GPU-Accelerated Differential Evolution for Robotic Planning and Control

Yandong Luo

July 31, 2025

1 Proposed Method

Robotic planning and control problems are often formulated as optimization tasks with both discrete and continuous variables. Common problem types include Mixed-Integer Programming (MIP) for task scheduling and mode switching, Mixed-Integer Quadratic Programming (MIQP) for footstep and contact planning, and Nonlinear Programming (NLP) for trajectory optimization and motion control. To handle non-convexity and ensure real-time performance, many of these problems are approximated or linearized using sequential convex programming or other relaxation techniques. These diverse optimization formulations expose several fundamental challenges in robotic planning and control:

- **High Computational Cost**: Many optimization-based methods require solving nonlinear, non-convex problems in real-time, which is computationally intensive, especially for high-DOF robots or whole-body control.
- **Non-differentiability**: Contact-rich tasks, discrete decisions, and hybrid dynamics often lead to non-differentiable cost functions, making gradient-based methods less effective or even infeasible.
- **Local Minima**: Traditional trajectory optimization is prone to local minima, particularly in cluttered or dynamic environments.
- **Limited Real-time Performance**: Real-time constraints in robotic control demand fast computation, but current methods often struggle to meet strict latency requirements without sacrificing solution quality.

To address these challenges, We propose a general GPU-accelerated framework integrating differential evolution and Bézier parameterization for efficient and smooth control optimization. Unlike conventional gradient-based optimizers or task-specific solvers, the proposed framework handles complex, high-dimensional, and non-differentiable optimization problems in a unified manner. By leveraging GPU parallelism, it enables large-scale population updates and fitness evaluations simultaneously, significantly reducing computational time and making real-time applications feasible. The contributions of this work can be summarized as follows:

- **Unified, Gradient-Free Optimization Framework**: We propose a GPU-accelerated SHADE and Bézier curve based optimization framework that handles MIP, MIQP, non-convex, and non-differentiable motion planning problems without gradients.

- **Task- and Model-Agnostic Generality**: The same optimization pipeline applies to diverse robotic tasks by simply changing the dynamics model or cost function, without modifying solver internals.
- **Broad Experimental Validation**: The generality of the proposed framework is demonstrated through experiments on three representative robotic domains: nonlinear cart-pole swing-up control, humanoid locomotion using a Linear Inverted Pendulum Model (LIPM) with discrete footstep planning, and autonomous vehicle trajectory planning under full dynamic constraints.

2 Differential Evolution

Differential Evolution (DE), proposed by Storn and Price in 1995, is a type of evolutionary algorithm mainly used to find the global optimum of functions defined in continuous parameter spaces, where the functions can be nonlinear, non-convex, non-differentiable, and multi-modal. This algorithm determines the search direction and step size based on the differences among individuals in the population. It is simple to implement, requires few control parameters, and has strong robustness. According to Table 1, evolutionary algorithms, especially the Success-History based Adaptive Differential Evolution (SHADE) family, have consistently ranked in the top three in the Congress on Evolutionary Computation over the past two decades. However, DE does not utilize gradient information and relies on iterative population updates, where each individual is modified and evaluated one by one, leading to low computational efficiency on CPUs, especially for large-scale problems.

Year	**1st**	**2nd**	**3rd**
2005	N/A	N/A	N/A
2006	ε**DE**ag	DMS-PSO	PCP
2007	N/A	N/A	N/A
2008	MTS	LSEDA-gl	j**DE**dynNP-F
2009	j**DE**	-	-
2010	ε**DE**ag	ECHT	j**DE**soco
2011	GA-MPC	**DE**-Λ_{cr}	SAMODE
2012	N/A	N/A	N/A
2013	NBIPOPaCMA	icmaesils	DRMA-LSCh-CMA
2014	L-**SHADE**	N/A	N/A
2015	SPS-L-**SHADE**-EIG	**DE**sPA	MVMO, L**SHADE**-ND
2016	N/A	N/A	N/A
2017	EBOwithCMAR	jSO	L**SHADE**-cnEpSin
2018	IU**DE**	MA-ES	L**SHADE**-IEpsilon
2019	j**DE**100	DISHchain1e+12	Hy**DE**-DF, SOMA-T3A
2020	IMO**DE**	AGSK	j2020
2021	APGSK_IMO**DE**	Made**DE**	NL-**SHADE**-RSP
2022	EAeigN100-10	NL-**SHADE**-LBC	NL-**SHADE**-RSP-MID
2023	ML_EA	L**SHADE**	AV_L**SHADE**_2

Table 1: Single-objective continuous optimization problem results from the Congress on Evolutionary Computation (CEC), 2005–2023.

To address these limitations, we propose a GPU-based SHADE framework that reduces the time complexity from $\mathcal{O}(G_{\max} \times PS \times D)$ on CPUs to $\mathcal{O}(G_{\max})$ under ideal parallelism, as illustrated in Algorithm 1. Beyond this asymptotic improvement, the massive parallelism of GPUs accelerates evolutionary algorithms in multiple aspects, including population-level parallel updates, batch fitness evaluations, and concurrent evolutionary operations such as mutation, crossover, and selection.

- Modern GPU enable evolutionary algorithms to efficiently process large-scale populations and evaluate multiple solution candidates in parallel. This capability comes from the hardware design of GPU, which support massive concurrency for example, the NVIDIA A100 features 108 Streaming Multiprocessors (SMs), each capable of running up to 2048 threads simultaneously.
- Parallel fitness evaluations and vector operations further reduce the computational overhead, making DE scalable to high-dimensional and computationally expensive problems.

