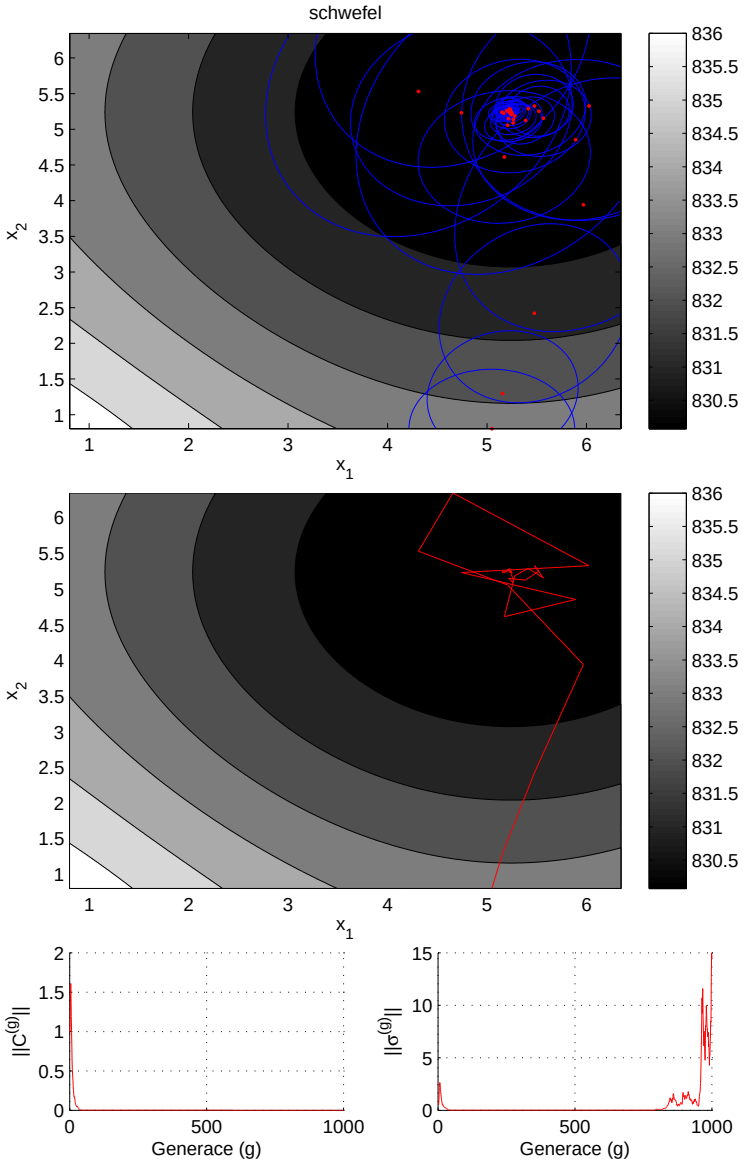
Obrázek C.12: Průběh algoritmu pro Matyasovu funkci ve dvou rozměrech
Globální minimum je $\mathbf{x}^* = (0,0)$, $f(\mathbf{x}^*) = 0$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



