2024 European Control Conference (ECC)
June 25-28, 2024. Stockholm, Sweden

AdaptiveNLP: a Framework for Efficient Online Adaptability in NLP Structures for Optimal Control Problems

Louis Callens*, Joris Gillis*, Wilm Decré*, Jan Swevers*
*MECO Research Team, Department of Mechanical Engineering, KU Leuven, Belgium
Flanders Make @ KU Leuven, Leuven, Belgium
email: firstname.lastname@kuleuven.be

Abstract— **Direct methods that transcribe an Optimal Control Problem (OCP) to a Nonlinear Program (NLP) have proven effective to solve OCPs. Flexibility in this transcription that can adapt online to a changing environment by adding or removing constraints or changing the discretization of the dynamics can benefit many applications such as motion planning in dynamic environments. This work presents AdaptiveNLP, a software framework that efficiently constructs NLP functions based on pre-computed derivative information and provides functionalities to modify the NLP problem structure with low overhead. This adaptability enables the user to discard constraints known to be inactive which reduces computation times. In Model Predictive Control (MPC), it also allows tailoring a specific MPC iteration's NLP to the environment at that time instance. An MPC example and an adaptive gridding example show the effective reduction of total computation time and the ability to refine the time-grid of an NLP to produce a sparse but highly accurate solution with little overhead, respectively.**

I. Introduction

One approach to compute optimal trajectories while accounting for constraints for example in motion planning for autonomous vehicles or robotic manipulators is to solve an Optimal Control Problem (OCP). OCPs are typically hard to solve because the solution is continuous in time and thus infinite-dimensional. A popular approach to deal with the infinite-dimensionality is a direct method where the OCP is transcribed to a finite-dimensional nonlinear program (NLP). This transcription requires many choices to be made such as time-discretization, integration methods and the granularity of constraint enforcement on the time grid.

In a nonlinear Model Predictive Control (MPC) setting, a series of NLPs is solved over time rather than a single NLP. An NLP of identical structure is often solved repeatedly while only updating the initial state constraint. However, in motion planning applications, the environment around the vehicle can change, potentially requiring the type and number of constraints (and therefore the NLP structure) to change. Another case in which a series of similar NLPs is solved, is in adaptive gridding methods such as hp-methods [3, 6, 7, 9, 10, 16, 19]. These methods attempt to find a sparse time-grid on which the NLP-solution satisfies the system dynamics to some desired accuracy. This is achieved by first solving an NLP with a coarse time-grid which is then iteratively refined based on the obtained accuracy. Even though multiple NLPs are solved, this approach can still be faster than solving a single NLP with a fine time-grid because the multiple NLPs can be warm-started and gradually increase in problem size. However, there is a need to be able to change the NLP structure efficiently to fully exploit the advantage in computation time. The software GPOPS [1, 11] efficiently implements adaptive gridding methods. By considering the specific NLP structure and computing derivative information of collocation constraints beforehand, the NLP functions can be assembled easily. However, GPOPS does not support applying constraints only at specific segments of the trajectory. *In summary, there is a need to add and remove constraints to tailor the NLP to a specific environment based on the knowledge of the solution of the previous NLP, to change the time-grid on which the NLP is solved, and to change the constraints enforcing the dynamics. All of this should be possible in an efficient way to reduce the overhead such that it can be done online.*

When solving a general NLP with both equality and inequality constraints, often some inner subproblem is solved. Interior-point methods (IPMs) solve a linear system of equations representing the KKT-conditions that are relaxed with a barrier parameter. The relaxation is gradually reduced. A well-known open-source interior-point solver is Ipopt [18]. An IPM solver exploiting the specific problem structure arising from transcribing OCPs is FATROP [17]. In [18], the authors describe the robustness of the interior-point method even for infeasible initial guesses. On the other hand, there are Sequential Quadratic Programming (SQP) methods in which the inner subproblem is an easier optimization problem. This inner problem can be solved for example using an IPM or using an active set method that iteratively finds the set of active constraints and enforces these as equality constraints. An example of an active set solver is qpOASES [5]. Even though IPMs can be robust, they suffer from the presence of many inactive constraints since all constraints are always considered. This leads to unnecessary evaluations of the constraints but more importantly also of their contributions to the constraint Jacobian and Lagrangian Hessian. *Constraints that are known to be inactive should be removed from the NLP to avoid unnecessary computations and therefore reduce computation times of the solver.*

It is often unclear a priori which constraints will be inactive and can therefore be removed without affecting the solution of the NLP. For example, no-collision constraints are often only active at select segments of the trajectory but these segments only become clear when the solution is available. On the other hand, the number of constraints depends on the shape and number of obstacles which is unknown and might change. Some solvers implementing collision avoidance are CHOMP [12], CIAO [13, 14] and TrajOpt [15]. They deal with all obstacles at once using a single (signed) distance function. A signed distance field provides the distance to the closest obstacle at each point in the environment. When new obstacles appear, this distance field has to be recomputed. In [4], the authors propose an approach to combine different no-collision constraints into a single smooth constraint using a Logarithmic Sum-of-Exponentials (LSE) formulation. This approach introduced conservatism and is still computing the distance to all obstacles to find the maximum. State-of-art methods using CasADi [2] like OMG-tools [8] deal with uncertainty in the number of obstacles by defining as many obstacles as might be needed and placing them far away if less obstacles are present. These far away obstacles are referred to as dummy obstacles and allow to keep the NLP structure fixed if new obstacles appear. However, in all above approaches, even if no obstacles are around, the constraint is still enforced on all discrete points along the trajectory which will often lead to wasteful computations. In MPC, the knowledge of the solution in a previous MPC iteration can be exploited to remove constraints known to be inactive in the next iteration. This further motivates the need to easily and efficiently change the constraints present in the NLP without recomputing the parts of the Jacobian and Hessian that have not changed.