Algorithm 1: Comparison of CPU-SHADE and CUDA-SHADE.
CPU time complexity: $\mathcal{O}(G_{\max} \times PS \times D)$;
CUDA time complexity: $\mathcal{O}(G_{\max})$

```
Notation: $PS$: population size; $D$: problem dimension; $G_{\max}$: maximum generation;
$MF$, $MCR$: memory of historical $F$ and $CR$; $SF$, $SCR$: successful $F$ and $CR$ archive;
$x_{i,g}$: individual $i$ at generation $g$; $x_{pbest,g}$: top-ranked individual; $A$: external archive;
$v_{i,g+1}$: mutant vector; $u_{i,g+1}$: trial vector; $CF(x)$: objective function.
Input: $PS$, $D$, $CF(x)$, $G_{max}$, archive $A = \varnothing$
Output: $x_{best,g}$
Initialize $MF$, $MCR$, $SF$, $SCR$
if CPU-SHADE then
    Generate population $P$ with $PS$ members (sequential)
if CUDA-SHADE then
    Generate population $P$ with $PS$ members in parallel
Compute $x_{best,g}$ (CPU: sequential; GPU: parallel reduction)
for each generation $g$ do
    if CPU-SHADE then
        for each member $x_{i,g}$ do
            Select $MF_k$, $CR_k$ from $MF$, $MCR$
            $F_i \sim \mathcal{N}(MF_k, 0.1)$, $CR_i \sim \mathcal{N}(CR_k, 0.1)$
            Select $x_{pbest,g}$, $x_{r1,g}$, $x_{r2,g}$
                $$v_{i,g+1} = x_{i,g} + F_i(x_{r1,g} - x_{r2,g}) + F_i(x_{pbest,g} - x_{i,g})$$
                $$u_{j,i,g+1} = \begin{cases} v_{j,i,g+1}, & \text{if } rand_j \leq CR \text{ or } j = rnb(i) \\ x_{j,i,g}, & \text{otherwise} \end{cases}$$
            Evaluate $CF(u_{i,g+1})$ sequentially
            Perform selection
    if CUDA-SHADE then
        In parallel for all $x_{i,g}$:
            Select $MF_k$, $CR_k$ from $MF$, $MCR$
            $F_i \sim \mathcal{N}(MF_k, 0.1)$, $CR_i \sim \mathcal{N}(CR_k, 0.1)$
            Select $x_{pbest,g}$, $x_{r1,g}$, $x_{r2,g}$
                $$v_{i,g+1} = x_{i,g} + F_i(x_{r1,g} - x_{r2,g}) + F_i(x_{pbest,g} - x_{i,g})$$
                $$u_{j,i,g+1} = \begin{cases} v_{j,i,g+1}, & \text{if } rand_j \leq CR \text{ or } j = rnb(i) \\ x_{j,i,g}, & \text{otherwise} \end{cases}$$
            Evaluate $CF(u_{i,g+1})$ in parallel
            Perform selection
    Update $SF$, $SCR$, $MF$, $MCR$, $A$, $PS$ (GPU: parallel reduction or atomic ops)
return $x_{best,g}$
```

3 Related Work

GPU-accelerated optimization has become an important research direction for robotic planning and control due to its potential to solve large-scale problems under real-time constraints.
A representative example of GPU-based SQP optimization is MPCGPU: Real-Time Nonlinear Model Predictive Control through Preconditioned Conjugate Gradient on the GPU [1], which formulates nonlinear MPC as a sequence of quadratic programs and solves them in real time using a GPU-accelerated Preconditioned Conjugate Gradient (PCG) method. The method constructs the Karush-Kuhn-Tucker (KKT) system at each iteration, applies a Schur complement to reduce system size, and solves the resulting condensed system of the form $S\lambda^* = \gamma$ using PCG. The approach achieves real-time performance by leveraging the structure of sparse linear systems and GPU-parallel matrix-vector operations.

$$S\lambda^* = \gamma$$

This system is efficiently solved on GPUs by leveraging parallel matrix-vector products and sparse linear algebra operations, enabling real-time execution. Although MPCGPU targets nonlinear optimization problems, it relies on sequential local approximations through linearization and quadratic expansion, following a standard SQP framework. As a result, its performance is sensitive to initializations and may degrade in problems with severe non-convexity or complex solution landscapes, where local approximations become inaccurate. In such cases, convergence to suboptimal solutions or poor local minima is likely.

3.1 MPCGPU

3.1.1 Sequential Quadratic Programming Formulation

Original Problem (Nonlinear Formulation) Given the discrete-time dynamics:

$$x_{k+1} = f(x_k, u_k, h), \quad x_0 = x_s$$

Cost function:

$$J(X, U) = \ell_f(x_N) + \sum_{k=0}^{N-1} \ell(x_k, u_k)$$

Local Quadratic Approximation (Sequential Quadratic Programming Idea) The Sequential Quadratic Programming (SQP) method works by **linearizing the nonlinear constraints**, such as system dynamics, through first-order Taylor expansion, and **quadraticizing the nonlinear cost function** via second-order Taylor expansion. Both expansions are performed **around a nominal trajectory** $(x_k^{\text{nom}}, u_k^{\text{nom}})$, which serves as the current reference solution in the iterative process.
Dynamics Linearization:

$$\delta x_{k+1} \approx A_k \delta x_k + B_k \delta u_k$$

where

$$A_k = \left.\frac{\partial f}{\partial x}\right|_{(x_k^{\text{nom}}, u_k^{\text{nom}})}, \quad B_k = \left.\frac{\partial f}{\partial u}\right|_{(x_k^{\text{nom}}, u_k^{\text{nom}})}$$

$$\delta x_k = x_k - x_k^{nom}, \quad \delta u_k = u_k - u_k^{nom}$$

Cost Quadraticization:
Expand the cost to second-order:
For stage cost:

$$\ell(x_k, u_k) \approx \ell(x_k^{\text{nom}}, u_k^{\text{nom}}) + \nabla_x \ell^T \delta x_k + \nabla_u \ell^T \delta u_k + \frac{1}{2}\delta x_k^T Q_k \delta x_k + \frac{1}{2}\delta u_k^T R_k \delta u_k$$