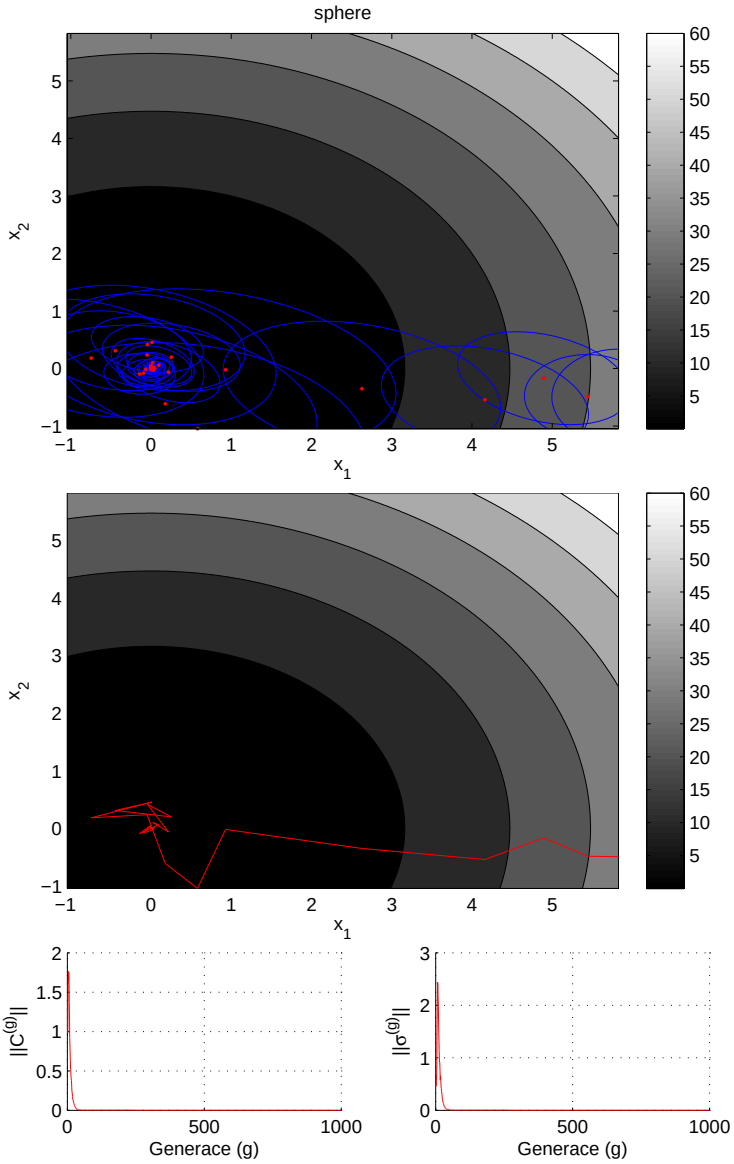
Obrázek C.13: Průběh algoritmu pro Rastriginovu funkci ve dvou rozměrech
Globální minimum je $\mathbf{x}^* = (0, 0)$, $f(\mathbf{x}^*) = 0$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



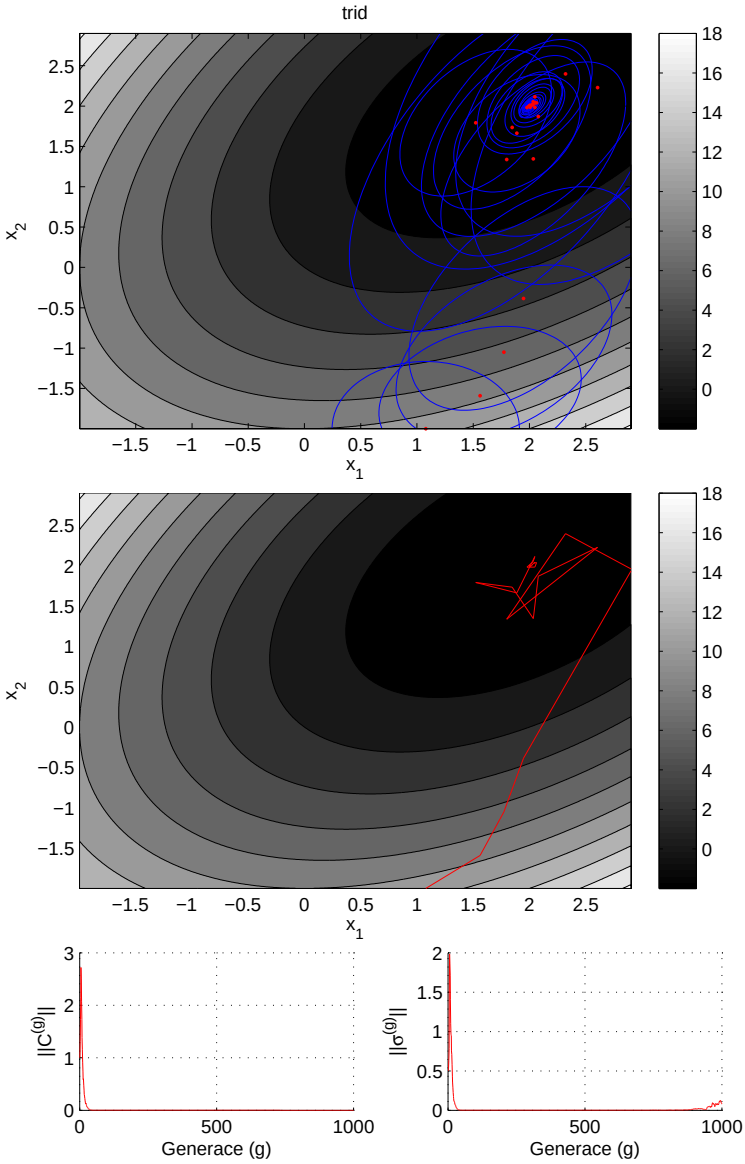
Obrázek C.14: Průběh algoritmu pro Rosenbrockovu funkci ve dvou rozměrech Globální minimum je $\mathbf{x}^* = (1, 1)$, $f(\mathbf{x}^*) = 0$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