This paper presents a software framework called AdaptiveNLP that allows an easy and flexible transcription from OCPs to NLPs. The framework implements the interface of Ipopt to solve the resulting NLPs but it is not limited to a specific solver. The flexibility is threefold. Constraints can efficiently be added and removed, the time-grid can be updated and the constraints enforcing the dynamics can be changed. To efficiently change the NLP structure, derivative information of elementary contributions is computed offline using CasADi and used online to assemble the problem, similar to the GPOPS software. AdaptiveNLP's novelty lies in the fact that it enables more flexibility in enforcing system dynamics and also provides the ability to enforce constraints only at a limited number of discrete points to reduce computation times. The provided functionalities allow to solve a sequence of related NLPs faster by reusing as much as possible the structure of the previous NLP, but still making changes with significantly less overhead compared to using the optimization framework of CasADi. These also allow to exploit knowledge about the solution of previous NLPs by providing the NLP with a smaller set of inequality constraints, making it faster to solve, especially for interior-point solvers.

The AdaptiveNLP software framework is presented in Section II. Section III contains an MPC-example and Section IV shows an adaptive gridding example. Finally, conclusions are formulated in Section V.

The code to run the examples in this paper is provided on `https://github.com/meco-group/AdaptiveNLP`.

II. AdaptiveNLP framework

This section describes the AdaptiveNLP framework. The main idea is presented along with a general description of the problem that is solved using this software framework. Finally, a note on implementation aspects is provided.

A. Concept

The AdaptiveNLP framework constructs NLPs of the form

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & f(x) \\ \text{s.t.} \quad & L \leq g(x) \leq U, \end{aligned} \tag{1}$$

where L is a lower bound and U is an upper bound on the constraint values. The specific structure of $f(x)$ and $g(x)$ is explained in more detail in Section II-B. An NLP solver needs to be provided with the objective function $f(x)$, the objective gradient $\nabla f(x)$, the constraint vector $g(x)$, the constraint Jacobian $\mathcal{J} := \nabla_x^\top g(x)$ and the Lagrangian Hessian $\mathcal{H} := \nabla_{xx}^2 (f(x) + \lambda^\top g(x))$. One approach to compute derivative information is to construct $f(x)$, $g(x)$ and $f(x) + \lambda^\top g(x)$ symbolically and use algorithmic differentiation (AD) to compute its derivatives. This approach is used by the optimization framework of CasADi. It allows the user to formulate an NLP and takes care of all derivative computations in its back-end. Suppose now one constraint $h_L \leq h(x) \leq h_U$ is added to problem (1). This affects $g(x)$ and therefore also $\mathcal{J}$ and $\mathcal{H}$. The new Lagrangian Hessian is given by $\mathcal{H}' := \nabla_{xx}^2 \left(f(x) + \lambda^\top g(x) + \mu^\top h(x)\right)$. The addition of the new constraint requires a change to the symbolic expression of the constraint vector and the Lagrangian and requires to compute $\mathcal{J}$ and $\mathcal{H}$ again using AD. However, instead of computing these complete expression graphs again (which is what CasADi would do), the previously computed Hessian $\mathcal{H}$ could be reused by exploiting the linearity of the derivative operator resulting in $\mathcal{H}' = \mathcal{H} + \mu^\top \nabla_{xx}^2 h(x)$. By reusing the earlier Hessian (assuming $\nabla_{xx}^2 h(x)$ is already known), the number of symbolic operations, i.e., the operations needed to update the symbolic expressions, is greatly reduced. This in turn greatly reduces the overhead of changing the problem structure because these symbolic operations are typically expensive.

The AdaptiveNLP framework extends this approach by computing the first and second derivative of all elementary contributions to the objective and of all constraints offline. The Hessian is then assembled numerically as $\mathcal{H} = \nabla_{xx}^2 f(x) + \sum_i \lambda_i^\top \nabla_{xx}^2 g_i(x)$. The objective gradient and constraint Jacobian $\mathcal{J}$ are assembled similarly. This eliminates the need for any online changes to symbolic expressions and allows to efficiently change the NLP structure. The derivative information of the elementary contributions are computed using CasADi. In this framework, Ipopt is used to solve the NLPs but note that, in principle, other solvers could also be used.

B. Problem Formulation

We consider a general OCP of the form

$$\begin{aligned}\min_{x(t),u(t),T} \quad & \varphi_0(x(0),T,p_{\varphi_0}) + \int_0^T \varphi(x(t),u(t),T,p_\varphi)\mathrm{d}t \\ & + \varphi_T(x(T),T,p_{\varphi_T}) \end{aligned} \tag{P_0-a}$$

$$\text{s.t.} \quad \dot{x}(t) = f(x(t),u(t),p_d) \tag{P_0-b}$$

$$L_0 \ \leq \ g_0(x(0),T,p_0) \leq U_0 \tag{P_0-c}$$

$$L_T \ \leq \ g_T(x(T),p_T) \leq U_T \tag{P_0-d}$$

$$L_p \ \leq \ g_p(x(t),u(t),p_p) \leq U_p \quad \forall t \in [0,T). \tag{P_0-e}$$

which is referred to as problem P_0. Problem P_0 is solved for the functions $x^*(t)$ and $u^*(t)$ which represent the state and control trajectories respectively. In the objective function (P_0-a), φ_0 represents an initial cost. It takes as inputs the initial state and the total time T in the case of a free-time problem such that a cost to T can also be added if needed. For fixed-time problems, the input T is omitted everywhere. The symbol $p_{(.)}$ denotes a parameter vector. φ_T defines a terminal cost. A stage cost φ is defined over the whole time horizon. Note that in this formulation, there is no explicit time-dependency in the objective, but this can be included by augmenting the state vector. The system dynamics are enforced by adding constraint (P_0-b). The initial state $x(0)$ and terminal state $x(T)$ are subject to the constraints (P_0-c) and (P_0-d), respectively. The initial constraint is dependent on T such that constraints on T can be added here. All constraint functions $g_{(.)}$ output column vectors and the bounds $L_{(.)}$ and $U_{(.)}$ are independent of all optimization variables. Finally, path constraints can be enforced using constraint (P_0-e).