For terminal cost:

$$\ell_f(x_N) \approx \ell_f(x_N^{\text{nom}}) + \nabla_x \ell_f^T \delta x_N + \frac{1}{2}\delta x_N^T Q_N \delta x_N$$

Quadratic Program Formulation Put it all together:
Objective Function:
Minimize w.r.t. $\delta X, \delta U$:

$$\begin{aligned} \min_{\delta X, \delta U} \quad & \frac{1}{2}\delta x_N^T Q_N \delta x_N + q_N^T \delta x_N \\ & + \sum_{k=0}^{N-1} \left(\frac{1}{2}\delta x_k^T Q_k \delta x_k + q_k^T \delta x_k + \frac{1}{2}\delta u_k^T R_k \delta u_k + r_k^T \delta u_k \right) \end{aligned}$$

Constraints:
Initial condition:

$$\delta x_0 = x_s - x_0^{\text{nom}}$$

Linearized dynamics:

$$\delta x_{k+1} = f(x_k^{\text{nom}}, u_k^{\text{nom}}) + A_k \delta x_k + B_k \delta u_k - x_{k+1}$$

This ensures deviation consistency.

3.1.2 KKT System Formulation

To solve the Quadratic Program, the Karush-Kuhn-Tucker (KKT) conditions are applied. The KKT system couples the cost function and constraints into a single linear system.
The system is written as:

$$\begin{bmatrix} G & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} -\delta Z \\ \lambda \end{bmatrix} = \begin{bmatrix} g \\ c \end{bmatrix}$$

where:

- δZ is the stacked vector of decision variable deviations:

$$\delta Z = \begin{bmatrix} \delta x_0 \\ \delta u_0 \\ \delta x_1 \\ \delta u_1 \\ \cdots \\ \delta x_{N-1} \\ \delta u_{N-1} \\ \delta x_N \end{bmatrix}$$

- G is the block diagonal Hessian matrix of the cost:

$$G = \text{blockdiag}(Q_0, R_0, \ldots, Q_{N-1}, R_{N-1}, Q_N)$$

- g is the stacked gradient vector of the cost:

$$g = \begin{bmatrix} q_0 \\ r_0 \\ q_1 \\ r_1 \\ \cdots \\ q_{N-1} \\ r_{N-1} \\ q_N \end{bmatrix}$$

- C encodes the linearized system constraints:

$$C = \begin{bmatrix} I & 0 & 0 & \cdots & 0 & \\ -A_0 & -B_0 & I & \cdots & 0 & \\ 0 & 0 & -A_1 & -B_1 & \cdots & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ 0 & \cdots & 0 & -A_{N-1} & -B_{N-1} & I \end{bmatrix}$$

- c is the defect vector, representing the dynamics residuals:

$$c = \begin{bmatrix} x_0 - x_s \\ x_1^{\text{nom}} - f(x_0^{\text{nom}}, u_0^{\text{nom}}) \\ x_2^{\text{nom}} - f(x_1^{\text{nom}}, u_1^{\text{nom}}) \\ \vdots \\ x_N^{\text{nom}} - f(x_{N-1}^{\text{nom}}, u_{N-1}^{\text{nom}}) \end{bmatrix}$$

Solution:
Solving the KKT system yields the optimal increments δZ^*, which are then used to update the trajectory:

$$x_k^{\text{new}} = x_k^{\text{nom}} + \delta x_k^*, \quad u_k^{\text{new}} = u_k^{\text{nom}} + \delta u_k^*$$

This iterative process continues until convergence is achieved.

3.1.3 The Schur Complement Method

To efficiently solve the KKT system from the previous section, the **Schur Complement Method** is applied. This reduces the computational complexity by eliminating the decision variables δZ and solving a smaller system for the dual variables λ.

Step 1: Form the Schur Complement System Given the KKT system:

$$\begin{bmatrix} G & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} -\delta Z \\ \lambda \end{bmatrix} = \begin{bmatrix} g \\ c \end{bmatrix}$$

The Schur complement S and right-hand side γ are defined as:

$$S = -CG^{-1}C^T, \quad \gamma = c - CG^{-1}g$$

Then, solve for λ^*:

$$S\lambda^* = \gamma$$

Step 2: Recover δZ^* Once λ^* is obtained, the optimal trajectory increments δZ^* can be computed via:

$$\delta Z^* = -G^{-1}\left(g - C^T\lambda^*\right)$$

Matrix Definitions To explicitly construct S, define the following intermediate terms:

$$\theta_k = A_k Q_k^{-1} A_k^T + B_k R_k^{-1} B_k^T + Q_{k+1}^{-1}$$

$$\phi_k = -A_k Q_k^{-1}$$

$$\zeta_k = -A_k Q_k^{-1} q_k - B_k R_k^{-1} r_k + Q_{k+1}^{-1} q_{k+1}$$

Schur Complement Matrix Structure Using the above definitions, the Schur complement matrix S becomes a block-banded symmetric matrix:

$$S = -\begin{bmatrix} Q_0^{-1} & \phi_0^T & & & \\ \phi_0 & \theta_0 & \phi_1^T & & \\ & \phi_1 & \theta_1 & \ddots & \\ & & \ddots & \ddots & \\ & & & & \theta_{N-1} \end{bmatrix}$$

Right-Hand Side The corresponding right-hand side γ is:

$$\gamma = c - \begin{bmatrix} Q_0^{-1} q_0 \\ \zeta_0 \\ \zeta_1 \\ \vdots \\ \zeta_{N-1} \end{bmatrix}$$

Advantages This method reduces the size of the system to be solved and exploits the problem's structure, allowing for efficient parallelization and faster solution, particularly suitable for GPU implementations.

