



**MATEMATICKO-FYZIKÁLNÍ  
FAKULTA**  
Univerzita Karlova

## **BAKALÁŘSKÁ PRÁCE**

Marek Bečvář

# **Evoluce robotů v simulovaném fyzikálním prostředí**

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. František Mráz, CSc.

Studijní program: Informatika

Studijní obor: Informatika se specializací Umělá  
inteligence

Praha 2023

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne .....

Podpis autora

Poděkování.

Název práce: Evoluce robotů v simulovaném fyzikálním prostředí

Autor: Marek Bečvář

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. František Mráz, CSc., Katedra softwaru a výuky informatiky

Abstrakt: Abstrakt.

Klíčová slova: klíčová slova

Title: Evolution of robots in a simulated physical environment

Author: Marek Bečvář

Department: Department of software and computer science education

Supervisor: RNDr. František Mráz, CSc., Department of software and computer science education

Abstract: Abstract.

Keywords: key words

# Obsah

<b>Úvod</b>	<b>2</b>
<b>1 Základní pojmy</b>	<b>3</b>
1.1 Evoluční algoritmy . . . . .	3
1.1.1 Existující implementace . . . . .	6
1.2 Neuronové sítě . . . . .	12
1.3 Neuroevoluce . . . . .	14
1.3.1 NEAT . . . . .	15
1.3.2 HyperNEAT . . . . .	17
1.4 Simulované prostředí . . . . .	17
1.4.1 Fyzikální simulátory . . . . .	19
<b>2 Specifikace</b>	<b>22</b>
2.1 Funkční požadavky . . . . .	22
<b>3 Implementace</b>	<b>23</b>
3.1 Programovací jazyk . . . . .	23
3.2 Simulované fyzikální prostředí . . . . .	23
3.2.1 Roboti . . . . .	24
3.3 Genetické algoritmy . . . . .	24
3.4 Implementace knihovny . . . . .	24
3.5 Grafické rozhraní . . . . .	24
<b>4 Experimenty a výsledky</b>	<b>25</b>
4.1 Vývoj řízení robotů . . . . .	25
4.2 Vývoj řízení a morfologie robotů . . . . .	25
4.3 Diskuze výsledků . . . . .	25
<b>Závěr</b>	<b>26</b>
<b>Seznam použité literatury</b>	<b>27</b>
<b>A Přílohy</b>	<b>29</b>
A.1 První příloha . . . . .	29

# Úvod

Následuje několik ukázkových kapitol, které doporučují, jak by se měla bakalářská práce sázet. Primárně popisují použití T<sub>E</sub>Xové šablony, ale obecné rady poslouží dobře i uživatelům jiných systémů.

# 1. Základní pojmy

V této kapitole vysvětlíme a rozebereme důležité pojmy, se kterými se v dalším popisu práce budeme setkávat. Znalost těchto pojmů je potřebná pro pochopení důvodů volby daných vybraných technologií a pro pochopení základního rozboru implementace řešení, kterou popíšeme v následujících kapitolách.

V této kapitole nejdříve vysvětlíme základní teorii evolučních algoritmů (oddíl 1.1) a dále si v oddíle 1.1.1 ukážeme již existující knihovny pracující nebo umožňující pracovat s genetickými algoritmy. Poté se v oddílu 1.2 podíváme na základ teorie umělých neuronových sítí a v oddílu 1.3 popíšeme neuroevoluci, kategorii pokročilých evolučních algoritmů sloužící přímo k vývoji umělých neuronových sítí (algoritmy NEAT v oddílu 1.3.1 a HyperNEAT v oddílu 1.3.2). Dále popíšeme simulátory prostředí (oddíl 1.4) a fyzikální simulátory (oddíl 1.4.1), které využijeme pro simulaci při vyvíjení našich robotů.

## 1.1 Evoluční algoritmy

Evoluční algoritmus je označení pro stochastické vyhledávací algoritmy, které svými procesy napodobují principy Darwinovy evoluční teorie o přirozeném výběru a přežívání nejlepších jedinců. Pomocí těchto algoritmů se často snažíme optimalizovat nějaké procesy, nebo vlastnosti systémů, a tedy evoluční algoritmy můžeme označovat i jako optimalizační algoritmy. Mohou být úspěšné v řešení komplexních problémů (např. aproximace NP-úplných problémů) a problémů se složitě definovanou hodnotící funkcí. Stochastická vlastnost těchto algoritmů způsobuje, že algoritmy nezaručují nalezení optimálního řešení, ale jsou schopné se tomuto řešení přiblížit nějakým kvalitním suboptimálním řešením.

Mezi známé varianty evolučních algoritmů patří:

- *Genetické algoritmy (GA)* - kódující řešení pomocí vektorů binárních hodnot,
- *Evoluční strategie (ES)* - namísto binárních hodnot v GA, využívají ES reálné hodnoty a vektory reálných hodnot pro reprezentaci řešení,
- *Genetické programování (GP)* - řešení reprezentováno skládáním složitějších operací (např. pravidla bezkontextových gramatik, stromy s funkcemi a konstantami).

Jednotlivá řešení v evolučních algoritmech jsou nazývána jako **jedinci** s vlastními genetickými informacemi (**genotyp**). Kvalitu genotypu (označujeme jako **fitness**) je možné pro daný problém otestovat, vyhodnocením předem definované hodnotící funkce. Fitness je potom číselná hodnota, kterou se v procesem evolučního algoritmu snažíme maximalizovat/minimalizovat (dle definice hodnotící funkce). Množinu takových jedinců v evolučních algoritmech nazýváme **populace**. Celý vývoj evolučního algoritmu potom probíhá v cyklech, kdy v každém cyklu je otestována kvalita genotypu celé populace. Tyto jednotlivé cykly nazýváme **generace**.

Vedle inicializace populace, která je obvykle provedena náhodně, jsou důležitými částmi evolučních algoritmů tzv. **genetické operátory**. Jedná se o procesy

silně inspirované přírodními jevy pozorovaných při evoluci organismů. Tak jako se vyvíjí organismy v přírodě, za účelem udržení těch nejvhodnějších vlastností pro dané prostředí, vyvíjí se i jedinci v evolučních algoritmech, aby se přibližovali optimálnímu řešení zadaného problému (optimalizovali hodnotící funkci). Proces vývoje v evolučních algoritmech je pak zajištěn třemi genetickými operátory – **selekce**, **křížení** a **mutace**, všechny blíže popsány v následujících odstavcích. Určité varianty evolučních algoritmů nemusí využívat všechny genetické operátory.

**Selekce** Selekcce je proces, který vybírá jedince z populace, kteří buď přímo postoupí do další generace, nebo budou vybráni jako rodič (zdroj genetických informací) pro další generaci. Selekcce má upřednostňovat jedince s vyšší kvalitou genotypu, což by mělo v průběhu vývoje produkovat kvalitnější jedince, a tedy směřovat vývoj k optimálnímu řešení. Při vývoji může být aplikována ve dvou bodech. Pomocí selekce můžeme vybírat rodičovské jedince, kteří budou použiti pro tvorbu potomků, ze kterých se následně stane nová populace pro další generaci (*mating selection=selekce pro křížení*). Zároveň můžeme selekci využít po vytvoření potomků, pro výběr těch nejlepších jedinců z původní populace a z nových potomků, kteří budou v další generaci tvořit novou populaci (*environmentální selekce*).

Nejpoužívanějšími způsoby implementace operátoru selekce jsou *turnajová selekce* a *ruletová selekce*.

Při *turnajové selekci* náhodně vybereme specifický počet jedinců z populace a dle jejich ohodnocení (fitness) je rozřadíme (uspořádáme turnaj mezi jedinci). V tuto chvíli nejčastěji selekce vybere vítěze turnaje (nejlepšího jedince z turnaje). Existují i pravděpodobnostní varianty *turnajové selekce*, kdy pořadí v turnaji určuje pravděpodobnost zvolení daného jedince (vítěz turnaje nejvyšší pravděpodobnost, další v pořadí mají klesající šanci na zvolení). Výhodou *turnajové selekce* je, že nedává žádná omezení na fitness jedinců. Stačí nám, když jdou jednotlivé hodnoty fitness porovnat a rozlišit tak, která je lepší a která horší.