This continuous-time problem P_0 can be transcribed to an NLP which will be referred to as problem P_1 written by

$$\begin{aligned}\min_{\mathbf{x},\mathbf{u},T} \quad & \varphi_0(x_0,T,p_{\varphi_0}) + \sum_{k=0}^{N-1} \phi(x_k,u_k,\Delta t_k,T,p_\phi) \\ & + \varphi_T(x_N,T,p_{\varphi_T}) \end{aligned} \tag{P_1-a}$$

$$\text{s.t.} \quad g_d^{(n_k)}(x_k,u_k,...,x_{k+n_k-1},\Delta t_k^{(n_k)},T,p_d) = 0 \tag{P_1-b}$$

$$L_0 \ \leq \ g_0(x_0,T,p_0) \leq U_0 \tag{P_1-c}$$

$$L_T \ \leq \ g_T(x_N,p_T) \leq U_T \tag{P_1-d}$$

$$L_f \ \leq \ g_f(x_k,u_k,p_f) \leq U_f \quad k = 0...N-1 \tag{P_1-e}$$

$$L_e^{(i)} \leq (g_e)_j^i \, (x_k,u_k,p_{e,j,k}^{(i)}) \leq U_e^{(i)}. \tag{P_1-f}$$

Instead of solving for functions $x^*(t)$ and $u^*(t)$, P_1 is solved for finite-dimensional vectors $\mathbf{x}$ and $\mathbf{u}$, representing the state and controls at discrete points. The objective (P_1-a) contains a sum which is a discrete approximation of the integral of φ. The function ϕ is a discrete approximation of φ in (P_0-a). Because the discrete time values t_k are not constrained to be equidistant, the function ϕ also takes as input a value $\Delta t_k = t_{k+1} - t_k$. The system dynamics in P_0 (P_0-b) are replaced by the constraint (P_1-b). Note that these equality constraints can still be written in the form of (1) by setting U equal to L for these constraints. The superscript n_k indicates the number of time-steps that are involved in this constraint. For example, shooting methods such as a fourth-order Runge-Kutta, n_k is equal to two and in the case of direct collocation, n_k can attain any integer value. Different functions $g_d^{(n_k)}$ can be defined for different integrators. This means that over the horizon, different integrators or even different dynamics can be used. The initial and terminal constraints in P_1 are similar to the ones in P_0. For the state constraints (P_0-e), a distinction is made in problem P_1 between fixed constraints g_f and extra constraints g_e. The fixed constraints g_f are applied to all (x_k, u_k)-pairs except for the terminal state. The extra constraints g_e allow flexibility in the problem formulation since they can be added and removed and are applied only sparsely to points (x_k, u_k). The idea is to apply these constraints only when needed. The framework allows to define multiple extra constraints that each can be applied to a different set of discrete indices k. The superscript i in $(g_e)_j^i$ refers to a specific constraint. The subscript j denotes a specific instance of this extra constraint. It can be useful to enforce the same constraint multiple times to the same index k with different parameter values e.g., when multiple obstacles are present.

C. Implementation

The AdaptiveNLP framework is implemented in `C++`. A class `BuildingBlocks` is defined to collect all the constraints and objective contributions in the above formulation along with their contribution to $\mathcal{J}$ and $\mathcal{H}$. These building blocks are CasADi functions provided by the user that are then used to assemble a complete NLP. The derivative information can be easily computed using CasADi. An instance of this class is provided to the `AdaptiveNLP` class which in turn creates instances of the `Bookkeeper` class and of the `NLPInterface` class, for which details are given below. The `AdaptiveNLP` class manages all changes to the NLP such as adding or removing constraints, extending (or shortening) the horizon and/or changing the (discretization of the) dynamics. At construction time, the `AdapativeNLP` object is also provided with a maximal number of time steps $N_{\max}$ and a maximal number of extra constraint instances. With this information, memory is pre-allocated to avoid dynamic memory allocation and thus an increase in computation times.

The `Bookkeeper` class contains attributes to keep track of all bookkeeping needed to assemble the NLP such as sparsities of $\mathcal{J}$ and $\mathcal{H}$ and ordening of constraints. The sparsities are each represented with two vectors containing the row and column indices of the nonzero elements denoted respectively by $\mathbf{r}$ and $\mathbf{c}$. When the solver queries the numerical values of the nonzero elements, a single vector $\mathbf{v}$ must be provided containing the numerical values of the nonzero elements. More formally, the i-th nonzero element of the Hessian is located on row $\mathbf{r}[i]$ and column $\mathbf{c}[i]$ and has a value of $\mathbf{v}[i]$. Note that multiple constraints and objective terms can contribute to the value of the same nonzero element.

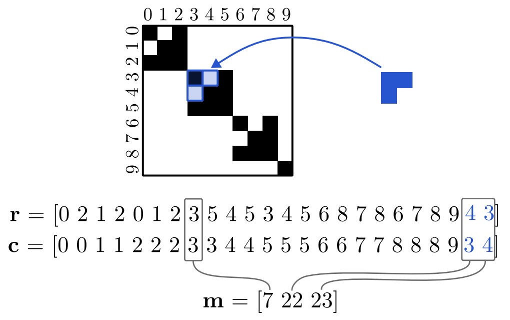

0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
r = [0 2 1 2 0 1 2 3 5 4 5 3 4 5 6 8 7 8 6 7 8 9 4 3]
c = [0 0 1 1 2 2 2 3 3 4 4 5 5 5 6 6 7 7 8 8 8 9 3 4]
m = [7 22 23]

Fig. 1: Illustration of the sparsity update when a new contribution (indicated in blue) is added.