3.1.4 Iterative Methods

To solve the Schur complement system:

$$S\lambda^* = \gamma$$

iterative methods are employed due to the large-scale and sparse structure of S. The most commonly used method is the **Preconditioned Conjugate Gradient (PCG)** algorithm, which is especially suitable for GPU implementations due to its reliance on matrix-vector products rather than direct matrix inversion.

Conjugate Gradient Method The conjugate gradient (CG) method iteratively refines the solution λ by minimizing the quadratic form associated with S. The convergence rate of CG is directly related to the condition number of S. A wide spread of eigenvalues leads to slower convergence.

Preconditioning To accelerate convergence, a **preconditioner** $\Phi \approx S$ is introduced. This transforms the original system into a better-conditioned equivalent system:

$$\Phi^{-1}S\lambda^* = \Phi^{-1}\gamma$$

The preconditioner reduces the spread of the eigenvalues, improving numerical properties and reducing iteration counts.

Preconditioned Conjugate Gradient Algorithm The following summarizes the PCG algorithm used to solve $S\lambda^* = \gamma$:

Algorithm 2: Preconditioned Conjugate Gradient (PCG)

```
Input: $S$, $\Phi^{-1}$, $\gamma$, initial $\lambda$, tolerance $\epsilon$
Output: Solution $\lambda^*$
$r \leftarrow \gamma - S\lambda$
$\tilde{r} \leftarrow \Phi^{-1}r$
$p \leftarrow \tilde{r}$
$\eta \leftarrow r^T\tilde{r}$
for $i = 1$ to $max\_iter$ do
    $\alpha \leftarrow \eta/(p^TSp)$
    $r \leftarrow r - \alpha Sp$
    $\lambda \leftarrow \lambda + \alpha p$
    $\tilde{r} \leftarrow \Phi^{-1}r$
    $\eta' \leftarrow r^T\tilde{r}$
    if $\eta' < \epsilon$ then
        return $\lambda$
    $\beta \leftarrow \eta'/\eta$
    $p \leftarrow \tilde{r} + \beta p$
    $\eta \leftarrow \eta'$
return $\lambda$
```

Advantages

- **Memory Efficient:** Only requires matrix-vector products.
- **GPU Friendly:** Exploits parallel reductions and sparse operations.
- **Scalable:** Suitable for large-scale systems in MPC and optimal control.

4 Differential Evolution

The main process of differential evolution can be described as shown in Figure 1

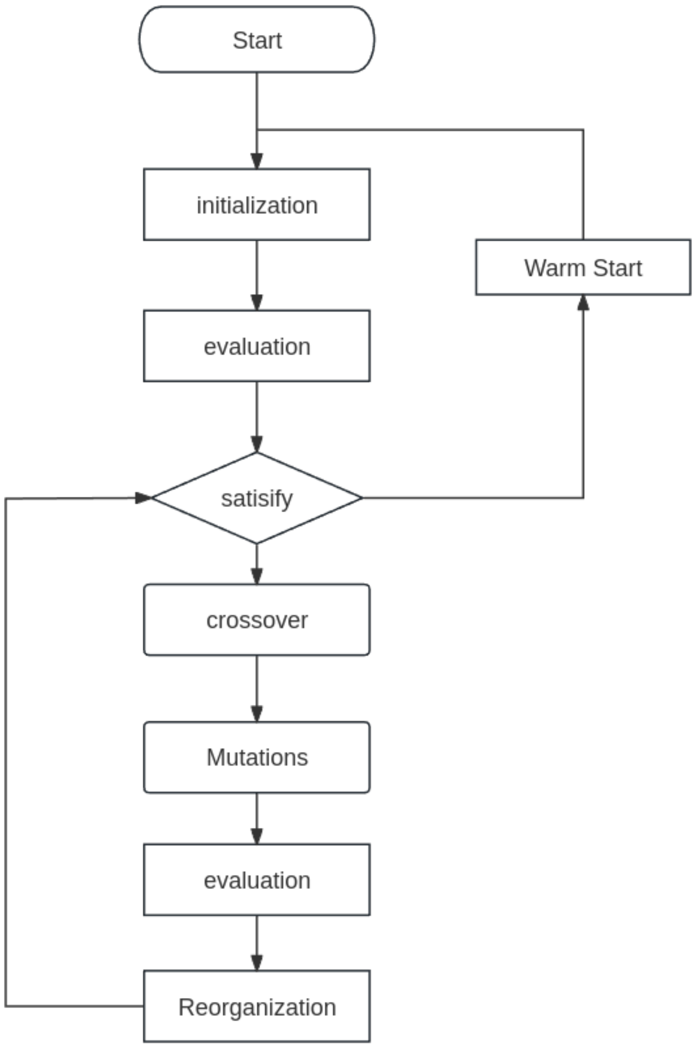

Figure 1: The main process of Differential Evolution

4.1 Start & Initialization

In the Start module, the optimization problem variables are encoded into a single vector, each representing a potential feasible solution, as illustrated in Figure 2. Each vector is also called an individual. In addition, the constraints of the problem are formulated and represented in matrix form to facilitate subsequent computations. In the initialization module, each individual will randomly initialize the vector within the bounds of the constraints. Each individual represents a guess for the solution.