*Ruletová selekce* je abstrakce ruletového kola, kde každý jedinec má proměnlivou šanci na zvolení (šance, že jeho hodnota padne na ruletovém kole), přímo úměrnou jeho fitness. Tato selekce se potom provádí opakovaně, dokud nevybereme požadovaný počet jedinců, nad celou civilizací a ne pouze nad náhodně vybranou podmnožinou jedinců z populace (jak tomu bylo u *turnajové selekce*). Pokud se v populaci nachází jedinec s dominantní hodnotou fitness, která výrazně překonává ostatní jedince v populaci, může se stávat, že bude docházet k opakovanému výběru jednoho a toho samého jedince, což může vést k velmi rychlému omezení prohledávaného prostoru a tedy nalezení neoptimálního řešení (vývoj může uvíznout v tzv. *lokálním optimu*).

**Křížení** Křížení je proces, při kterém nějakým stylem kombinujeme genetické informace dvou (nebo více) jedinců. Tímto procesem vznikají potomci původních jedinců, kteří se mohou stát novou populací v následující generaci. Výběr jedinců pro křížení zajišťuje **selekce**. Křížení umožňuje důkladnější prohledávání prostoru možných řešení mezi již prohledanými možnostmi. Existuje mnoho variant křížení. Často využívanými variantami křížení jsou např. *jednobodové křížení* / *vícebodové křížení* nebo *uniformní křížení*.



Při *jednobodovém křížení* (pro dva rodičovské jedince) vybereme náhodně jedno místo (bod dělení), ve kterém genotyp rozdělíme. Genetickou informaci potomků následně vytvoříme tak, že složíme zpět rozdělené části genotypů různých rodičů (např. první potomek vznikne složením první části genotypu prvního rodiče a druhé části genotypu druhého rodiče a druhý potomek naopak). *Vícebodové křížení* probíhá stejně, pouze na začátku volíme vícero bodů, ve kterých genotyp rozdělíme a tak máme více možností, jak genotyp složit zpět.

*Uniformní křížení* při tvorbě potomků najednou prochází celý genotyp všech rodičů a u každé hodnoty uniformně náhodně vybere, z jakého rodiče se daný kus genetické informace zkopíruje do potomka. U tohoto typu křížení lze jednoduše použít i větší množství rodičů pro tvorbu potomků.

Jedná se o příklady metod křížení, často používané pro **genetické algoritmy**. Algoritmy pracující s reálnými hodnotami nebo jinak složitými strukturami mohou využívat jiné metody, pracující specificky s danými hodnotami genotypu (příkladem může být *křížení průměrováním* reálných hodnot dvou nebo více rodičů).

**Mutace** Při mutaci náhodně měníme malé části genotypu jedinců. Obvykle probíhá po vygenerování nových potomků pomocí **křížení**. Pomocí mutace evoluční algoritmy rozšiřují prostor prohledávaných řešení. Správně zvolená hodnota mutace zabraňuje uvíznutí evolučního algoritmu v lokálních optimech. Příliš silná mutace ale může způsobovat velké změny v genotypech a tedy tak zabránit vývoji jakýchkoli vlastností jedinců, které by vedly k optimalizaci řešení. Hodnotou mutace označujeme pravděpodobnost, že část genotypu bude pozměněna.

Styl, jakým se genotyp mutuje, je závislý na typu evolučního algoritmu. Pro genetické algoritmy s genotypem tvořeným vektorem binárních hodnot se může jednat o např. náhodný *bit flip* (tedy změnu dané hodnoty z 0 na 1 nebo naopak). U jiných typů evolučních algoritmů je možné používat např. přiřazení nové náhodně vygenerované hodnoty z rozsahu povolených hodnot pro danou část genotypu, nebo posun o malou náhodně vygenerovanou hodnotu.

**Elitismus** Elitismus je speciální proces, řadí se většinou k **selekci**, který umožňuje zachovat určité množství jedinců mezi generacemi beze změny. Obvykle jeden nebo malé množství nejlepších jedinců jsou po každé generaci přesunuti beze změny do další (tedy neprojdou žádnou mutací a jejich genotyp je stejný mezi generacemi). Jelikož je evoluční vývoj proces postavený na náhodě, mohlo by se stát, že náhodnou mutací přijdeme o nějaké potřebné vlastnosti z nejlepších jedinců a těch už nikdy nebudeme moci dosáhnout. Elitismus umožní tyto aktuálně nejlepší vlastnosti mezi generacemi zachovat.

Následující ukázka pseudokódu předvede jednoduchý příklad běhu evolučního algoritmu.

```

POP = náhodná inicializace populace
otestuj populaci a vypočti fitness
dokud problém není vyřešen:
    PARENTS = pomocí selekce vyber množinu rodičů
    OFFSPRING = křížením z rodičů vytvoř množinu potomků
    MUT_OFFSPRING = aplikuj mutaci na potomky

    POP = MUT_OFFSPRING z potomků vytvoř populaci další generace
    otestuj novou populaci a vypočti fitness

```

V ukázce můžeme vidět jednotlivé kroky jednoduchého evolučního algoritmu. Nejdříve inicializujeme populaci a zjistíme fitness jedinců. Následně dokud se nesplní podmínka pro ukončení evolučního vývoje (např. dosažení specifické fitness hodnoty), bude cyklicky v generacích probíhat vývoj populace, dokud se neobjeví takový jedinec, který problém vyřeší podmínku splní (vyřeší zadaný problém).

**Možnosti paralelizace** Všechny části evolučních algoritmů jsou ve své podstatě velmi jednoduché a tedy i rychlé na výpočet. Jediný bod, který může vývoj zpomalit je samotné testování jedinců. Pro příklad v našem problému má každý robot určitý čas pro vykonání svého pohybu. Jelikož každý robot musí být simulován zvlášť, je tento krok v našem evolučním algoritmu zdaleka ten nejnáročnější co se týče délky běhu. Naštěstí ve většině příkladů použití evolučních algoritmů, je tento krok možné poměrně jednoduše paralelizovat a i v našem případě tomu tak je. Jednotliví jedinci jsou v absolutní většině evolučních algoritmů naprosto nezávislí na ostatních. Pokud to simulační prostředí dovolí, je možné nechat každého robota simulovat nezávisle na ostatních ve vlastním vlákne (procesu) a tak využít moderních vícevláknových procesorů. Tato změna je poměrně jednoduchá na implementaci a absolutní většina evolučních algoritmů může tohoto využít pro mnohonásobné urychlení celého procesu evolučního vývoje.

### 1.1.1 Existující implementace

Pro vývoj řízení robotů budeme využívat evoluční algoritmy. Naše knihovna tedy bude implementovat několik alespoň základních genetických operátorů, používaných při vývoji jedinců a co nejjednodušeji umožňovat jejich konfiguraci před spouštěním jednotlivých experimentů. Naším cílem je co možná nejvíce zpřístupnit knihovnu, která má být výsledkem této práce, aby uživatel se základní znalostí genetických algoritmů a programovacího jazyka byl schopný pochopit běh algoritmu a v případě potřeby mohl jednoduše provádět zásahy do jeho běhu. Není tedy naším cílem najít tu nejefektivnější knihovnu, nýbrž tu, která přinese výhody jako přehlednost a snadnou úpravu algoritmů, bez větších obtíží s implementací a pochopením knihovny.

Dále představíme několik knihoven implementujících nebo usnadňujících implementaci genetických algoritmů nebo jejich částí. Celkem se podíváme na dvě knihovny – DEAP (sekce 1.1.1.1) a Inspyred (sekce 1.1.1.2). Existují další podobné knihovny (Pyevolve, PyGAD), které ale oproti DEAP a Inspyred nepřinášejí mnoho dalších užitečných možností.

### 1.1.1.1 DEAP

DEAP (*Distributed Evolutionary Algorithms in Python*) (DEAP) je open-source Python knihovna pro rychlou tvorbu a prototypování evolučních algoritmů. Snaží se tvorbu evolučních algoritmů zjednodušit pomocí přímočarého postupu, podobného pseudokódu, který je se základní znalostí knihovny poměrně jednoduchý na porozumění.

Knihovna je tvořena ze dvou hlavních struktur **creator**, který slouží k vytváření genetických informací jedinců z libovolných datových struktur a **toolbox**, který je seznamem nástrojů (genetických operátorů), které mohou být použité při sestavování evolučního algoritmu. Dalšími menšími strukturami jsou **algorithms** obsahující 4 základní typy algoritmů a **tools** implementující další základní operátory (části operátorů), které je posléze možné přidávat do **toolbox**. Pomocí těchto základních stavebních bloků mohou uživatelé poměrně jednoduše začít tvořit skoro libovolné evoluční algoritmy (Fortin a kol., 2012).

