iLQGames.jl: Rapidly Designing and Solving Differential Games in Julia

Lasse Peters[1][0000−0001−9008−7127] and Zachary N. Sunberg[2][0000−0001−9707−3035]

[1] Hamburg University of Technology, Hamburg, Germany
`lasse.peters@tuhh.de`
[2] University of Colorado Boulder, CO, USA
`zachary.sunberg@colorado.edu`

Abstract. In many problems that involve multiple decision making agents, optimal choices for each agent depend on the choices of others. Differential game theory provides a principled formalism for expressing these coupled interactions and recent work offers efficient approximations to solve these problems to non-cooperative equilibria. iLQGames.jl is a framework for designing and solving differential games, built around the iterative linear-quadratic method presented in [3]. It is written in the Julia programming language to allow flexible prototyping and integration with other research software, while leveraging the high-performance nature of the language to allow real-time execution. The open-source software package can be found at https://github.com/lassepe/iLQGames.jl.

Keywords: differential games · multi-agent systems · Julia · open-source.

1 Introduction and Related Work

In order for a robot to be truly autonomous it needs to be capable of interacting with environments that are not isolated, but rather shared with other agents. Naturally, these shared environments are manipulated by multiple decision making agents at a time and different agents may pursue different objectives. For any non-trivial interaction scenario, the success of a given agent depends not only on its own actions but also on the decisions of others. Therefore, agents must consider the effects of their actions on the behavior of others.

Dynamic game theory offers an expressive theoretical framework for formulating these types of interaction problems. Differential games are a special class of dynamic games that consider problems in continuous domain. In this framework, the evolution of the state is characterized by a differential equation which depends upon each players input, and the objectives of all players are expressed by their respective cost functions. The solution of a differential game to an appropriate equilibrium concept, e.g. a Nash equilibrium in the case of non-cooperative scenarios, provides a strategy assignment for *all* players. Game solutions may be used for centralized control of multiple agents or for (decentralized) control of an individual agent interacting with others [2, 8, 5]. Recent work has demonstrated the effectiveness of game theoretic planning approaches in several multi-agent

problems, such as human-robot interaction for intersection and highway driving [2, 3, 5] or multi-robot racing [7, 8, 9].

Unfortunately, the complexity involved with describing and solving these problems impedes application to scenarios characterized by high-dimensional states, real-time constraints and fast planning rates, such as robotics. Here, the term *complexity* is explicitly used with threefold meaning: First, the algorithmic time and space complexity of solution methods; Second, the challenges involved with implementing these algorithms efficiently; And third, the conceptual complexity of the interface used to describe and set up the problem. While the first two aspects are crucial for quick solution of the problem, we emphasize that the latter aspect is particularly important to admit quick iteration of different designs, e.g. to experiment with different cost structures that encode the behavior of each player.

In terms of computational complexity recent work offers efficient approximations to non-cooperative games [3, 7] and several works have demonstrated real-time performance of these algorithms in C++ implementations [3, 7, 8, 9]. However, to the best of our knowledge, only [3] provide a publicly available implementation of their solver[3] and little work has focused on providing flexible interfaces and tools for the design phase of differential games.

This work presents iLQGames.jl, a framework for designing and solving differential games using the iterative linear-quadratic (iLQ) method proposed in [3]. iLQGames.jl is written in the Julia programming language [1] and makes use of the language's genericity to provide a flexible interface that admits quick iteration of different problem designs and keep up with execution times of a comparable C++ implementation. This paper describes the key aspects of the framework that enable its flexibility and performance, and make it an effective tool for differential game research.

2   Architecture for Rapid Design and Solution

2.1   Rapid Design

When modelling a practical scenario of multi-agent interaction as a differential game, it may not be immediately clear what are suitable dynamics and costs to describe the problem. Therefore, iLQGames.jl provides a thin interface for describing differential games that allows users to set up a model in few lines of code. [4]

Two examples of differential games that are designed, solved and visualized using iLQGames.jl are given in Fig. 1. In these problems, each player is in control of a subsystem with 4D unicycle dynamics and the objective of each agent is to reach their respective goal while avoiding collisions with the wall or other agents. After providing the differential equation characterizing the game dynamics (here, a product system composed of multiple unicycle subsystems) and specifying the

---

[3] https://github.com/HJReachability/ilqgames

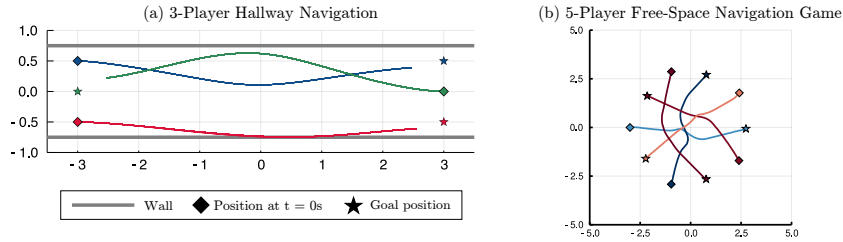[4] See https://github.com/lassepe/iLQGames.jl for a code example.

Fig. 1. Two differential games solved with iLQGames.jl: a 3-player collision avoidance problem in a hallway and a 5-player collision avoidance problem in free-space.

cost functions for each player (here, penalizing proximity, final distance to the goal location and control effort), the user can directly invoke the iLQ solver. Most notably, even though the iLQ solver is based on successive linear-quadratic (LQ) approximations of the game, the user does *not* need to hand-specify partial derivatives for linearization of the dynamics or quadratization of the costs. Instead, iLQGames.jl can compute LQ approximations efficiently via automatic differentiation of the game dynamics and player costs using [6].