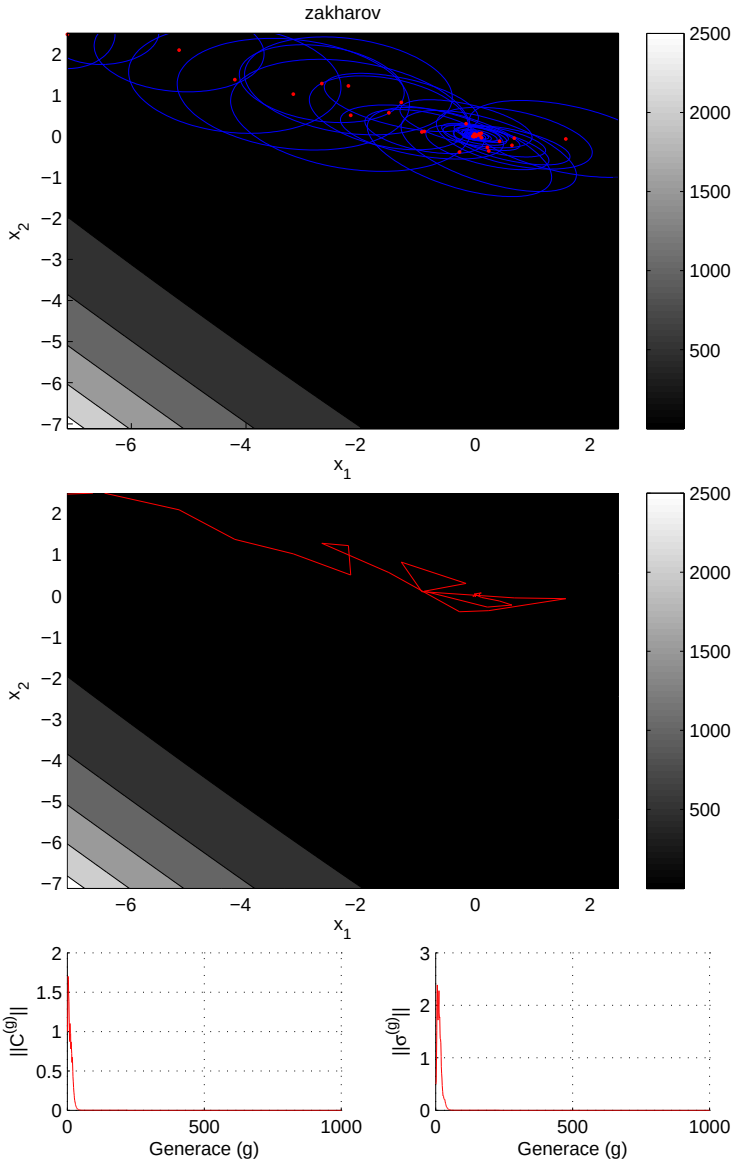
Obrázek C.15: Průběh algoritmu pro Schwefelovu funkci ve dvou rozměrech
Globální minimum je $\mathbf{x}^* = (1, 1)$, $f(\mathbf{x}^*) = 0$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