Následuje ukázka kódu tvorby základních částí evolučního algoritmu pro *One Max* problém, popsaná v oficiální dokumentaci knihovny DEAP. V *One Max* problému máme populaci jedinců, kteří reprezentují vektor binárních čísel předem zvolené délky. Cílem je potom vyvinout takového jedince, který má na všech pozicích vektoru nastavené jedničky.

Nejprve jsme v kódu importovali potřebné části knihovny. Dále využijeme **creator** pro tvorbu specifických tříd, které budeme potřebovat pro popis jedinců v našem evolučním algoritmu.

**Creator** **Creator** je třída sloužící jako *factory* pro uživatelem definované třídy. Tedy s její pomocí můžeme vytvářet nové třídy za běhu programu. To se hodí, protože různé problémy mohou vyžadovat velmi rozdílné typy jedinců. Tvorba třídy probíhá pomocí funkce **create**, která jako argumenty přijímá jméno vytvářené třídy, dále třídu, od které nová třída bude dědit a poté může následovat řada argumentů, které mohou být využity jako další argumenty naší nové třídy.

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)
```

První řádek popisuje tvorbu třídy **FitnessMax**, která dědí od třídy **Fitness** knihovny DEAP a zároveň obsahuje argument **weights**, který specifikuje, že fitness bude maximalizovat jediný cíl (pomocí DEAP můžeme specifikovat hned několik cílů najednou, ve kterých chceme, aby se jedinci zlepšovali).

Na druhém řádku vytváříme třídu jedince **Individual**, která dědí od třídy **list** a bude obsahovat parametr **fitness**, do které přiřadíme vytvořenou třídu **FitnessMax**.

**Toolbox** Dále využijeme vlastní třídy pro tvorbu specifických typů, které budou reprezentovat jedince a celou populaci. **Toolbox** se stane úložištěm pro všechny objekty, které budeme při tvorbě evolučního algoritmu používat. Do tohoto úložiště můžeme objekty přidávat funkcí **register** a můžeme je odebrat funkcí **unregister**.

```

# založení úložiště
toolbox = base.Toolbox()

# generátor atributů pro jedince
toolbox.register("attr_bool", random.randint, 0, 1)

# inicializace struktur jedince a populace
toolbox.register("individual",
                 tools.initRepeat,
                 creator.Individual,
                 toolbox.attr_bool,
                 100)
toolbox.register("population",
                 tools.initRepeat,
                 list,
                 toolbox.individual)

```

Nejdříve jsme vytvořili `toolbox` jako úložiště pro naše funkce. Dále jsme přidali generátor `toolbox.attr_bool()`, tvořený z funkce `random.randint()`, který když zavoláme, náhodně vygeneruje celé číslo mezi 0 a 1.

Dále jsme přidali dvě inicializační funkce `toolbox.individual()` pro inicializaci jedinců a `toolbox.population()` pro inicializaci celé populace.

Pro vytvoření jedince používáme funkci `tools.initRepeat()`, která jako první argument přijímá kontejner (v našem případě třídu jedince, kterou jsme definovali dříve jako potomka třídy `list`). Ten bude při inicializaci naplněn funkcí `toolbox.attr_bool()`, která se zavolá 100 krát, jak specifikují následující dva argumenty. Při inicializaci celé populace budeme postupovat stejně, jen jsme ještě v tento moment nespecifikovali, kolik jedinců bude do populace vytvořeno.

**Hodnotící funkce** Hodnotící funkce je pro tento problém jednoduchá. Potřebujeme pouze spočítat, kolik jedniček obsahuje daný jedinec (vektor binárních čísel).

```

def evalOneMax(individual):
    return sum(individual)

```

**Genetické operátory** Knihovna umožňuje dva přístupy, jak můžeme využívat genetické operátory. Buď je můžeme volat přímo z `tools`, nebo je nejdříve registrujeme do úložiště `toolbox` a z něho je budeme používat. Registrace je brána jako vhodnější varianta, protože to v budoucnu ulehčí proces, pokud bychom chtěli měnit používané operátory.

```

toolbox.register("evaluate", evalOneMax)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", tools.selTournament, tournsize=3)

```

Ohodnocení populace bude zajišťovat funkce `toolbox.evaluate()`, tvořící alias na dříve vytvořenou hodnotící funkci. Funkce `toolbox.mate()` je v tomto příkladu alias za `tools.cxTwoPoint()`, což je knihovnou připravená funkce provádějící dvoubodové křížení. Podobně tvoříme i funkce pro mutaci jedinců (v tomto případě binární mutaci – `tools.mutFlipBit`), kde argument `indpb` určuje pravděpodobnost mutace jednotlivých parametrů jedince a v poslední řadě funkci pro selekci (turnajová selekce s turnajem mezi třemi jedinci).

**Evoluce** Když jsou všechny části připravené, vlastní evoluční algoritmus se vytvoří kombinací jednotlivých definovaných částí, aplikováním registrovaných funkcí na populaci nebo jedince dle potřeby.

Na inicializované populaci se provádí evoluce, dokud nějaký z jedinců nesplní zadaný úkol, nebo dokud evoluce nedosáhne určitého počtu generací.

Pro stručnější popis zbytek kódu vynecháme, protože jsme si již předvedli všechny části spojené s definováním evolučního algoritmu, které jsou specifické pro práci s knihovnou DEAP. Úplnou ukázkou lze najít v oficiální dokumentaci knihovny (DEAP).

Podle článku (Fortin a kol., 2012), který porovnává několik Python modulů pro usnadnění práce s evolučními algoritmy, je DEAP nejefektivnější, tedy tvoří nejkratší kód, v porovnání počtu řádků potřebných pro tvorbu algoritmu řešící *One Max* problém z ukázky.

### 1.1.1.2 Inspyred

Inspyred (Garrett, 2012) poskytuje většinu z nejpoužívanějších evolučních algoritmů a dalších přírodou inspirovaných algoritmů (simulace reálných biologických systémů – např. optimalizace mravenčí kolonií) v jazyce Python.

Knihovna přichází s funkční implementací řady základních evolučních algoritmů ve formě komponentů (Python funkcí), které si uživatel může sám upravovat, rozšiřovat, nebo je úplně nahradit vlastními funkcemi. Při tvorbě algoritmu pak uživatel skládá dohromady několik komponentů, které ovlivňují, jak celý vývoj bude probíhat. Těmito komponenty jsou:

a) komponenty specifické danému problému:

- **generator** – určuje jak se generují řešení (jedinci),
- **evaluator** – definuje jak se vypočítává fitness jedinců,

b) komponenty specifické danému evolučnímu algoritmu:

- **observer** – definuje jak uživatel sleduje evoluci,
- **terminator** – rozhoduje, kdy by měla evoluce skončit,
- **selector** – rozhoduje, kteří jedinci by se měli stát rodiči další generace,
- **variator** – definuje jak jsou potomci vytvořeni z rodičovských jedinců,
- **replacer** – volí, kteří jedinci mají přežít do další generace,
- **migrator** – jak se přenáší jedinci mezi různými populacemi/generacemi,

- **archiver** – jak jsou jedinci ukládání mimo stávající populaci.

Libovolná z těchto komponent pak může být nahrazena odpovídající vlastní implementací (Tonda, 2020).

Následuje jednoduchý příklad z dokumentace knihovny *Inspyred*, který nám rychle představí možnou práci s knihovnou. Budeme řešit problém srovnatelný s problémem *One Max* představeným u příkladu knihovny *DEAP*. Nyní je cílem, aby hodnota vektoru v binárním zápisu byla co nejvyšší (opět chceme, aby výsledný jedinec měl na všech místech vektoru nastavené jedničky).

Na začátku se importuje knihovna *Inspyred* a modul **random**.

**Generator** Pro řešení budeme vytvářet vlastní generátor jedinců populace. Všechny generátory knihovny *Inspyred* mají vždy stejné dva argumenty:

- **random** – objekt pro generování náhodných čísel,
- **args** – slovník dalších argumentů, které můžeme libovolně přidat.