Furthermore, being written in pure Julia — a language with strong focus on scientific programming — iLQGames.jl can directly be used with various other packages from the ecosystem. An example that demonstrates this advantage is presented in [5], where iLQGames.jl is combined with ParticleFilters.jl to reason about behavioural uncertainty of other players in differential games.

2.2 Rapid Solution

Despite being a high-level language that offers wide-ranging abstraction, the Julia compiler generates highly optimized code. Most notably for our use case, this allows iLQGames.jl to solve each LQ iterate very efficiently via a fully stack-allocated dynamic program implemented in a readable high-level style in less than 70 lines of code. In fact, for moderately sized games, this optimization allows our implementation to outperform the C++ implementation presented in [3].

Furthermore, the generic function dispatch mechanism used in Julia allows users to overload default implementations at various levels of the solver to make problem specific optimizations. For example, users can specify a custom method to perform LQ approximations once they have chosen a design for the problem.

Beyond that, iLQGames.jl supports exploitation of special structure of the dynamics to speed up computation. This aspect is realized via the `LinearizationStyle` trait concept. By default, a dynamical system is attributed the `JacobianLinearization` trait and automatic differentiation or a user-defined linearization is used to obtain LQ approximations of the game. However, if the dynamics are feedback linearizable, the user can optionally specify the `FeedbackLinearization` trait for a model to invoke a specialized version of the solver presented in [4]. This trait concept can be easily extended to other

special types of systems and thus allows users to seamlessly customize the solver with small local changes without the need to overload other parts of the routine.

3 Performance

The performance of iLQGames.jl is evaluated by benchmarking it on three problems against the C++ implementation presented in [3]. For additional reference, a Python implementation of an LQ solver is benchmarked as well. The benchmark problems are a minimal LQ game, a nonlinear nonquadratic collision avoidance problem similar to the examples depicted in Fig. 1, and a feedback linearized version of the latter.

Table 1 summarizes the benchmark results. The Python implementation for the LQ case is multiple orders of magnitude slower than the C++ version and iLQGames.jl and thus would not scale well to nonlinear nonquadratic problems. When utilizing manually specified partial derivatives to compute LQ approximations, as also done in the C++ version, iLQGames.jl outperforms the baseline. When using automatic differentiation iLQGames.jl still achieves moderate runtime and is sufficiently fast to evaluate different problem designs.

Table 1. Benchmark results. The tuple behind each problem indicates the number of players (P) and the dimensionality of the state (D). LQ-Python denotes a Python implementation of the dynamic program used at the inner loop of the iLQ game algorithm. iLQGames-C++ refers to the implementation used in [3]. iLQGames.jl-MD and iLQGames.jl-AD refer to our implementation, using manual differentiation and automatic differentiation, respectively. Each game is solved over a horizon of 100 time steps on a standard laptop.

	LQ (2P, 2D)	Nonlinear (3P, 12D)	FBLinearized (3P, 12D)
LQ-Python	20.800 ms	n/a	n/a
iLQGames-C++	0.3490 ms	16.27 ms	13.25 ms
iLQGames.jl-MD	0.0044 ms	7.19 ms	3.98 ms
iLQGames.jl-AD	n/a	63.57 ms	52.50 ms

4 Conclusion

iLQGames.jl is a framework for designing and solving differential games, built around the iterative linear-quadratic method presented in [3]. This manuscript provides an overview of the framework and discusses key design aspects that enable its flexibility and performance. iLQGames.jl provides a first step towards making differential games an easily accessible tool for multi-agent interaction research. Future directions include abstraction of a high-level problem interface that can be shared between multiple solvers to simplify the process of benchmarking algorithms against one another.

Bibliography

[1] Bezanson, J., Karpinski, S., Shah, V.B., Edelman, A.: Julia: A fast dynamic language for technical computing. arXiv:1209.5145 (2012)

[2] Fisac, J.F., Bronstein, E., Stefansson, E., Sadigh, D., Sastry, S.S., Dragan, A.D.: Hierarchical game-theoretic planning for autonomous vehicles. In: 2019 International Conference on Robotics and Automation (ICRA). pp. 9590–9596. IEEE (2019)

[3] Fridovich-Keil, D., Ratner, E., Dragan, A.D., Tomlin, C.J.: Efficient iterative linear-quadratic approximations for nonlinear multi-player general-sum differential games. arXiv:1909.04694 (2019)

[4] Fridovich-Keil, D., Rubies-Royo, V., Tomlin, C.J.: An iterative quadratic method for general-sum differential games with feedback linearizable dynamics. arXiv:1910.00681 (2019)

[5] Peters, L., Fridovich-Keil, D., Tomlin, C.J., Sunberg, Z.N.: Inference-based strategy alignment for general-sum differential games. arXiv:2002.04354 (2020)

[6] Revels, J., Lubin, M., Papamarkou, T.: Forward-mode automatic differentiation in Julia. arXiv:1607.07892 (2016)

[7] Spica, R., Falanga, D., Cristofalo, E., Montijano, E., Scaramuzza, D., Schwager, M.: A real-time game theoretic planner for autonomous two-player drone racing. arXiv:1801.02302 (2018)

[8] Wang, M., Wang, Z., Talbot, J., Gerdes, J.C., Schwager, M.: Game theoretic planning for self-driving cars in competitive scenarios. In: Robotics: Science & Systems (2019)

[9] Wang, Z., Spica, R., Schwager, M.: Game theoretic motion planning for multi-robot racing. In: Distributed Autonomous Robotic Systems, pp. 225–238. Springer (2019)