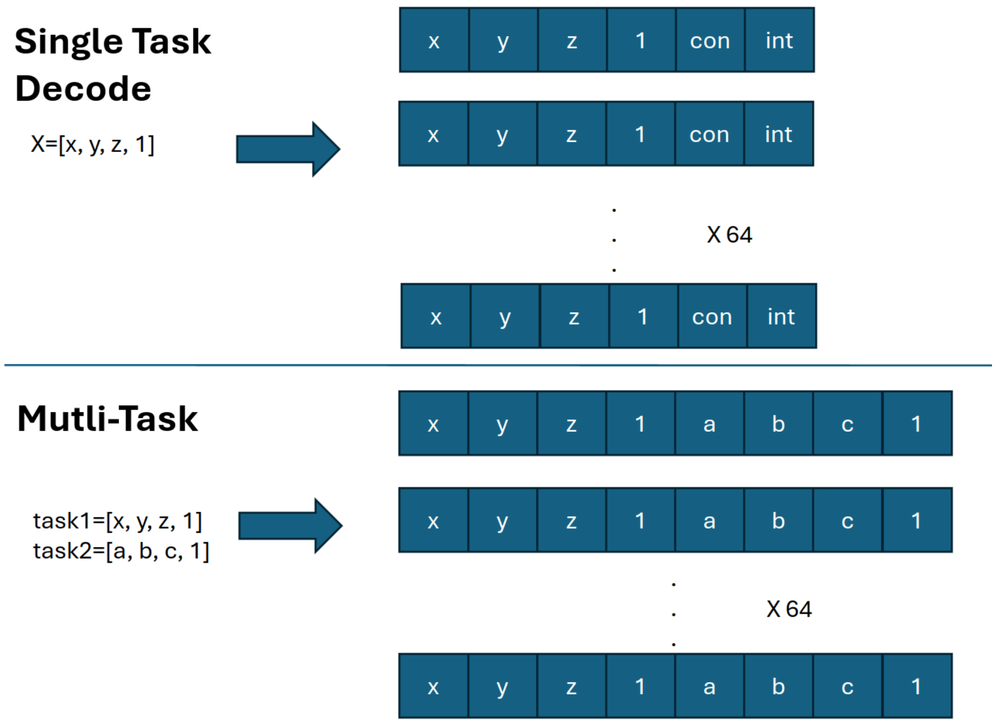

Figure 2: Decode variable as the vector

4.2 Evaluation

In the evaluation module, the constrained problem is reformulated into an unconstrained problem using the augmented Lagrangian method. The Lagrange multipliers act as penalty weights for constraint violations. Using matrix multiplication, the fitness of each individual is computed, incorporating both the constraints and the objective function. This fitness value reflects the feasibility and precision of the solution.
In addition, this module has the capability to incorporate dynamic perception information, enabling the robot to evaluate its surrounding environment, such as performing parallel assessments of multiple obstacles. It can also integrate IMU data to facilitate the evaluation of the robot's posture. For control-related tasks, the module can further support the evaluation of future multi-step motion control. Notably, all these computations are designed to be executed efficiently through GPU-based multi-threading, resulting in a highly efficient and scalable evaluation process.

4.3 Evolution

When the fitness of the current best individual fails to meet the convergence criteria, the iterative update process of the Differential Evolution (DE) algorithm is initiated. The entire iterative update process is executed in parallel on the multi-threaded GPU. Each element of each individual is assigned to a corresponding thread, where the evolution steps of the differential evolution algorithm are independently performed. This design enables the framework to achieve a high degree of parallelism at both the multi-task level and the task-solving level as shown in Figure 3

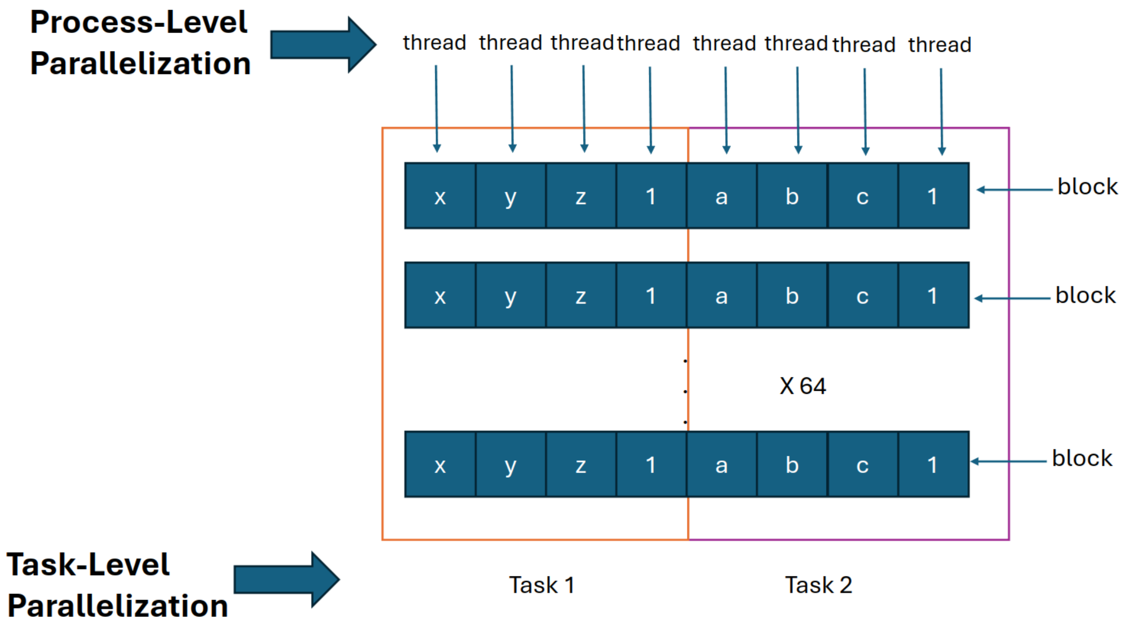

Figure 3: Parallelization in process-level and task-level.