Whenever a constraint $\bar{g}$ is added or the horizon is extended, new elementary contributions need to be added to $\mathcal{J}$ and $\mathcal{H}$. A new contribution to $\mathcal{J}$ is given by $\nabla^\top \bar{g}$ and to $\mathcal{H}$ is given by $\lambda^\top \nabla^2 \bar{g}$ for a constraint and $\nabla^2 \phi$ for an extension of the horizon. Fig. 1 illustrates how the `Bookkeeper` updates the matrix sparsities. First, the `Bookkeeper` queries the sparsity of the new contribution (illustrated in blue) and infers where these nonzero elements should be located in the complete matrix. The first nonzero has to be placed at $(3, 3)$ which overlaps with an already existing nonzero meaning the sparsity does not need to be updated. The other two nonzero elements are new nonzero elements, therefore, the `Bookkeeper` extends **r** and **c**. Other than updating the sparsity, the `Bookkeeper` constructs a vector **m**. This vector maps the nonzero elements of the contribution to indices in **r** and **c** and is used by the `NLPInterface` to correctly construct the numerical vector **v**. More precisely, for every contribution, the `NLPInterface` evaluates the nonzero elements and adds the value of the j-th nonzero element to **v**[**m**[j]]. In the example of Fig. 1, the `NLPInterface` will evaluate the new contribution and add the numerical values to elements **v**[7], **v**[22] and **v**[23]. This process allows the `NLPInterface` to efficiently construct **v** in the correct sequence that matches the sequence of nonzeros in **r** and **c**.

A sequence of variables is chosen to easily infer where new contributions need to be placed. This sequence of variables determines the sequence of columns in $\mathcal{J}$ and $\mathcal{H}$ and also determines the sequence of rows in $\mathcal{H}$. For free-time problems, the first variable is always the total time variable T. Next, the state variables x_k follow, alternated with the corresponding control inputs u_k. Note that there is no control value corresponding to the terminal state. It is important to state that x_k and x_{k+1} are not assumed to be chronologically consecutive time-steps. Instead, bookkeeping is done to keep track of the chronological sequence of indices k. This assumption is not made because it significantly reduces the amount of changes needed in the bookkeeping when the horizon length is changed and therefore allows to more efficiently change the problem structure. It does however destroy a banded sparsity structure but this can be resolved by permuting columns and rows. To illustrate this further, suppose a time-step is to be inserted in between

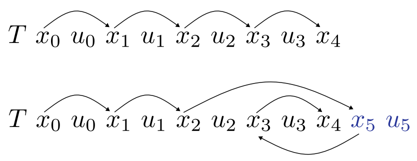

T x0 u0 x1 u1 x2 u2 x3 u3 x4
T x0 u0 x1 u1 x2 u2 x3 u3 x4 x5 u5

Fig. 2: Variable sequence before and after inserting a time-step in x_2 and x_3.

(x_2, u_2) and (x_3, u_3). For simplicity, consider the case where n_x and n_u are equal to 1. Fig. 2 shows the variable sequence for N equal to 4. The arrows indicate the chronological sequence which is kept track of by the `Bookkeeper`. If the new variables would be inserted in between (x_2, u_2) and (x_3, u_3), this would shift the columns corresponding to x_3, u_3 and x_4 to the right. Therefore, the nonzeros corresponding to a constraint or objective function contribution involving x_3, u_3 and x_4 have to be shifted which can require significant updates to the bookkeeping information. By not imposing a chronological sequence, the new variables can just be appended, leaving the existing columns and sparsity structure untouched. The `Bookkeeper` updates the links accordingly. The column index in $\mathcal{J}$ and $\mathcal{H}$ corresponding to the first element of the state vector x_k is given by $i = \delta_{\text{free-time}} + (n_x + n_u) \cdot k - n_u \cdot \delta_{\text{terminal}}(k)$ where n_x is the number of states, n_u is the number of controls and $\delta_{\text{free-time}}$ is equal to 1 for free-time problems and is equal to 0 otherwise. To compensate for the fact that the terminal state has no corresponding control value, $\delta_{\text{terminal}}(k)$ is equal to 1 if k is strictly larger than the index of the terminal state, otherwise it is equal to 0. In the example of Fig. 2, the column index corresponding to x_5 is indeed given by $1 + 2 \cdot 5 - 1 = 10$.

Whenever constraints or time-steps are removed from the problem, they still leave traces in the bookkeeping. For example the lower and upper bounds of the constraint vector might contain values for constraints that do not exist anymore or the sparsity of the Jacobian can be changed by replacing a constraint by a different one. To update all of the bookkeeping, a function `clearStructuralZeros()` is defined. This function evaluates the constraints vector, $\mathcal{J}$ and $\mathcal{H}$, identifies structural zeros and updates all bookkeeping. The structural zeros are identified by providing these NLP functions vectors filled with `NaN` values in which to write the results. Any `NaN` left after evaluation indicates a structural zero.

III. MPC Example problem

The added value of the AdaptiveNLP framework can be demonstrated in the following MPC example problem. The ability to add and remove constraints is exploited to reduce computation times and to deal with a varying environment of the vehicle.

A. Problem Description

Suppose a vehicle has to move through a warehouse environment. For some parts of the warehouse, free-space corridors (in which no obstacles are present) are known to the vehicle. In other parts of the warehouse, such information is not available and the vehicle has to detect obstacles and avoid them. The vehicle might also encounter people to which it can only get close if it slows down for safety reasons.