```
def generate_binary(random, args):
    bits = args.get('num_bits', 8)
    return [random.choice([0, 1]) for i in range(bits)]
```

Zde vytváříme vlastní funkci **generate\_binary**, která bude sloužit jako generátor jedinců. V této funkci nejdříve do proměnné **bits** načteme hodnotu z argumentu **num\_bits** (s jeho definicí se setkáme později) a poté vytvoříme jedince jako list, který naplníme náhodně zvolenými binárními hodnotami.

**Evaluator** Podobně jako generátor, tak i pro vyhodnocení fitness jedinců vytvoříme vlastní funkci. Funkce tohoto typu opět vyžadují dva argumenty:

- **candidate** – jedinec, kterého ve funkci vyhodnocujeme
- **args** – slovník dalších argumentů, které můžeme libovolně přidat

V knihovně se často setkáme s dekorátory funkcí. Přesněji funkce, které tvoříme pro **evaluator** vyžadují dekorátor **@evaluator**. Dekorátory se používají, protože vytváříme funkce, které budou použité v rámci jiných interních funkcí (pro příklad naše vlastní vyhodnocovací funkce pracuje pouze s jedním jedincem, ale funkce se bude při vyhodnocení fitness interně používat pro celou populaci).

```
@inspyred.ec.evaluators.evaluator
def evaluate_binary(candidate, args):
    return int("".join([str(c) for c in candidate]), 2)
```

Vyhodnocení jedince tedy vezme všechny prvky jeho vektoru, přečte je a vyhodnotí výstup jako binární zápis nějakého celého čísla. Toto číslo je potom výstupní ohodnocení pro daného jedince.

**Genetický algoritmu** Vytvořili jsme všechny potřebné části, specifické pro tento problém a tedy je můžeme použít pro vytvoření výsledného genetického algoritmu. Knihovna *Inspyred* nabízí několik různých typů evolučních algoritmů (genetické algoritmy, evoluční strategie, simulované žíhání a další). Pro tento problém vybereme základní formu genetického algoritmu, který je nám v této knihovně dostupný.

```
rand = random.Random()
ga = inspyred.ec.GA(rand)
ga.observer = inspyred.ec.observers.stats_observer
ga.terminator = inspyred.ec.terminators.evaluation_termination
```

Zde nejprve vytváříme objekt pro generování náhodných čísel, který bude využíván v algoritmu. Na druhém řádku pak vytváříme objekt samotného genetického algoritmu. Jak jsme zmínili výše, všechny algoritmy mají určité komponenty, které uživatel může měnit za jiné, nebo je celé sám upravovat. Pro ukázkou zde měníme komponenty **observer** a **terminator**. Pro **observer** volíme připravený **stats\_observer**, což je funkce, která v průběhu algoritmu bude vypisovat statistiky z běhu evoluce. Výstup tohoto **observeru** bude mít následující podobu:

Generation	Evaluation	Worst	Best	Median	Average	Std Dev
0	100	6	1016	564.5	536.02	309.833954
Generation	Evaluation	Worst	Best	Median	Average	Std Dev
1	200	29	1021	722.5	675.22	247.645576

Zároveň měníme i **terminator** za funkci **evaluation\_termination**, která jednoduše zahlásí, že evoluce má skončit, pokud evoluce dosáhla maximálního počtu vyhodnocení.

**Evoluce** Samotné spuštění a vyhodnocení evoluce je pak velmi jednoduché.

```
final_pop = ga.evolve(evaluator=evaluate_binary,
                      generator=generate_binary,
                      max_evaluations=1000,
                      num_elites=1,
                      pop_size=100,
                      num_bits=10)