4.4 Crossover

The first step in this process is the crossover operation, which plays a critical role in determining which elements of an individual are selected for mutation to generate a new candidate solution. The specific mechanism of the crossover operation is described as follows:

$$u_{j,i} = \begin{cases} v_{j,i} & \text{if } \mathrm{rand}[0,1) \leq CR_i, \\ x_{j,i} & \text{otherwise.} \end{cases}$$

where, j represents the j-th dimension of an individual, i denotes the i-th individual in the entire population, CR is the random number, and $x_{j,i}$ corresponds to the j-th element of the i-th original individual. Similarly, $v_{j,i}$ represents the corresponding element of the new individual.

In SHADE, the CR value is not fixed but dynamically adjusted. It is drawn from a memory archive that stores successful parameters from previous iterations. By utilizing parameters that performed well in past iterations, the algorithm adaptively selects more effective crossover probabilities, enhancing its performance over time.

4.5 Mutation

In the mutation module of the SHADE algorithm, a mutant vector is generated for each individual based on both historical success and current population dynamics. The mutation formula in SHADE is defined as:

$$v_{j,i} = x_{j,i} + F_i \cdot (x_{p,\text{best}} - x_{j,i}) + F_i \cdot (x_{r1} - x_{r2}),$$

where $v_{j,i}$ is the j-th element of the mutant vector for individual i, and $x_{j,i}$ represents the j-th element of the current individual i. The term $x_{p,\text{best}}$ is a randomly selected individual from the top p-percent of the population ranked by fitness, while x_{r1} and x_{r2} are two randomly selected individuals from the population, ensuring $r1 \neq r2 \neq i$. The scaling factor F_i is defined as $F_i = \text{rand}(M_F, r_i, 0.1)$, where M_F is sampled from a historical memory archive that records successful scaling factors from previous iterations.

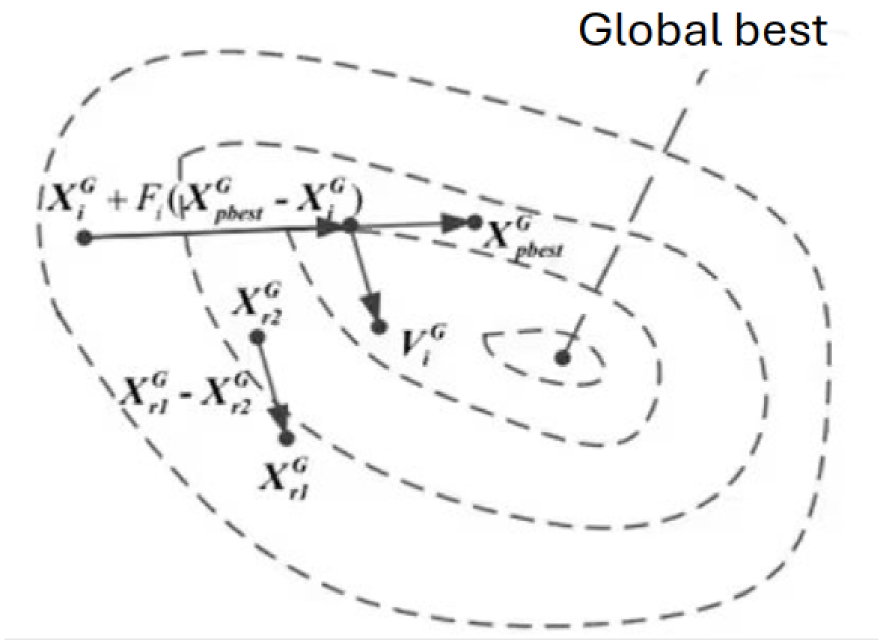

Figure 4: Mutation process [2]

This mutation process integrates several key mechanisms to enhance the optimization process. First, the selection of $x_{p,\text{best}}$ from the top-performing individuals directs the search toward promising regions in the solution space, promoting exploitation. Second, the adaptive sampling of F_i from the memory archive allows the algorithm to learn from historical successes and dynamically adjust the mutation behavior, ensuring robust performance across iterations. Lastly, the difference vectors $(x_{p,\text{best}} - x_{j,i})$ and $(x_{r1} - x_{r2})$ balance exploitation and exploration by introducing both focused improvement and diversity into the search process as shown in Figure 4. These mechanisms together make the mutation operation in SHADE highly adaptive and effective for solving high-dimensional optimization problems.

4.6 Evaluation & Reorganize

After the mutation step, the newly generated individuals are evaluated to assess their fitness and determine the quality of the new population. This evaluation is performed to ensure that only promising solutions are considered for the subsequent selection process.

In the Reorganization module, the evaluated individuals are compared to the current population to select the better-performing solutions. The update process for each individual is defined as:

$$x_{i,G+1} = \begin{cases} u_{i,G}, & \text{if } f(u_{i,G}) \leq f(x_{i,G}), \\ x_{i,G}, & \text{otherwise.} \end{cases}$$

Here, $x_{i,G}$ represents the i-th individual in generation G, $u_{i,G}$ is the new individual generated through mutation, and f is the fitness function. If the fitness of $u_{i,G}$ is better (or equal), it replaces $x_{i,G}$ in the next generation; otherwise, the original individual is retained.