The vehicle itself is modeled using a bicycle model with a state vector $x = \begin{bmatrix} p_x & p_y & v & \theta \end{bmatrix}^\top \in \mathbb{R}^4$ (x and y position coordinates [m], forward velocity [m/s] and vehicle heading angle [rad]) and a control vector $u = \begin{bmatrix} a & \delta \end{bmatrix}^\top \in \mathbb{R}^2$ (acceleration [m/s^2] and steering angle [rad]). The dynamics are given by $\dot{x} = \begin{bmatrix} v\cos\theta & v\sin\theta & a & v\tan\delta \end{bmatrix}^\top$. The objective in this example is to move as far as possible along the x-axis while large control inputs are penalized. More formally, there is no cost to the initial state leading to $\varphi_0(x_0, p_{\varphi_0}) = 0$. The cost to the terminal state is given by $\varphi_T(x_N, p_{\varphi_T}) = -w_N p_{x,N}$ where $w_N = 1$ m^{-1} and $p_{k,N}$ is the x-coordinate of the terminal position such that the cost decreases as this x-coordinate increases. The stage cost is given by $\phi(x_k, u_k, \Delta t_k, p_\phi) = \Delta t_k \cdot (w_1 a_k^2 + w_2 \delta_k^2)$ where $w_1 = 1\text{s}^4\text{m}^{-2}$ and $w_1 = 1$ rad^{-2}. The initial state is assumed to be known and there is no constraint on the terminal state. The fixed constraint (P_1-e) is given by the inequalities

$$\begin{bmatrix} 0 \\ -\frac{\pi}{2} \\ -1 \\ -\frac{\pi}{3} \end{bmatrix} \leq \begin{bmatrix} v \\ \theta \\ a \\ \delta \end{bmatrix} \leq \begin{bmatrix} 2 \\ \frac{\pi}{2} \\ 1 \\ \frac{\pi}{3} \end{bmatrix}. \tag{4}$$

There are three extra constraints (P_1-f) in this problem corresponding to a corridor constraint, an obstacle-avoidance constraint and a safety constraint to slow down when approaching people. The corridor constraint takes four parameters ($x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$) and the constraint is defined as

$$0 \leq \begin{bmatrix} p_x - x_{\min} \\ x_{\max} - p_x \\ p_y - y_{\min} \\ y_{\max} - p_y \end{bmatrix} \leq \infty. \tag{5}$$

The no-collision constraint takes three parameters representing the coordinate of the center of the obstacle (x_{obs}, y_{obs}) and the radius of the obstacle (R_{obs}). This constraint is written as

$$0 \leq (p_x - x_{\text{obs}})^2 + (p_y - y_{\text{obs}})^2 - R_{\text{obs}}^2 \leq \infty. \tag{6}$$

Finally, the safety constraint imposes an additional velocity constraint. There is a safety radius R_{safety} defined around the position of every person (x_p, y_p) in which the velocity limit is enforced by adding the constraint

$$0 \leq h\left(\sqrt{(p_x - x_p)^2 + (p_y - y_p)^2}\right) - v \leq \infty, \tag{7}$$

where $h(r)$ is a third-degree polynomial that takes the distance to the person as input and satisfies $h(0) = 0$, $h(R_{\text{safety}}) = 2$ and $h'(R_{\text{safety}}) = h''(R_{\text{safety}}) = 0$. Note that $h(r) > 2\ \forall r > R_{\text{safety}}$ meaning this constraint only influences the velocity for points that satisfy $r < R_{\text{safety}}$. In this example $R_{\text{safety}} = 2.7$ m.

Fig. 3 shows the motion plan in the environment. There are two free-space corridors available for the vehicle with some people present in these corridors. Around every person, a safety zone with reduced velocity is shown. The second part of the world is an environment cluttered with obstacles. Additionally, the vehicle has a limited viewing radius of 10 m indicated with the white circle.

B. Strategy to Apply Extra Constraints

In between every MPC iteration, extra constraints can be added and removed. All points of the trajectory start of with a constraint enforcing them to be within the first corridor. Whenever a point gets close to the end of the corridor, the constraint parameter is updated to the parameter vector of the next corridor, if such a corridor exists. Otherwise, the corridor constraint is removed for that point.

If an obstacle appears in sight, the most recently obtained solution is used to determine at which discrete points the obstacle constraint has to be applied. These constraints are applied to any time-step k for which the corresponding coordinate $(p_{x,k}, p_{y,k})$ satisfies the condition

$$(p_{x,k} - x_{\text{obs}})^2 + (p_{y,k} - y_{\text{obs}})^2 \geq (R_{\text{obs}} + M)^2, \tag{8}$$

with M being a margin around the obstacle, in this case $M = 4$ m. Intuitively, the constraint is only enforced to points close to the obstacle. In Fig. 3, the obstacle-aware zone shows the space around the obstacles where the constraints are enforced. The same approach is used to deal with the safety constraints around people that are detected.

C. Discussion of Results

Three cases are considered and compared, each using a different approach to deal with the varying number of corridors, people and obstacles. Firstly, a single NLP can be modelled using CasADi Opti that contains all constraints that will be needed to account for the maximum number of obstacles, corridors and people. This is achieved by including dummy obstacles somewhere far away, dummy corridors that are very large or dummy people far away if less obstacles, corridors or people are seen similar to what OMG-tools does. In every iteration, an NLP of identical structure is solved (with potentially different parameter values). This case will be referred to as CasADi Opti 1. Secondly, a new NLP can be defined every iteration adding only the relevant constraints using the strategy explained in Section III-B. This case is referred to as CasADi Opti 2. Finally, the AdaptiveNLP framework is used to add constraints to and remove constraints from the problem without performing operations on symbolic expressions. The NLPs solved using the AdaptiveNLP framework are exactly the same as those in the second case using CasADi Opti.

Fig. 3 shows the travelled trajectories for the three compared cases. In each case, the same solution is found. Fig. 4 shows the number of constraints in the NLP over the different MPC iterations for the case CasADi Opti 2 and the AdaptiveNLP case. It shows how constraints come and go as the environment changes. In the case CasADi Opti 1, the maximal number of constraints is always present because it includes dummy obstacles, corridors and people.