final_pop.sort(reverse=True)
```

Genetický algoritmus se lehce spustí pomocí funkce **evolve**, které předáme požadované parametry jako náš zvolený **evaluator** a **generator**, dále pro **terminator** vkládáme maximální počet vyhodnocení, které chceme při vývoji dovolit. Dále je možné pro tyto algoritmy specifikovat jevy jako třeba elitismus. V poslední řadě **pop\_size** určuje velikost populace, se kterou bude evoluce pracovat a vkládáme zde dříve zmíněný argument **num\_bits**, určující velikost vektoru jedince.

Další informace o příkladu a jednotlivých funkcích lze nalézt v oficiální dokumentaci *Inspyred* (Garrett, 2012).

### 1.1.1.3 Porovnání

Při srovnání těchto knihoven jsme převážně sledovali jak intuitivní práce s nimi je. Cílem této práce je vytvořit co možná nejvíce otevřenou platformu, se kterou bude moci uživatel provádět experimenty při vývoji řízení robotů. Uživatel se základní znalostí evolučních algoritmů by tak měl být schopný jednoduše pochopit všechny části knihovny a pokud by měl potřebu, sám si doplnit nějaké specifické části.

Z vlastního pohledu, ačkoli knihovna DEAP umožňuje tvorbu asi libovolného evolučního algoritmu velmi kompaktním způsobem, potřeba pochopit a seznámit se s procesem tvorby vlastních tříd a objektů, na kterém je DEAP postavený, je poměrně velkou překážkou pro možného nového uživatele naší knihovny, který by si mohl chtít upravit proces evolučních algoritmů dle svých požadavků. Ačkoli méně kompaktní, řešení knihovny Inspyred 1.1.1.2, které dělí algoritmy do základních stavebních bloků, kde každá část může být se základní znalostí Pythonu pozměněna, se mi pro náš účel zamlouvá více. Inspyred ale zároveň implementuje řadu dalších algoritmů, které by v našem případě vůbec nemusely být využitelné a pouze by mohly zvyšovat minimální množství znalostí potřebných k práci s naší knihovnou.

Na základě vyzkoušených a předvedených knihoven, které se dle různých zdrojů (Fortin a kol., 2012) zdály jako nejvhodnější pro naše využití, jsme se rozhodli inspirovat se knihovnou Inspyred (stylem, jakým knihovna dělí algoritmus na základní stavební bloky) a vytvořit vlastní implementaci většiny základních stavebních bloků, které bude možné použít při tvorbě vlastních evolučních algoritmů v naší knihovně. Tyto bloky budou co možná nejvíce obecné a snadno konfigurovatelné funkce s předepsaným výstupním typem, aby uživatel mohl snadněji porozumět implementaci každého z bloků a zároveň aby měl možnost vytvářet vlastní bloky (Python funkce) a ty jednoduše zapojit do evolučního algoritmu, právě tak jak tomu je v knihovně Inspyred.

Celý projekt je směřován zároveň jako možný studijní materiál, a tak navíc věřím, že pro studenty bude možnost vidět v kódu funkční implementaci jednotlivých částí algoritmů tak, jak běží na pozadí experimentů přínosnější a ulehčí to jejich další experimenty, třeba i s implementací vlastních bloků evolučních algoritmů.

## 1.2 Neuronové sítě

Umělé neuronové sítě jsou matematickou abstrakcí biologických neuronů a jejich chování v nervové soustavě. Umělé neurony jsou zjednodušeně vrcholy v grafu, propojeného hranami (podobně jako reálné neurony jsou propojeny synapsemi) proměnlivých vah. Hodnota na neuronu (vzruch) je pro každý neuron vypočtena jako vážený součet příchozích vzruchů propojených korespondujícími hranami. V poslední řadě je na neuron aplikována nelineární aktivační funkce dále transformující výslednou hodnotu neuronu.

**Neuron** Nejmenší jednotkou neuronových sítí je jeden neuron. Pro neuron s  $n$  vstupy, jejichž hodnoty označíme  $x_1, \dots, x_n$  a váhy korespondujících hran označíme



$w_1, \dots, w_n$ , potom výstupní hodnotu tohoto neuronu, označenou  $O$ , můžeme spočítat jako,

$$O = f\left(\sum_{i=1}^n x_i w_i\right) \quad (1.1)$$

kde  $f$  je nelineární aktivační funkce aplikovaná na neuron.

**Aktivační funkce** Aktivační funkce slouží k transformaci hodnoty neuronu. Nejčastěji se jedná o nelineární transformaci pomocí nelineární funkce. Dnes nejčastěji používanějšími aktivačními funkcemi jsou například *logistická sigmoida* (vzorec 1.2), *hyperbolický tangens* (vzorec 1.3) nebo *ReLU* (vzorec 1.4). Pokud výstup neuronu před použitím aktivační funkce  $f$  označíme  $x$ ,

$$f(x) = \frac{1}{1 + e^{-x}} \quad (1.2)$$

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \quad (1.3)$$

$$f(x) = \max(0, x) \quad (1.4)$$

Grafy funkcí??

**Neuronová síť** Architektura neuronové sítě je tvořena ze třech základních typů neuronů – vstupní (ze kterých pouze vycházejí spojení), výstupní (do kterých pouze přicházejí spojení) a skryté (spojení přicházejí a jsou předávány dál). Neuronová síť pak jde popsat jako orientovaný graf. Pokud se jedná o graf bez orientovaných cyklů můžeme síť označit za *dopřednou neuronovou síť*. Pokud obsahuje nějaké orientované cykly, označujeme ji jako *rekurentní neuronovou síť*. Dále se budeme zaměřovat na *dopředné neuronové sítě*.

V těchto sítích vstup přechází po orientovaných hranách vrstvami neuronů, kde každý neuron z předchozí vrstvy je spojen hranou s každým vrcholem následující vrstvy. První vrstva se nazývá *vstupní vrstva*, tvořena ze *vstupních neuronů*, a slouží pro vstup parametrů (vstupů) do sítě. Poslední vrstva se nazývá *výstupní vrstva*, tvořena z *výstupních neuronů*, a můžeme na ní sledovat výstupní hodnoty neuronové sítě. Každá neuronová síť musí nutně mít alespoň jeden vstupní a alespoň jeden výstupní neuron. Libovolná další vrstva mezi *vstupní* a *výstupní vrstvou* se nazývá *skrytá vrstva*. Těchto vrstev může být v síti teoreticky neomezené množství.

**Trénování neuronových sítí** Trénování (nebo učení) neuronových sítí je často velmi složitý proces, při kterém se snažíme nastavit hodnoty vah jednotlivých spojů mezi neurony tak, abychom pro konkrétní vstup na *vstupní vrstvě*, dostali požadovaný výstup na *výstupní vrstvě*.

Toto je nejčastější požadavek pro neuronové sítě při tzv. *učení s učitelem* (*supervised learning*). Při tomto učení síť dostává dvojici (*vstup, chtěný výstup*). Následně vloží předložené vstupní hodnoty na vstup a sleduje svůj výstup. Poté porovná chtěný výstup se svým skutečným výstupem a na základě rozdílů (chyba) těchto hodnot provede úpravu vah jednotlivých spojů tak, aby se při dalším spuštění chyba zmenšila. Tohoto lze dosáhnout často používaným *backpropagation*

algoritmem, který hledá derivace chybové funkce, aby vždy prováděl úpravy vah spojů ve směru klesající chybové funkce.

Tento přístup ale přestává fungovat, pokud nedokážeme vytvořit vstupní trénovací dvojice (*vstup, chtěný výstup*), a tedy bychom nevěděli jakým směrem váhy spojů upravovat. Toto je překážka v mnoha praktických problémech, na které by se neuronové sítě hodilo použít. Možným řešením je nevyužívat učící metody založené na propagaci výsledné chyby sítě, ale použít nějakou hodnotící funkci, která ohodnotí kvalitu konfigurace dané sítě (např. po simulačním běhu). Toto nás vede na možnost využití evolučních algoritmů pro trénování neuronových sítí evolučním vývojem.

## 1.3 Neuroevoluce

Neuroevoluce Lehman a Miikkulainen (2013) je technika pro evoluční vývoj umělých neuronových sítí pomocí principů evolučních algoritmů (popsáno v sekci 1.1). Vývoj pomocí neuroevoluce je méně omezený než vývoj pomocí klasických trénovacích metod, jelikož vývoj, na rozdíl od trénovacích metod založených na principu *učení s učitelem*, může probíhat i na sítích proměnlivé architektury a nepotřebuje znát korektní výstupy pro daný vstup sítě. Pro trénování stačí, když jsme schopni nějakým stylem ohodnotit kvalitu řešení, k jakému se s využitím dané konfigurace (architektura a váhy spojení) sítě dostaneme. Díky tomuto se neuroevoluce hodí pro vývoj neuronových sítí v případech, kdy nejsme schopni přesně určit správné výstupy sítě a pouze můžeme pozorovat kvalitu chování daného vyvíjeného systému.

**Vývoj neuronových sítí** Vývoj pomocí neuroevoluce probíhá stejně jako u jiných evolučních algoritmů. Neuronová síť je zakódovaná do genotypu jedinců, množina těchto jedinců potom tvoří populaci, procházející vývojem skrz opakované generace. V každé generaci je genotyp každého jedince rozkódován a z těchto informací vytvořena neuronová síť pomocí zakódované architektury a vah spojení. Sítě odpovídajících jedinců jsou následně otestovány v testovací prostředí, které ohodnotí kvalitu jedince (fitness).

Pro příklad jednoduchým typem kódování může být uložení hodnot vah všech hran v neuronové síti do jediného vektoru a tento vektor využít jako genotyp jedinců. To umožňuje optimalizaci vah sítě s fixní architekturou. Může být ale velmi náročný, kvůli rychle narůstající délce genotypu s velikostí architektury neuronové sítě.