To enhance the algorithm's adaptability, the memory archives for the crossover rate (CR) and scaling factor (F) are updated based on the success of evolved individuals. The update rules are as follows:

$$M_{CR,k,G+1} = \begin{cases} \frac{\sum_{k=1}^{S_{CR}} w_k S_{CR,k}^2}{\sum_{k=1}^{S_{CR}} w_k S_{CR,k}}, & \text{if successful,} \\ M_{CR,k,G}, & \text{otherwise.} \end{cases}$$

$$M_{F,k,G+1} = \begin{cases} \frac{\sum_{k=1}^{S_F} w_k S_{F,k}^2}{\sum_{k=1}^{S_F} w_k S_{F,k}}, & \text{if successful,} \\ M_{F,k,G}, & \text{otherwise.} \end{cases}$$

Here, S_{CR} and S_F denote the sets of successful CR and F values, respectively, while the weights w_k are computed as:

$$w_k = \frac{\Delta f_k}{\sum_{k=1}^{|S_{CR}|} \Delta f_k},$$

where $\Delta f_k = |f(\mu_{k,G}) - f(x_k, G)|$ quantifies the improvement in fitness for successful individuals. This weighting ensures that more significant improvements have a greater impact on the memory updates.

By dynamically updating these memory archives, the algorithm adapts its parameters based on historical success, improving its convergence and robustness over successive generations.

4.7 Warm Start

The GPU-accelerated Differential Evolution (DE) algorithm exhibits high efficiency in solving optimization problems, demonstrating significant potential for fast solutions in large-scale scenarios. By leveraging GPU-based parallelism, the algorithm is capable of handling complex, high-dimensional optimization tasks with reduced computational overhead, making it well-suited for applications requiring real-time performance.

To further enhance the framework's efficiency and accuracy in continuous and dynamic robotic tasks, a warm start module can be incorporated. In such environments, where optimization problems evolve over time, the solution to the previous task often provides valuable guidance for solving the current problem. The warm start module initializes the population by utilizing the previous solution as a reference point. Around this reference, new individuals are generated by introducing random perturbations within a defined neighborhood. This approach enables the population to retain diversity while benefiting from prior knowledge, facilitating faster convergence to high-quality solutions.

By adapting previous solutions to the current context, the warm start module significantly reduces the computational cost associated with reinitialization and accelerates optimization in dynamic environments. This strategy enhances the framework's ability to efficiently and effectively solve sequential tasks, making it highly suitable for real-time robotic systems operating in continuous and dynamic settings.

5 Experiment

To demonstrate the scalability and generality of our proposed algorithm, we evaluate it on three representative tasks with increasing complexity: from low-dimensional control with contact constraints to high-dimensional hybrid planning under nonlinear dynamics. These tasks cover cart-pole balancing, humanoid locomotion, and autonomous vehicle planning.

- **CartPole:** We consider a nonlinear cart-pole system under strong stochastic noise and dynamic wall contacts. The environment features continuously moving walls, and the controller uses the full nonlinear dynamics for planning. Our method successfully balances the pole and enforces contact constraints, achieving over $10\times$ speedup compared to Gurobi-based solvers.
- **Humanoid-LIMP:** This experiment focuses on whole-body humanoid control using centroidal momentum dynamics. We perform LIPM-based motion planning with nonlinear constraints and discrete footstep switching, formulated as a mixed-integer optimization problem with obstacle avoidance. Our solver achieves stable and feasible locomotion while running up to $20\times$ faster than Gurobi.
- **Autonomous Vehicle:** We scale our approach to a high-dimensional autonomous driving task using a full nonlinear vehicle model with drag, friction, tire slip, and actuator limits. The planner handles geometric constraints from road boundaries, traffic rules, and dynamic/static agents. It enables parallel multi-behavior planning including lane-keeping, lane-change, and overtaking. This scenario highlights the extensibility of our algorithm to real-world autonomous systems.

5.1 Cart-Pole System

As shown in Figure 5, the coordination of this system is located at the center. The right side is positive, left side is negative. The pole swings leftwards for positive angles and rightwards for negative angles.

5.1.1 Linear MPC Model

The optimization uses a stage cost and terminal cost:

$$x_g[k] \text{ represents the control target for } x[k].$$

The matrices E_k and E_N are positive definite. X_k is the domain of $x[k]$. The system is formulated in a compact form as shown in Figure 7.

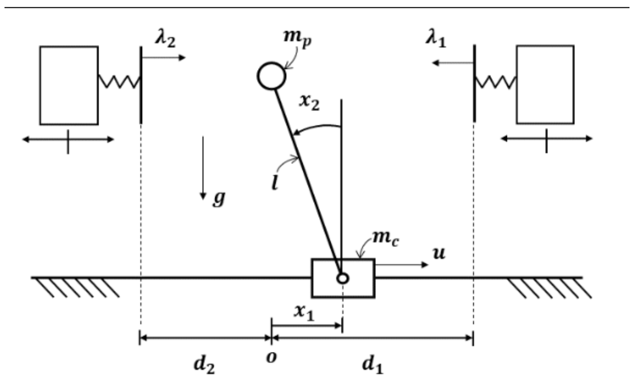

Figure 5: Cart-Pole System with moving soft contact walls

$$\begin{aligned}
\underset{\boldsymbol{x}[k]\in X_k,\ \boldsymbol{u}[k],\ \boldsymbol{\delta}[k]}{\text{minimize}} \quad & \sum_{k=0}^{N-1}(||\boldsymbol{x}[k]-\boldsymbol{x}_g[k]||^2_{\boldsymbol{Q}_k}+||\boldsymbol{u}[k]||^2_{\boldsymbol{R}_k})+ \\
& ||\boldsymbol{x}[N]-\boldsymbol{x}_g[N]||^2_{\boldsymbol{Q}_N} \\
\text{s.t.} \quad & \boldsymbol{x}[0]=\boldsymbol{x}_{in} \\
& \boldsymbol{x}[k+1]=\boldsymbol{E}\boldsymbol{x}[k]+\boldsymbol{F}\boldsymbol{u}[k]+\boldsymbol{G}\boldsymbol{\delta}[k] \\
& \boldsymbol{H}_1\boldsymbol{x}_k+\boldsymbol{H}_2\boldsymbol{u}[k]+\boldsymbol{H}_3\boldsymbol{\delta}[k]\le \boldsymbol{h}(\boldsymbol{\theta}) \\
& \boldsymbol{\delta}[k]\in\{0,1\}^{n_\delta},\ k=0,...,N-1
\end{aligned}$$