Obrázek C.16: Průběh algoritmu pro kvadratickou funkci ve dvou rozměrech Globální minimum je $\mathbf{x}^* = (0,0)$, $f(\mathbf{x}^*) = 0$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



Obrázek C.17: Průběh algoritmu pro Tridovu funkci ve dvou rozměrech
Globální minimum je $\mathbf{x}^* = (0, 0)$, $f(\mathbf{x}) = -2$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.



Obrázek C.18: Průběh algoritmu pro Zakharovovu funkcui ve dvou rozměrech Globální minimum je $\mathbf{x}^* = (0,0)$, $f(\mathbf{x}^*) = 0$. Červenou barvou je vyznačena vážená střední hodnota $\mathbf{m}^{(g)}$ a modrou kovarianční matice $\sigma^{(g)}\mathbf{C}^{(g)}$.

Příloha D

Tutorial

D.1 Prerequisites

D.1.1 Required applications

Before you run JCool user interface, check whether you have following applications:

- Subversion
- Maven
- Java Development Kit 1.6 or newer.

D.1.2 Checking out from repository

To checkout source code from repository use command:

```
svn co $REPOSITORY_ADDRESS/jcool/trunk jcool
```

First part of command **svn** calls subversion client. Second **co** command is for checkout source from defined path in third part of command. Third part is path to subversion repository server and you download latest release **trunk** of **jcool** project. Sources will be downloaded to **jcool** directory.

Currently JCool is hosted on sourceforge SVN repository on address

```
https://fakegame.svn.sourceforge.net/svnroot/fakegame
```

D.1.3 Building with Maven

After correct checkout all source files from repository you have to build all source into jar file.

To make package from whole project, you have to call following Maven command:

```
mvn package
```

This command call Maven which make packages for each of the module. Maven build five jar files for each module of the JCool project.

D.1.4 Running JCool user interface

If you succeed in previous steps you can run the UI. In module **ui** folder there is **target** folder and serveral files whose suffixes are **jar** and there should be **jcool-ui-1.0-SNAPSHOT.jar** file. To run the user interface, type following command:

```
java -jar jcool-ui-1.0-SNAPSHOT.jar
```

where `1.0` is current version of JCool in trunk.

D.2 Solving probles with JCool library

D.2.1 Writing fitenss function f

To solve any problem, that can be written as a parametric function of at least one variable (so mapping $f(\mathbf{x}) : \mathbf{R}^n \rightarrow \mathbf{R}$) with some contraints.

To write your own function, you should create java class in **benchmark** module in following package

```
cz.cvut.felk.cig.jcool.benchmark.function
```

your class must be derived at least from interface **Function** from package in **core** module.

```
cz.cvut.felk.cig.jcool.core
```

for example, to write linear function $f(x) = x_1 + x_2$ as a instance of Function class you can write:

```
package cz.cvut.felk.cig.jcool.benchmark.function;

import cz.cvut.felk.cig.jcool.core.Function;
import cz.cvut.felk.cig.jcool.core.Point;

public class LinearFunction implements Function{
    public double valueAt(Point point) {
        double x1 = point.toArray()[0];
        double x2 = point.toArray()[1];
        return x1+x2;
    }

    public int getDimension() {
        return 2;
    }
}
```

If function has a gradient, Hessian or bounds your class can implement interfaces of *FunctionGradient*, *FunctionHessian* or *FunctionBounds* respectively.