**Pokročilé metody** S těmito problémy se můžeme potýkat hned několika způsoby. Různé styly zakódování neuronových sítí do genotypu umožňují vývoj mnohem rozsáhlejších sítí, zachovávající udržitelně malou velikost genetické informace jedinců.

Některé metody Gomez a kol. (2008) navrhovaly postup, jakým můžeme omezit vývoj z celých neuronových sítí na pouze menší komponenty, které dále mohly být spojovány dohromady s ostatními jedinci v kooperativní evoluci. To umožnilo evoluční vývoj sítí rozsáhlejších architektur s menšími nároky na velikost genotypu.

Jiné metody se poté zaměřily na vývoj jak vah, tak topologie neuronové sítě. Navíc se ukázalo, že současný vývoj topologie často přináší lepší výsledky, než vývoj pouze vah sítě. Vývoj v těchto metodách začíná s nejzákladnější strukturou sítě, která se podle nároků problému sama vyvíjí a rozšiřuje, dokud tyto změny přináší kvalitativní zlepšení. Často využívaným algoritmem využívající těchto metod je algoritmus NEAT (*NeuroEvolution of Augmenting Topologies*), který si předvedeme v následujícím oddílu.

### 1.3.1 NEAT

NEAT (*NeuroEvolution of Augmenting Topologies*) (Stanley a Miikkulainen, 2002) je neuroevoluční algoritmus vyvíjející najednou váhy synapsí i topologii neuronové sítě, který se díky své výkonnosti stal jedním z nejznámějších algoritmů používaných pro účely evolučního vývoje neuronových sítí. Autoři jeho efektivitu připisují třem základním principům, se kterými algoritmus pracuje:

1. značení genetických informací pomocí tzv. *historických značek*, umožňující křížení genotypů napříč různými topologiemi,
2. ochrana nových genetických informací v populaci pomocí rozdělení do druhů,
3. postupný vývoj topologie sítí začínající nejjednoduššími strukturami.

S těmito principy se NEAT ukázal jako algoritmus, který často nachází efektivní sítě rychleji než ostatní algoritmy a výkony překonal nejlepší neuroevoluční algoritmy pracující s neuronovými sítěmi s fixní topologií.

**Algoritmus** NEAT používá tzv. *přímé kódování*, tedy genotyp každého jedinců přímo popisuje celou topologii sítě (všechny neurony a všechny hrany mezi neurony). Každý genotyp obsahuje list obsahující *geny synapsí* a list obsahující *geny neuronů*. Seznam genů neuronů popisuje vstupní, výstupní a skryté neurony sítě, které mohou být spojené hranami. Každý z genů synapsí obsahuje záznam o informacích:

- vstupní neuron – neuron, do kterého synapse přichází,
- výstupní neuron – neuron, ze kterého synapse vychází,
- hodnotu váhy synapse,
- příznak, jestli je spojení v síti použito,
- *inovační/historické číslo*, popisující kdy v historii byla daná hrana do sítě přidána, umožňující nacházet odpovídající geny pro křížení.

Algoritmus začíná s nejzákladnější strukturou, připomínající jednoduchý perceptron, pouze s předem daným počtem vstupních a výstupních neuronů a hranami mezi nimi. Tato jednoduchá topologie je rozšiřována pomocí následujících genetických operátorů.

**Mutace** Mutace v algoritmu NEAT má schopnost měnit jak spojení v síti, tak její strukturu. Mutace synapsí sítě mutují jednoduchou změnou váhy spojení nebo změnou příznaku použití daného spojení v síti. Struktura sítě může mutovat dvěma styly. První typ mutace přidává nový gen synapsí, spojující dva doposud nespojené neurony. Druhý typ mutace přidává nový neuron. Tato mutace probíhá tak, že na místo existující synapse se přidá nový neuron. Původní synapse se označí jako nepoužívaná a namísto toho se vytvoří dvě nové rozdělující tu původní. Váha spojení je v tomto procesu zachována. Při těchto operacích je vždy novým genům synapsí navýšeno jejich *inovační číslo* (používán globální čítač inovací, jehož číslo je s každým novým genem navýšeno). Růst sítí popisovaných genotypy probíhá právě díky mutaci.

**Křížení** Křížení probíhá za pomoci *inovačních čísel*. Dva jedinci se nejdříve hranami zarovnají pomocí jejich *inovačních čísel*. Stejná čísla totiž v síti značí stejnou strukturu. Synapse, které se v obou jedincích schodují se dědí náhodně. Synapse, které má jen nějaký z jedinců se dědí z toho lepšího z dvojce rodičů a pokud je v nějakém jedinci hrana neaktivní a v druhém aktivní, s určitou pravděpodobností se tento stav v novém jedinci změní. Algoritmus je tímto stylem schopný vytvářet velké množství různých topologií. Tyto nové topologie, přestože často důležité pro řešení zadaného problému, ale mají jen velmi malou šanci se v populaci menších topologií udržet, protože původní menší topologie jsou na začátku často optimálnější než větší topologie. Proto NEAT využívá systém, kterým chrání tyto nové topologie před vyhynutím z populace.

**Ochrana nových druhů** Nové topologie jsou chráněny rozřazením genotypů do odlišných druhů. Jednotlivé genotypy poté primárně soutěží s jedinci stejného druhu a nové druhy tak mají šanci se vyvinout a optimalizovat na jejich úroveň. NEAT pro výpočet odlišností jedinců opět využívá *inovační čísla*, pomocí kterých hledá společné hrany a vypočítá vzdálenost dvou genomů. V každé generaci je pak tvořen seznam různých druhů a pokud je genom, který nezapadá do žádného z druhů, je pro něj vytvořen jeho vlastní nový druh.

**Fitness** Rozdělení do druhů má vliv i na fitness jedinců. Kvalita každého jedince se při výpočtu fitness dělí počtem jedinců stejného druhu. To zároveň omezuje druhy v ovládnutí celé generace a dále ochraňuje nové topologie.

**Minimalizace dimenzionality jedinců** Jak již bylo zmíněno, algoritmus NEAT inicializuje celou jedinců s minimální topologií, obsahující pouze potřebný počet vstupních a výstupních neuronů, které jsou navzájem plně propojené a žádné skryté neurony. Nové topologie vznikají díky genetickým operátorům a všechna zvětšení genotypů jsou tedy v evoluci opodstatněná. Díky tomuto NEAT samovolně vede k vývoji minimálních topologií. To umožňuje tomuto algoritmu být často výkonnější než ostatní, protože musí prohledávat minimální prostor, aby našel řešení, oproti neuroevolučním algoritmům používající fixní topologie neurových sítí.

### 1.3.2 HyperNEAT

Algoritmus HyperNEAT (*Hypercube-based NEAT*) (Stanley a kol., 2009) (Eplex) je neuroevoluční algoritmus rozšiřující algoritmus NEAT. HyperNEAT slouží pro vývoj umělých neuronových sítí fixní topologie (typicky omezená tvarem hyperkrychle).

Na rozdíl od algoritmu NEAT, používá HyperNEAT nepřímou reprezentaci vah sítě. Tyto váhy jsou reprezentovány pomocí jiné neuronové sítě (*Compositional Pattern-Producing Network*, zkráceně *CPPN*), která jako vstup dostává pozice dvou neuronů v prostoru a vrací váhu jejich spojení (synapse). Tímto stylem může *CPPN* být využita pro reprezentaci sítě libovolné topologie.

Síť *CPPN* je poté v algoritmu HyperNEAT vyvíjena pomocí NEAT.

Díky této nepřímé reprezentaci vah má HyperNEAT schopnost efektivně vyvinout velmi rozsáhlé neuronové sítě s předem určenou strukturou (schopnost napodobit regularitu velkého množství spojů v lidském mozku).

Následuje naznačení průběhu HyperNEAT algoritmu:

1. Zvolit konfiguraci vyvíjené sítě (vstupní a výstupní neurony a rozložení skrytých neuronů),
2. Inicializovat NEAT algoritmus s *CPPN* sítěmi,
3. Běh NEAT algoritmu, dokud není nalezeno řešení:
  - (a) Pomocí *CPPN* daného jedince vytvoř synapse pro původní síť,
  - (b) Síť otestuj v testovacím prostředí pro výpočet kvality řešení,
  - (c) S fitness hodnotami pokračuj vývoj genotypů popisujících *CPPN* síť.

## 1.4 Simulované prostředí

++ Rozlišit simulátory

Jelikož chceme vyvíjet řízení robotů založené na interakcích s prostředím, je pro tuto práci důležité vybrat vhodný simulátor prostředí, založený na skutečných fyzikálních zákonech. Přáli bychom si mít možnost jednoduše konfigurovat co nejvíce vlastností prostředí a zároveň mít co nejjednodušší přístup k morfologii simulovaných robotů. Zároveň chceme, abychom měli možnost do morfologie robotů nějakým způsobem zasahovat i v průběhu evolučního vývoje a aktivně ji za běhu měnit. Jelikož plánujeme v prostředí provádět experimenty s různými typy robotů, používajícími různé styly pohybu (typy motorů, kloubů, tvarů končetin, atd.), je potřebné, aby fyzikální simulátor (*fyzikální řešič=solver*) byl schopný simulovat i složitější typy robotů. Takovými mohou být právě třeba kráčející roboti, neboli roboti používající k pohybu končetiny připomínající nohy, na rozdíl od jednodušších typů robotů, kteří se mohou pohybovat pomocí kol, jejichž simulace bývá mnohdy jednodušší.