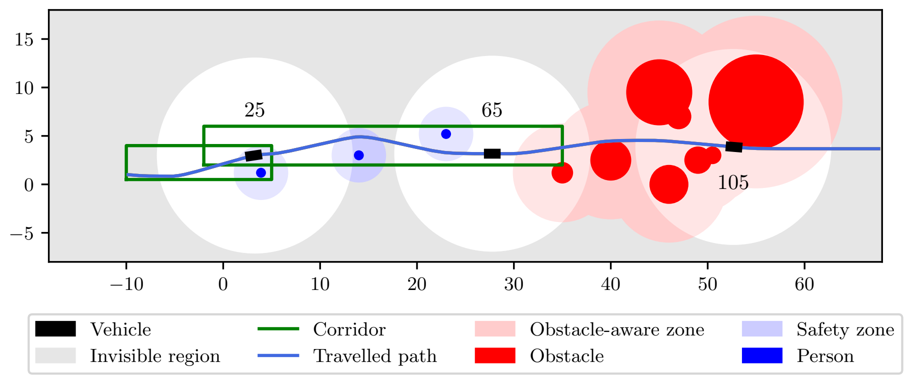


Fig. 3: Travelled trajectory through the world with corridors, people and obstacles with snapshots of the vehicle (with viewing radius) at different time instances.

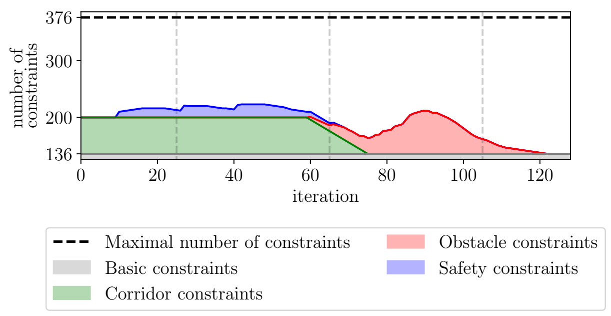


Fig. 4: Number of constraints throughout the different trajectories. The basic constraints contain the initial and terminal state constraints, the fixed constraints and the dynamics. The dotted lines correspond to the time instances shown in Fig. 3.

For every case, the time to solve the NLP is measured (t_{solve}) as well as the time needed to compute an initial guess for the next NLP, update parameter values and make any necessary changes to the NLP (t_{update}). A total of 130 MPC iterations are executed and this process is repeated a hundred times to compute the medians of these measured times, which are shown in Fig. 5. For the first case (CasADi Opti 1), in which every iteration solves the same problem with all constraints, the update time t_{update} is negligible compared to the solution time t_{solve}. This is expected since only inexpensive updates are needed such as updating parameters and shifting the solution by one time-step to provide an initial guess. In the second case (CasADi Opti 2), the time needed to solve the problem is significantly lower. This is due to the fact that there are fewer constraints to be evaluated while still resulting in the same solution. However, this improvement is completely undone by the long update time, arising from having to reconstruct a new NLP every iteration. In the AdaptiveNLP case, the solution time is similar to that of the second case, which is to be expected since they solve exactly the same NLPs. Now however, the updating time only includes changes to the bookkeeping which is

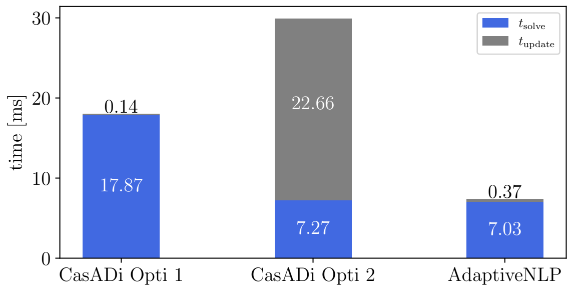


Fig. 5: Measured computation times for the three cases in the MPC example.

much cheaper than constructing a new problem. Because the AdaptiveNLP framework combines the low solution times of the second case with the low update times of the first case, it outperforms both strategies.

This reduced computation time allows choosing higher MPC frequencies or frees computational resources for other processes like camera image processing. Moreover, the AdaptiveNLP framework allows to consider free-space corridors and different types of obstacles all at once as they arise in the environment. There is no need to think about a maximal number of obstacles to consider because the NLP can efficiently adapt to the environment.

D. Scaling of Computation Times

Fig. 6 shows how the computation times and update times scale as N increases. Both t_{solve} and t_{update} (and as a consequence also the sum t_{total}) scale linearly with N. Considering the solution time t_{solve}, it is clear that the first case (CasADi Opti 1) scales worse. This is caused by the increasing number of irrelevant constraints as N increases. The updating times of the first case is close to constant. For the AdaptiveNLP case, the updating time grows linearly because more constraints are added and removed as N increases. However, the updating time of the second case (CasADi Opti 2) rises significantly. This is explained by the fact that the overhead of constructing a new problem every

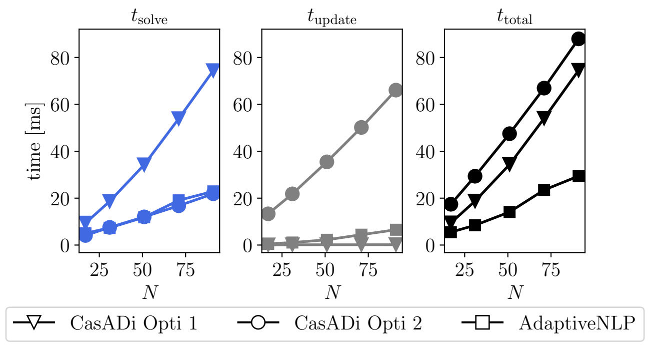


Fig. 6: Scaling of computation times as N increases.

iteration grows with the size of the problem, and thus with N.

It can be concluded that the total times in the AdaptiveNLP scale better as N increases because the AdaptiveNLP is able to reduce the amount of constraints in the problem but does this at low cost (even for large N). The benefit of the flexibility in applied constraints becomes even larger for larger problems.