For completeness I extend instance *LinearFunction* of *FunctionGradient* and *FunctionHessian*.

```
public class LinearFunction implements Function,
FunctionGradient, FunctionHessian
{
    public double valueAt(Point point) {
        double x = point.toArray()[0];
        double y = point.toArray()[1];
        return x+y;
    }
    public int getDimension() {
        return 2;
    }
    public static void main(String [] argch){
        Solver solver = SolverFactory.getNewInstance(1000);
        solver.init(new LinearFunction(),new PureCMAESMethod());
        solver.solve();
        System.out.println(solver.getResults().getSolution());
    }

    public Gradient gradientAt(Point point) {
        double[] gradeint = new double[getDimension()];
        Arrays.fill(gradeint,1);
        return Gradient.valueOf(gradeint);
    }
    public Hessian hessianAt(Point point) {
        double [][] hessian =
                new double[getDimension()][getDimension()];
        Arrays.fill(hessian,1);
        return Hessian.valueOf(hessian);
    }
}
```

D.2.2 Solving problem with JCool

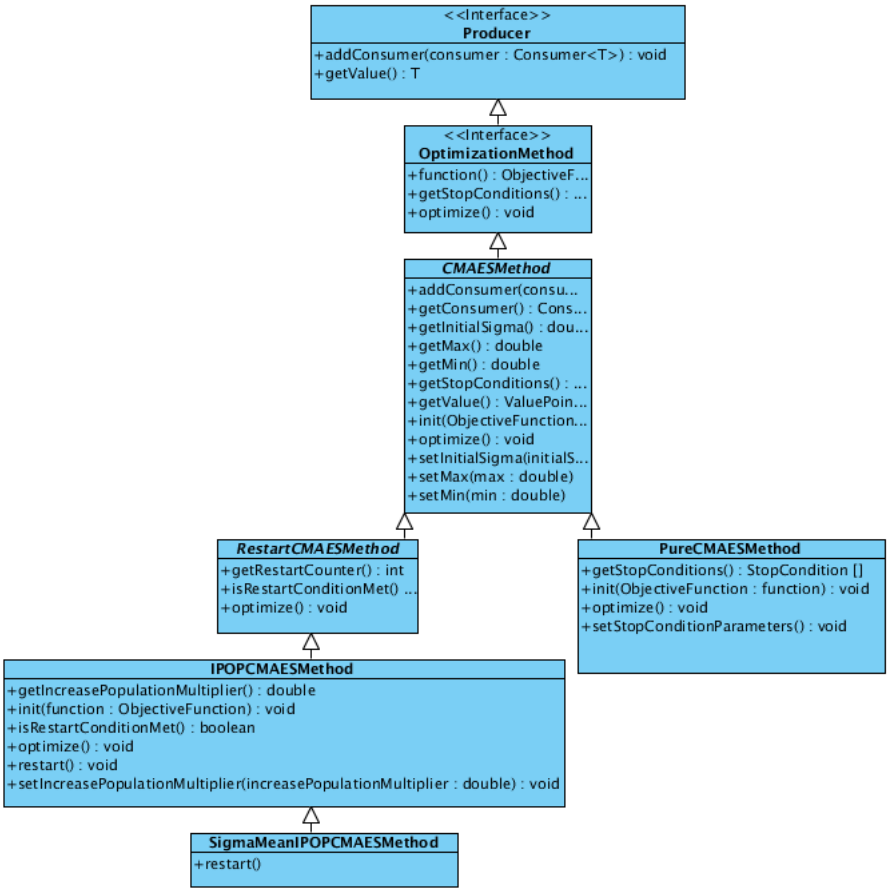
To solve problem you can create instance of **BasicSolver** class with **SolverFactory. getNewInstance**, initialize function **init** with your function and method and call method **solve**. Method **solver** try to solver problem in **MAX_ITERATIONS**.

```
int MAX_ITERATIONS = 1000;
Solver solver = SolverFactory.getNewInstance(MAX_ITERATIONS);
solver.init(new LinearFunction(),new PureCMAESMethod());
solver.solve();
```