Stejně tak, jak potřebujeme umožnit simulaci složitějších robotů, protože nebudeme mít možnost vlastnoručně kontrolovat každý parametr, který bude při vývoji robotům přiřazen, potřebujeme zajistit, aby fyzikální simulátor zvládal velké rozsahy parametrů a simulace zůstala s těmito parametry stabilní. Zároveň

chceme, aby simulátor v prostředí byl deterministický, což umožní, že předváděné experimenty můžeme dle potřeby opakovat a výsledky tak náležitě prezentovat.

Evoluční algoritmy jsou velmi lehce paralelizovatelné a tedy pro urychlení procesu vývoje a experimentů bude pro nás výhodné, pokud by simulace zvládala paralelní běh na více vláknech (více simulací, každá na vlastním vlákne). V poslední řadě pro lehčí integraci do vlastního modulu bude užitečné, aby modul spravující zvolený simulátor byl open-source, což nám dá volnost v případě, že si budeme chtít chování systémů v prostředí nějak vlastnoručně upravit.

Při hledání simulátorů prostředí, které by vyhovovali našim požadavkům a umožňovali kontrolu a ovládání prostředí prostřednictvím zvoleného jazyka Python, jsme narazili na několik možností. Omezený výčet těchto simulátorů zde popíšeme – Gazebo (v oddílu 1.4.0.1), Webots (v oddílu 1.4.0.2) a CoppeliaSim (v oddílu 1.4.0.3). Dále se pak podíváme na několik nejpoužívanějších fyzikálních simulátorů 1.4.1.

#### 1.4.0.1 Gazebo

Gazebo (OpenRobotics) je sada open-source víceplatformních knihoven pro vývoj, výzkum a aplikaci robotů, která vznikla v roce 2002. Umožňuje kompletní kontrolu nad simulací dynamického 3D prostředí s více agenty a generování dat ze simulovaných senzorů. Fyzikálně korektní interakce v prostředí pak od začátku projektu zajišťuje známý fyzikální simulátor ODE (viz sekce 1.4.1.1), nad kterým Gazebo tvoří abstraktní vrstvu, umožňující snazší tvorbu simulovaných objektů různých druhů. V dnešní době je stále výchozím fyzikálním simulátorem ODE, nicméně uživatel již může vybrat celkem ze čtyř různých simulátorů – Bullet (sekce 1.4.1.2), Simbody, Dart (sekce 1.4.1.3) a ODE. Uživatel s knihovnou pracuje prostřednictvím grafického rozhraní založené na knihovně Open Scene Graph používající OpenGL, nebo prostřednictvím příkazové řádky. Prostor a roboti mohou být tvořené buď z grafického rozhraní prostředí, nebo v textovém formátu XML. Limitací Gazebo je pak chybějící možnost rozdělit simulace mezi vícero vláken kvůli vnitřní architektuře spojené s fyzikální simulací (Koenig a Howard, 2004).

#### 1.4.0.2 Webots

Webots (Webots) je open-source víceplatformní robustní a deterministický robotický simulátor vyvíjený od roku 1998, umožňující programování a testování virtuálních robotů mnoha různých typů a jednoduchou následnou aplikaci softwaru na reálné roboty. Simulátor je možné použít pro simulaci prostředí s vícero agenty najednou s možnostmi lokální i globální komunikace mezi agenty. Výpočty fyzikálních interakcí zajišťuje fyzikální simulátor ODE. Pro vývoj robotů a prostředí je možné využít řady programovacích jazyků a to C, C++, Python, Java, MATLAB nebo ROS (*Robot Operating System*). Prostor umožňuje práci v grafickém rozhraní a vizualizaci simulací pomocí OpenGL. Knihovna dále nabízí využití připravených modelů robotů, vlastní editor robotů a map a možnosti vložení vlastních robotů z 3D modelovacích softwarů v CAD formátu (Michel, 2004).

### 1.4.0.3 Coppeliasim

CoppeliaSim (Rohmer a kol., 2013) (původně známý pod jménem *V-REP* = *Virtual Robot Experimentation Platform*) je víceplatformní simulační modul pro vývoj, testování a jednoduchou aplikaci softwaru pro roboty. Dovoluje vývoj ovladačů pomocí 7 různých programovacích jazyků a ulehčuje jejich aplikace v simulovaných a skutečných robotech. Simulaci ovladačů je možno jednoduše roz distribuovat mezi vícero vláken dokonce vícero strojů, což urychluje vývoj a snižuje nároky na procesor v době simulace. Navíc je možné vyvíjený ovladač nechat v době simulací běžet na vlastním na dálku připojeném robotovi, co dále ulehčuje přenos finální verze ovladačů od vývoje do skutečného světa. Prostředí umožňuje práci s širokou řadou typů objektů, druhů kloubů, senzorů a dalších objektů obvykle používaných při vývojích robotických ovladačů. Obsahuje lehce použitelný editor prostředí a robotů samotných s řadou předem vytvořených modelů, které může uživatel hned využít. Modely zároveň mohou být přidány v řadě různých formátů (XML, URDF, SDF). Prostředí podporuje pět různých fyzikálních simulátorů (Bullet, ODE, MuJoCo (v sekci 1.4.1.4), Vortex (v sekci 1.4.1.5), Newton), mezi kterými si uživatel může vybrat dle potřeb přesnosti (reálnosti), rychlosti a dalších možností jednotlivých fyzikálních simulátorů (Nogueira, 2014).

## 1.4.1 Fyzikální simulátory

V této podkapitole se podíváme na základní popis a možné výhody a nevýhody jednotlivých fyzikálních simulátorů, na které jsme narazili při hledání simulátorů prostředí.

### 1.4.1.1 ODE

ODE (*Open Dynamics Engine*) (Smith) je víceplatformní open-source fyzikální simulátor, jehož vývoj začal v roce 2001. Je vhodný pro simulaci pevných těles s různými druhy kloubů a pro detekci kolizí. Byl navržen pro využití v interaktivních nebo real-time simulacích, upřednostňující rychlost a stabilitu nad fyzikální přesností (Smith a kol., 2007). Vyžaduje používat menší simulační kroky kvůli stabilitě. Hodí se pro simulaci vozidel, pochoduujících robotů a virtuálních prostředí. Má široké využití v počítačových hrách a 3D simulačních nástrojích CoppeliaRobotics.

### 1.4.1.2 Bullet

Bullet je open-source fyzikální knihovna, podporující detekci kolizí a simulaci pevných a měkkých těles. Bullet je používán jako fyzikální simulátor pro hry, vizuální efekty a robotiku (Coumans). Byl použit jako hlavní fyzikální simulátor pro simulaci NASA *Tensegrity* robotů (s vlastními úpravami pro simulaci měkkých těles, kvůli nerealistickým metodám řešení simulace provazů) (Izadi a Bezuijen, 2018).

### 1.4.1.3 Dart

Dart (*Dynamic Animation and Robotics Toolkit*) je víceplatformní open-source knihovna pro simulace a animace robotů. Od předchozích se odlišuje stabilitou

a přesností, díky zobecněné reprezentaci souřadnic pevných těles v simulaci. Na rozdíl od ostatních fyzikálních simulátorů, aby dal vývojáři plnou kontrolu nad simulací, umožňuje Dart plný přístup k interním hodnotám simulace. Zároveň se díky línému vyhodnocování hodí pro vývoj real-time ovladačů pro roboty (Lee a kol., 2018).

#### 1.4.1.4 MuJoCo

MuJoCo (*Multi-Joint Dynamics with Contact*) (DeepMind, 2021) je open-source fyzikální simulátor pro vývoj v oblasti robotiky, biomechaniky a dalších. Často je využíváno pro testování a porovnávání různých metod navrhování robotických systémů jako jsou třeba evoluční algoritmy nebo metody zpětnovazebního učení (Salimans a kol., 2017). V simulacích je pro roboty možné nakonfigurovat využití mnoha druhů aktuátorů, včetně těch simulujících práci svalů a k dispozici je i velké množství kloubů. Simulátor zároveň umožňuje velký nárůst v rychlosti běhu simulace za pomoci plné podpory paralelizace na všech dostupných vláknech a stabilitě simulace i při velmi velkých simulačních krocích (Todorov a kol., 2012). Zároveň nabízí jednoduchý styl, jakým si může uživatel konfigurovat všechny detaily simulace a samotných simulovaných robotů pomocí jednoduchých XML konfiguračních souborů (XML formát modelů *MJCF*). V komplexním rozboru řady čteně používaných fyzikálních simulátorů byl simulátor MuJoCo hodnocen jako jeden z nejlepších co se týče stability, přesnosti a rychlosti simulací. Další výhodou zlepšující přesnost tohoto simulátoru je, že MuJoCo pro simulaci používá kloubní souřadnicový systém, který předchází narušení fyzikálních pravidel a tedy nepřesností v kloubech (Erez a kol., 2015).

#### 1.4.1.5 Vortex

Vortex je uzavřený, komerční fyzikální simulátor určený pro tvorbu reálnému světu odpovídajících simulací. Obsahuje mnoho parametrů, umožňující nastavení reálných fyzikálních parametrů dle potřeb, většinou industriálních a výzkumných aplikací (CoppeliaRobotics) (Yoon a kol., 2023).

#### 1.4.1.6 Porovnání simulátorů

V dnešní době se nám nabízí velké množství potencionálních kandidátů, vhodných k využití pro naši aplikaci. Prakticky každý z open-source simulátorů, které jsme našli a předvedli, by bylo možné použít pro simulaci robotů složitosti, jakou máme předběžně v plánu. Hlavním z rozhodujících faktorů pro tento projekt bude jak jednoduše půjde prostředí používat pro vývoj pomocí genetických algoritmů. Chceme tedy nějaký jednoduchý přístup k simulaci a ovládání robotů, rychlost a přesnost simulace.