Figure 6: Model Predictive Control (MPC) Formulation for cart pole

Let the state variables be defined as:

$$x_1 = \text{position } (pos), \quad x_2 = \theta, \quad x_3 = v, \quad x_4 = \dot{\theta}$$

with a runtime step $\Delta t = 0.02$.
Here, λ is the wall force:

$$\lambda_1 \text{ from left wall}, \quad \lambda_2 \text{ from right wall}$$

Linear Model Constraints

- Cart position: $-0.6 \le x_1 \le 0.6$
- Pole angle: $-\frac{\pi}{2} \le x_2 \le \frac{\pi}{2}$
- Cart velocity:
$$-\frac{2\times 0.6}{\Delta t} \le x_3 \le \frac{2\times 0.6}{\Delta t}$$
- Angular velocity:
$$-\frac{\pi}{\Delta t} \le x_4 \le \frac{\pi}{\Delta t}$$
- Control input: $-20 \le u \le 20$

$$\dot{\mathbf{x}} = Ax + Bu + E\lambda$$

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{gm_p}{m_c} & 0 & 0 \\ 0 & \frac{g(m_c+m_p)}{lm_c} & 0 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ 0 \\ \frac{1}{m_c} \\ \frac{1}{lm_c} \end{bmatrix} u + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \frac{1}{lm_p} & \frac{1}{lm_p} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix}.$$

$$\mathbf{x}_{k+1} = \left(\mathbf{I} + \Delta t \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{gm_p}{m_c} & 0 & 0 \\ 0 & \frac{g(m_c+m_p)}{lm_c} & 0 & 0 \end{bmatrix} \right) \mathbf{x}_k + \Delta t \begin{bmatrix} 0 \\ 0 \\ \frac{1}{m_c} \\ \frac{1}{lm_c} \end{bmatrix} u_k + \Delta t \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \frac{1}{lm_p} & \frac{1}{lm_p} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix}.$$

Figure 7: Linearized Cart-Pole Model

- Contact forces:

$$0 \leq \lambda_1, \lambda_2 \leq 20$$

- Cart and pole position constraints:

$$x_1 - lx_2 \leq wall_{\text{right}}, \quad x_1 - lx_2 \geq wall_{\text{left}}$$

where $D_{\max} = 1.2$ is the maximum distance from the origin.

5.1.2 Nonlinear Dynamics

The nonlinear dynamics of the cart-pole system are given by:

$$a = \frac{-\dot{\theta}^2 lm_p \sin(\theta) + \frac{gm_p \sin(2\theta)}{2} + \lambda_1 \cos^2(\theta) - \lambda_1 - \lambda_2 \cos^2(\theta) + \lambda_2 + F}{m_c + m_p \sin^2(\theta)}$$

$$\ddot{\theta} = \frac{-\dot{\theta}^2 lm_p^2 \sin(2\theta)/2 + gm_c m_p \sin(\theta) + gm_p^2 \sin(\theta) + \lambda_1 m_c \cos(\theta) - \lambda_2 m_c \cos(\theta) + m_p F \cos(\theta)}{lm_p(m_c + m_p \sin^2(\theta))}$$

State Transfer

$$\begin{aligned} v_{t+1} &= v_t + a\Delta t \\ x_{t+1} &= x_t + v_t \Delta t + \frac{1}{2} a \Delta t^2 \\ \dot{\theta}_{t+1} &= \dot{\theta}_t + \ddot{\theta} \Delta t \\ \theta_{t+1} &= \theta_t + \dot{\theta}_t \Delta t + \frac{1}{2} \ddot{\theta} \Delta t^2 \end{aligned}$$

where:

- v_t: velocity at time t
- x_t: position at time t
- θ_t: angle at time t
- $\dot{\theta}_t$: angular velocity at time t

- $\ddot{\theta}$: angular acceleration
- Δt: time step
- a: linear acceleration

Contact with Wall

$$\begin{aligned} x_{\text{pole1}} &= -(l \sin(\theta) - x) \\ x_{\text{pole2}} &= l \sin(\theta) - x \end{aligned}$$

where x_{pole1} is for pole tilting to the right and x_{pole2} to the left.

Wall Contact Forces

$$\lambda_1 = \begin{cases} k_1(x_{\text{pole1}} - x_{\text{wall,R}}), & \text{if } x_{\text{pole1}} \geq x_{\text{wall,R}} \\ 0, & \text{otherwise} \end{cases}$$

$$\lambda_2 = \begin{cases} k_2(x_{\text{pole2}} - x_{\text{wall,L}}), & \text{if } x_{\text{pole2}} \leq x_{\text{wall,L}} \\ 0, & \text{otherwise} \end{cases}$$

References

[1] Emre Adabag, Miloni Atal, William Gerard, and Brian Plancher. Mpcgpu: Real-time nonlinear model predictive control through preconditioned conjugate gradient on the gpu. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9787–9794. IEEE, 2024.

[2] Sakshi Aggarwal and Krishn K. Mishra. X-mode: Extended multi-operator differential evolution algorithm. *Mathematics and Computers in Simulation*, 211:85–108, 2023.