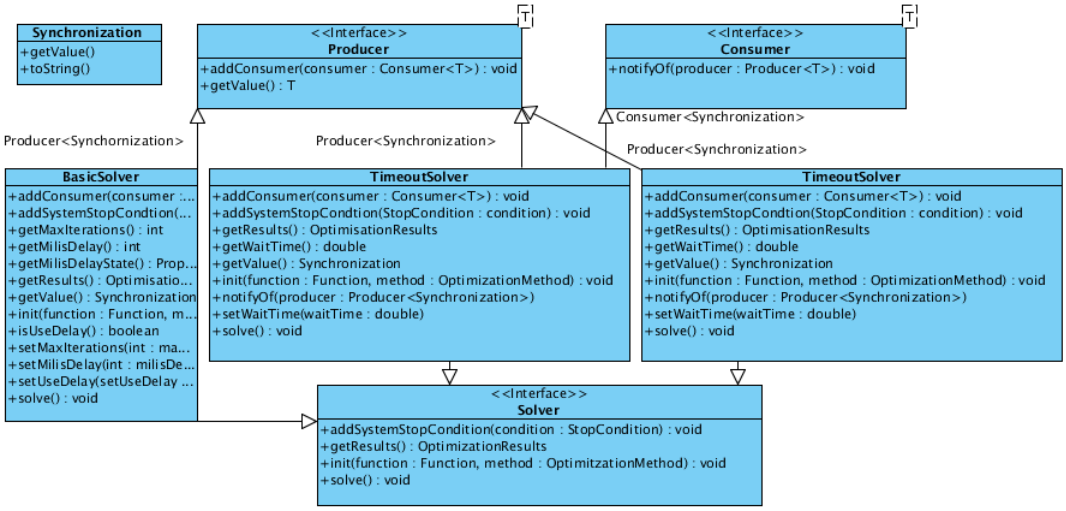
to get results of optimization method call solver instance **solver.getResults(). getSolution()** which returns Telemtry instance of results.