Opět většina ze simulátorů prostředí toto nabízí. Osobně se nám ale nejvíce zalíbilo MuJoCo. Díky nedávnému otevření fyzikálního simulátoru MuJoCo a změně prostředí (nejprve do **OpenAI Gym** a nyní do **Farama Foundation Gymnasium**) jsme dostali možnost využít jednoduché Python API pro ovládání robotů a zároveň konfiguraci celé simulace.

Tato abstrakce od vlastní simulace je pro tuto práci velmi přínosná, protože se především chceme zabývat o vývoj řízení robotů pomocí evolučních algoritmů.



Řešit zároveň složité ovladače robotů, které by se mohly lišit pro různé typy robotů, by mohlo bezdůvodně komplikovat celý proces spojení evolučních algoritmů s řízením robotů. Takové věci by pak mohly být problematické pro možného uživatele, který by si chtěl sám evoluční algoritmy upravovat.

MuJoCo se zároveň ukazuje jako jeden z nejlepších volně dostupných fyzikálních simulátorů dnes. Z výsledků článku testujících různé vlastnosti známých fyzikálních simulátorů Erez a kol. (2015) vychází, že MuJoCo má navrch jak v rychlosti, tak ve kvalitě simulací. Zároveň interně využívá kloubní souřadnicový systém, který je přesnější, protože zabraňuje nepřesnostem v kloubech. To se hodí o to více, když v této práci chceme vyvíjet hlavně krácející roboty, u kterých můžeme mít i větší počty kloubů.

Simulátor MuJoCo a roboti, které můžeme používat, je zároveň možné jednoduše konfigurovat pomocí vlastního XML formátu a spojení s Python API navíc umožní tyto konfigurace provádět jak často bude potřeba.

## 2. Specifikace

### 2.1 Funkční požadavky

## 3. Implementace

V předchozí kapitole jsme prošli funkční požadavky, očekávané od vyvíjeného souboru programů. Následuje rozbor jednotlivých modulů, které vznikly při vlastní implementaci. Zároveň zde projdeme možné alternativy, které se pro vývoj nabízejí a probereme důvody stojící za zvolením jednotlivých z možností.

Nejprve vysvětlíme volbu programovacího jazyka, ve kterém je celá knihovna vytvořena. Poté projdeme systémy umožňující vývoj ve fyzikálním prostředí a ovládání uživatelem definovaných robotů. Zde představíme i možnosti tvorby vlastních robotů. Dále ukážeme možné varianty modulů umožňující vývoj řízení robotů pomocí genetických algoritmů a popíšeme vlastní implementaci. Následně projdeme všechny části implementace spojující tyto moduly do přístupné rozšiřitelné knihovny. V poslední části představíme implementaci grafického rozhraní, které slouží uživateli, který chce používat knihovnu a provádět experimenty, bez nutnosti využití příkazové řádky.

### 3.1 Programovací jazyk

Jako programovací jazyk, ve kterém tento projekt bude psán, jsme zvolili **Python**. Cílem projektu je vytvořit platformu, kterou bude uživatel moci použít k vývoji robotů pomocí evolučních algoritmů. Pokud uživatel bude mít potřebu jakkoli připravený proces vývoje měnit, Python lehce umožní nahlédnout do zdrojových kódů vypracované knihovny a provést úpravy dle vlastních potřeb. Zároveň to umožňuje rozšiřování knihovny o nové metody, které bude chtít uživatel zkusit zařadit do již funkčního procesu. Jednoduchá čitelnost Pythonu spojená s rychlostí, jakou mohou být prováděny iterace změn bez potřeby zdoluhavého překladu celé knihovny, se zdají býti dostatečně dobré vlastnosti pro volbu programovacího jazyka pro tento projekt.

### 3.2 Simulované fyzikální prostředí

Po zhodnocení vypsanych a dalších možností jsme vybrali pro využití v tomto projektu fyzikální simulátor MuJoCo. Na rozdíl od ostatních se zdá býti přístupnější do začátku a zároveň dostatečně robustní a konfigurovatelný tak, aby splnil veškeré požadavky, které od fyzikálního simulátoru máme.

**MuJoCo 1.50** je fyzikální simulátor zpřístupněný skrz volně dostupné aplikační rozhraní společnosti **OpenAI** v rámci jejich sady různých prostředí **Gym** (textové hry, jednoduché 2D i plně fyzikálně simulované 3D prostředí, Atari hry aj.) pro vývoj metod zpětnovazebného učení na různých problémech. Toto rozhraní umožňuje uživatelům jednoduchý přístup k datům z poskytnutých prostředí a ovládání prostředím definovaných agentů, pomocí standardizovaných vstupů i výstupů napříč všemi prostředími. Tímto způsobem můžeme velmi lehce ovládat i roboty v prostředích simulátoru MuJoCo. Navíc otevřená vlastnost tohoto aplikačního rozhraní umožňuje úpravu částí procesu tak, aby se lépe hodil při řešení námi zvolených problémů. Přestože je **Gym** převážně používána pro vývoj metod zpětnovazebného učení agentů, nic nám nebrání a je velmi jednoduché namísto

toho využít vlastního agenta, který je vyvíjen pomocí evolučních algoritmů.

### **3.2.1 Roboti**

## **3.3 Genetické algoritmy**

Po porovnání různých možností modulů pro tvorbu a použití evolučních algoritmů v naší knihovně, jsme se rozhodli pro vlastní implementaci jednotlivých částí evolučních algoritmů (genetických operátorů) a jejich propojení mezi sebou. Důvodem je hlavně jednodušší zapojení do zbytku knihovny a snížení nároků na znalosti mnohdy složitých výše popsaných externích knihoven pro uživatele, který by případně mohl chtít si do naší knihovny dopsat vlastní kus evolučního algoritmu. Tímto způsobem, pokud bude chtít něco takového udělat, dojde-li k dodržení zdrojovým kódem stanovených pravidel, vlastní kus kódu (metoda popisující genetický operátor) bude možné hned bez problému využít při dalším vývoji robotů.

## **3.4 Implementace knihovny**

## **3.5 Grafické rozhraní**

## 4. Experimenty a výsledky

4.1 Vývoj řízení robotů

4.2 Vývoj řízení a morfologie robotů

4.3 Diskuze výsledků

# Závěr

# Seznam použité literatury

- COPPELIAROBOTICS. <https://www.coppeliarobotics.com>. URL <https://www.coppeliarobotics.com>. Robot simulation software.
- COUMANS, E. Bullet real-time physics simulation. URL <https://pybullet.org/wordpress/>. Accessed: March 19, 2023.
- DEAP, P. Deap documentation. URL <https://deap.readthedocs.io/>. Accessed: March 19, 2023.
- DEEPMIND (2021). Mujoco. URL <https://mujoco.org/>. Accessed: March 26, 2023.
- EPLEX. URL <http://eplex.cs.ucf.edu/hyperNEATpage/>. Accessed: April 10, 2023.
- EREZ, T., TASSA, Y. a TODOROV, E. (2015). Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *2015 IEEE international conference on robotics and automation (ICRA)*, pages 4397–4404. IEEE.
- FORTIN, F.-A., DE RAINVILLE, F.-M., GARDNER, M.-A. G., PARIZEAU, M. a GAGNÉ, C. (2012). Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, **13**(1), 2171–2175.
- GARRETT, A. (2012). Inspyred: Bio-inspired algorithms in python. URL <https://pythonhosted.org/inspyred/>. Accessed: March 26, 2023.
- GOMEZ, F., SCHMIDHUBER, J., MIIKKULAINEN, R. a MITCHELL, M. (2008). Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, **9**(5).
- IZADI, E. a BEZUIJEN, A. (2018). Simulating direct shear tests with the bullet physics library: A validation study. *PLOS one*, **13**(4), e0195073.
- KOENIG, N. a HOWARD, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE.
- LEE, J., X. GREY, M., HA, S., KUNZ, T., JAIN, S., YE, Y., S. SRINIVASA, S., STILMAN, M. a KAREN LIU, C. (2018). Dart: Dynamic animation and robotics toolkit. *The Journal of Open Source Software*, **3**(22), 500.
- LEHMAN, J. a MIIKKULAINEN, R. (2013). Neuroevolution. *Scholarpedia*, **8**(6), 30977. doi: 10.4249/scholarpedia.30977. revision #137053.
- MICHEL, O. (2004). Cyberbotics ltd. webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, **1**(1), 5.
- NOGUEIRA, L. (2014). Comparative analysis between gazebo and v-rep robotic simulators. *Seminario Interno de Cognicao Artificial-SICA*, **2014**(5), 2.

- OPENROBOTICS. URL <https://gazebosim.org/>. Accessed: March 26, 2023.
- ROHMER, E., SINGH, S. P. N. a FREESE, M. (2013). Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*. [www.coppeliarobotics.com](http://www.coppeliarobotics.com).
- SALIMANS, T., HO, J., CHEN, X., SIDOR, S. a SUTSKEVER, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.
- SMITH, R. URL <http://ode.org/>. Accessed: March 26, 2023.
- SMITH, R. A KOL. (2007). Open dynamics engine.
- STANLEY, K. O. a MIIKKULAINEN, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, **10**(2), 99–127.
- STANLEY, K. O., D’AMBROSIO, D. B. a GAUCI, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, **15**(2), 185–212.
- TODOROV, E., EREZ, T. a TASSA, Y. (2012). Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE.
- TONDA, A. (2020). Inspyred: Bio-inspired algorithms in python. *Genetic Programming and Evolvable Machines*, **21**(1-2), 269–272.
- WEBOTS. <http://www.cyberbotics.com>. URL <http://www.cyberbotics.com>. Open-source Mobile Robot Simulation Software.
- YOON, J., SON, B. a LEE, D. (2023). Comparative study of physics engines for robot simulation with mechanical interaction. *Applied Sciences*, **13**(2), 680.



# A. Přílohy

## A.1 První příloha