IV. Adaptive Gridding Example Problem

The example problem discussed in this section implements an adaptive gridding approach for a simple moonlander case. It is a suitable problem to showcase an adaptive gridding method because it is known that the solution is a bang-bang solution which can only be represented on a suitable time-grid.

A. Problem Description

The goal in this example is to land a moonlander as fast as possible on the surface of the moon without crashing into it. The lander is subject to a lunar gravitational pull and has one thruster to be used to slow down. Formally, the OCP has two states representing the height (h) [m] and vertical velocity (v) [m/s] of the lander that adhere to the dynamics given by $f(x,u) = \begin{bmatrix} v & -1.62\text{m/s}^2 + u \end{bmatrix}^\top$ where u is the control input representing the thrust (m/s^2). The control is bounded by the inequalities $0 \leq u \leq 8$. The initial state is $x_0 = \begin{bmatrix} 1 & 0 \end{bmatrix}^\top$ and the terminal state is $x_f = \begin{bmatrix} 0 & 0 \end{bmatrix}^\top$. There are no extra constraints in this problem.

The AdaptiveNLP framework is used to solve this problem by providing it a number of different collocation constraints. In this specific example, collocation constraints to enforce dynamics $g_d^{(4)}$, ..., $g_d^{(8)}$ are provided, meaning 3 up to 7 collocation points can be used. The function `changeIntervalDiscretization()` provided by the `AdaptiveNLP`-class is used to change the discretization. The gridding strategy follows the strategy presented by Patterson et al [10] with a tolerance of 10^{-6}.

B. Discussion of Results

Fig. 7 shows the resulting control inputs. Starting with 3 intervals of 3 collocation points each, 10 refinements in the

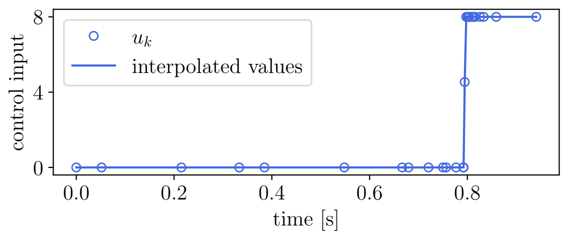


Fig. 7: Control input for moonlander problem.

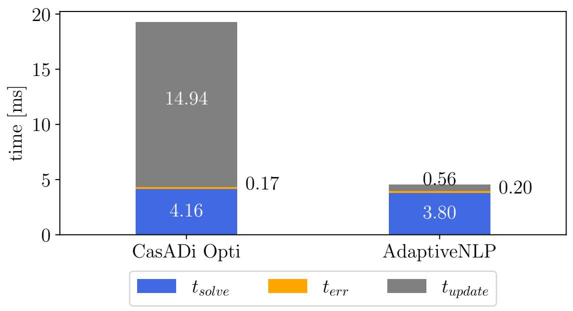
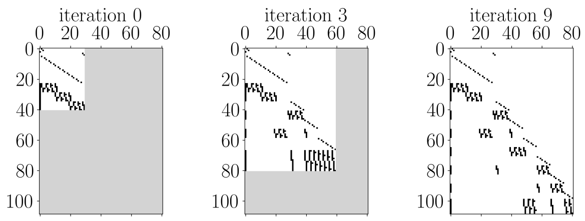


Fig. 8: Computation times for the moonlander example.

time-grid have been made to achieve the desired accuracy in the dynamics. As can be seen, the resulting time-grid is more fine around 0.8 s to capture the jump in the control value.

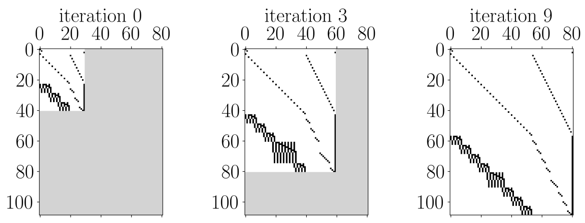
To benchmark computation times of the AdaptiveNLP framework CasADi Opti is used again. A new Opti instance has to be constructed every iteration because the problem structure changes. The time to solve the NLPs, the time to compute the error estimates and the time to make changes to the NLP are all measured. The median of 100 runs is computed and shown in Fig. 8. Because both approaches solve the same sequence of NLPs, the time to solve the NLPs is similar (4.16 ms for the CasADi Opti case and 3.80 ms for the AdaptiveNLP case). The time needed to compute the error estimates is 0.17 ms for the CasADi Opti case and 0.20ms for the AdaptiveNLP case. The main difference lies in the update time. For the CasADi Opti case, constructing new NLPs takes 14.94 ms. This time includes constructing the objective and constraints symbolically and using AD to compute $\mathcal{J}$ and $\mathcal{H}$. In the AdapativeNLP case, only 0.56 ms is needed to update the bookkeeping information, almost completely removing the overhead of refining the time-grid and creating a new NLP.

Fig. 9 shows Jacobian sparsities for a few iterations. Note that because of the imposed variable sequence as explained in Section II-C, the Jacobian computed in an earlier iteration is reused and extended which is visible in the limited amount of changes from one iteration to another. The sparsities of the Jacobians using Opti are shown in Fig. 10. It has a different sparsity pattern because of a different variable sequence. It is clear that inserting time steps in this



(a) Iteration 1 (b) Iteration 4 (c) Iteration 10

Fig. 9: Jacobian sparsities using AdaptiveNLP.



(a) Iteration 1 (b) Iteration 4 (c) Iteration 10

Fig. 10: Jacobian sparsities using Opti.

variable sequence requires shifting many nonzero elements which is not necessary in the variable sequence used in the AdaptiveNLP.