Příloha E

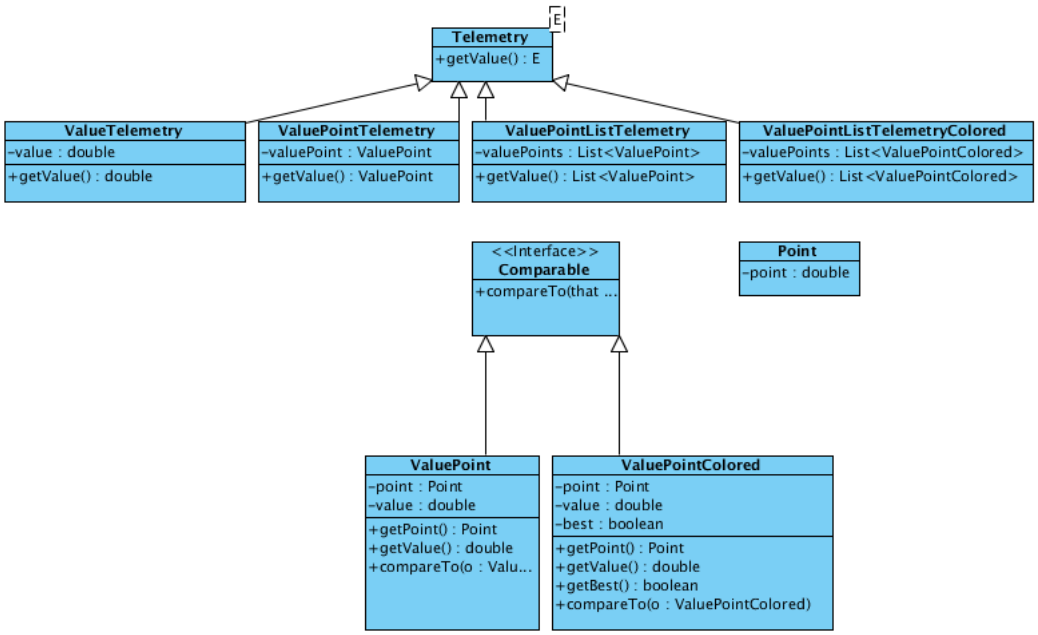
UML diagramy



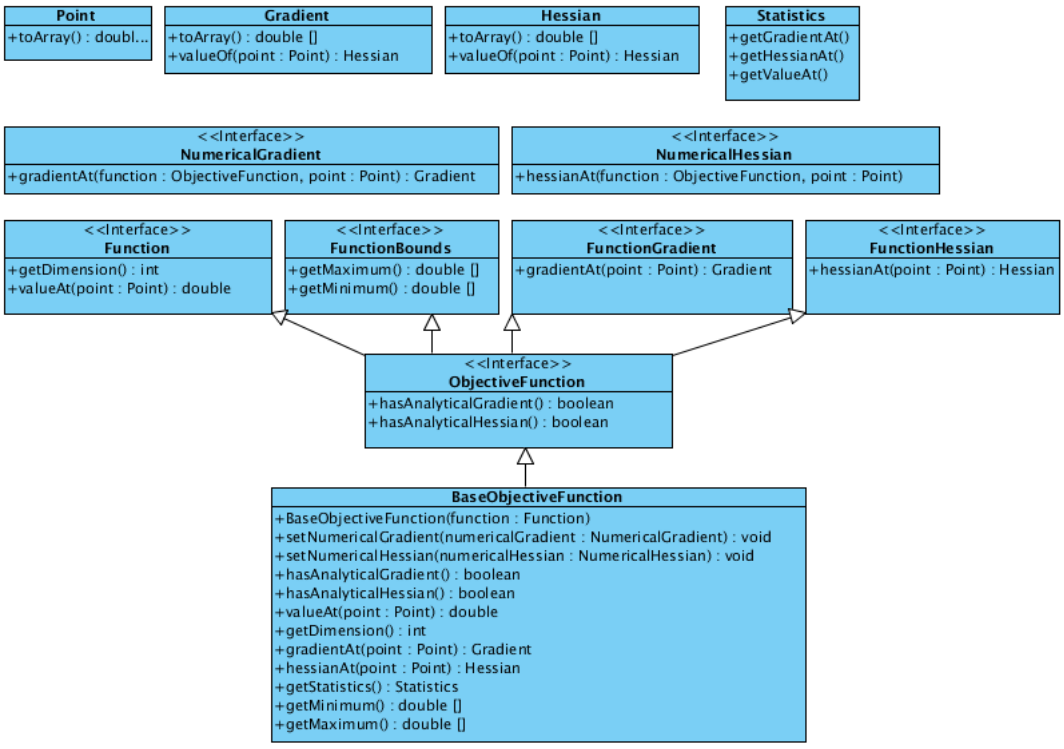
Obrázek E.1: Základní třídní hierarchie algoritmu CMA-ES



Obrázek E.2: Třídy Consumer, Producer a Solver.



Obrázek E.3: Hierarchie tříd pro telemetrie



Obrázek E.4: Hierarchie tříd, zapouzdřující fitness funkci. Potomci a předci třídy ObjectiveFunction.

Příloha F

Seznam použitých zkratek

EA Evoluční algoritmus

ES Evoluční strategie

GA Generický algoritmus

CMA-ES Evoluční strategie adapatace kovariančních matic

IPOP-CMA-ES Evoluční strategie adaptace kovariančních matic s restarty a nárůstem populace

Příloha G

Obsah přiloženého CD

```
|-articles
|---CMAES
|---ES
|-code
|-experiments
|---cmasteps
|---comparsion
|---figures
|-----cmasteps
|-----comparsion
|-----final_results
|-------1000
|-------10000
|-----fitnessfunctions
|---finalresults
|-reporting_code
|---solver
|-text
|--source
```

Obrázek G.1: Seznam přiloženého CD