V. Conclusions and Future Work

A software framework is presented showing how the ability to adapt the transcription from OCP to NLP can be exploited to reduce computation times. More precisely, because of the low overhead in adding and removing constraints, constraints that are known to be inactive can easily be removed from the problem. This is especially relevant for interior-point solvers that can suffer from evaluating irrelevant constraints. As shown, a significant reduction in computation is observed, especially for large problems. Other than reducing computation times, the design of the NLP is made easier because there is no longer a need to construct a general NLP that can be used in all scenarios.

Other than constraints, the time-grid can also be refined (or extended) with little overhead as demonstrated in the adaptive gridding example.

There exist many downstream applications that will benefit from AdaptiveNLP in addition to adaptive time-gridding and collision-free motion planning, e.g., using different vehicle dynamics in different parts of the trajectory or changing the horizon length online to match an MPC update rate. Multi-agent applications where the number of agents is unknown to each agent might also offer an opportunity to exploit flexibility in the number of constraints.

Acknowledgement This work has been carried out within the framework of Flanders Make SBO project ARENA (Agile and Reliable Navigation). Flanders Make is the Flemish strategic research centre for the manufacturing industry.

We thank Dr. Ajay Sathya for his guidance in shaping the initial stages of this research and dr. Alejandro Astudillo for his availability for questions during the C++ implementation.

References

[1] Yunus M. Agamawi and Anil V. Rao. "CGPOPS: A C++ Software for Solving Multiple-Phase Optimal Control Problems Using Adaptive Gaussian Quadrature Collocation and Sparse Nonlinear Programming". In: *ACM Trans. Math. Softw.* 46.3 (July 2020). ISSN: 0098-3500.

[2] Joel A E Andersson et al. "CasADi – A software framework for nonlinear optimization and optimal control". In: *Mathematical Programming Computation* 11.1 (2019), pp. 1–36.

[3] Christopher Darby, William Hager, and Anil Rao. "An hp-adaptive pseudospectral method for solving optimal control problems". In: *Optimal Control Applications and Methods* 32 (July 2011), pp. 476–502.

[4] Dries Dirckx et al. "A Smooth Reformulation of Collision Avoidance Constraints in Trajectory Planning". In: *2022 IEEE 17th International Conference on Advanced Motion Control (AMC)*. 2022, pp. 132–137.

[5] H.J. Ferreau, H.G. Bock, and M. Diehl. "An online active set strategy to overcome the limitations of explicit MPC". In: *International Journal of Robust and Nonlinear Control* 18.8 (2008), pp. 816–830.

[6] Fengjin Liu, William W. Hager, and Anil V. Rao. "Adaptive Mesh Refinement Method for Optimal Control Using Decay Rates of Legendre Polynomial Coefficients". In: *IEEE Transactions on Control Systems Technology* 26.4 (July 2018), pp. 1475–1483. ISSN: 1558-0865.

[7] Fengjin Liu, William W. Hager, and Anil V. Rao. "Adaptive mesh refinement method for optimal control using nonsmoothness detection and mesh size reduction". In: *Journal of the Franklin Institute* 352.10 (2015), pp. 4081–4106. ISSN: 0016-0032.

[8] Tim Mercy, Ruben Van Parys, and Goele Pipeleers. *Spline-Based Motion Planning for Autonomous Guided Vehicles in a Dynamic Environment*. eng. 2017-08-30.

[9] Alexander T. Miller, William W. Hager, and Anil V. Rao. "Mesh refinement method for solving optimal control problems with nonsmooth solutions using jump function approximations". In: *Optimal Control Applications and Methods* 42.4 (2021), pp. 1119–1140.

[10] Michael A. Patterson, William W. Hager, and Anil V. Rao. "A ph mesh refinement method for optimal control". In: *Optimal Control Applications and Methods* 36.4 (2015), pp. 398–421.

[11] Michael A. Patterson and Anil V. Rao. "GPOPS-II: A MATLAB Software for Solving Multiple-Phase Optimal Control Problems Using Hp-Adaptive Gaussian Quadrature Collocation Methods and Sparse Nonlinear Programming". In: *ACM Trans. Math. Softw.* 41.1 (Oct. 2014). ISSN: 0098-3500.

[12] Nathan Ratliff et al. "CHOMP: Gradient optimization techniques for efficient motion planning". In: *2009 IEEE International Conference on Robotics and Automation*. 2009, pp. 489–494.

[13] Tobias Schoels et al. "An NMPC Approach using Convex Inner Approximations for Online Motion Planning with Guaranteed Collision Avoidance". In: May 2020, pp. 3574–3580.

[14] Tobias Schoels et al. "CIAO∗: MPC-based Safe Motion Planning in Predictable Dynamic Environments". In: *IFAC-PapersOnLine* 53.2 (2020). 21st IFAC World Congress, pp. 6555–6562. ISSN: 2405-8963.

[15] John Schulman et al. "Finding locally optimal, collision-free trajectories with sequential convex optimization." In: *Robotics: science and systems*. Vol. 9. 1. Berlin, Germany. 2013, pp. 1–10.

[16] Maedeh Souzban, Omid Solaymani Fard, and Akbar H Borzabadi. "A rapid-based improvement on some mesh refinement strategies in solving optimal control problems". In: *IMA Journal of Mathematical Control and Information* 37.1 (Mar. 2020), pp. 351–376. ISSN: 1471-6887.

[17] Lander Vanroye et al. *FATROP: A Fast Constrained Optimal Control Problem Solver for Robot Trajectory Optimization and Control*. 2023-06-21.

[18] Andreas Wächter and Lorenz T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Mathematical Programming* 106.1 (Mar. 2006), pp. 25–57. ISSN: 1436-4646. DOI: 10.1007/s10107-004-0559-y.

[19] Long Xiao, Xinggao Liu, and Shiming He. "An Adaptive Pseudospectral Method for Constrained Dynamic Optimization Problems in Chemical Engineering". In: *Chemical Engineering & Technology* 39 (June 2016).