An Object Oriented Framework *for a* Generic Model Predictive Controller

Timothy A. V. Teatro, *Student Member, IEEE* and J. Mikael Eklund, *Senior Member, IEEE*

***Abstract*—We present an object oriented design for a reusable nonlinear model predictive controller framework. The framework balances reusability with flexibility and performance. Separation of interface from implementation simplifies development and facilitates use of dynamic polymorphism to change the controller behaviour at runtime. The work is presented in language agnostic terms, but the success of a particular implementation for an omni-wheeled robot is briefly described. That implementation is available online, but an application non-specific version will be made available with examples as starting templates.**

***Index Terms*—Predictive control, Mobile robot motion-planning**

I. Introduction

HAVING WRITTEN software for Nonlinear Model Predictive Control (NMPC) several times, it became clear to us that core elements of the controller code could be separated and reused. Such reuse would lead to fewer bugs, a shorter development cycle and less fragile code all of which fosters innovation.

The familiar task of cloning a previously written code base and undertaking a massive re-write is a difficult and precarious endeavour. During some large portion of the rewrite, the intermediate states of your code are unlikely to compile or to give any meaningful behaviour to test if it did. In the mean time, you are potentially writing bugs with no feedback to inform you of them. By the time your code is in a state where you may consider debugging, your output will likely be the emergent product of several interacting bugs.

We have embarked on an endeavour to author a reusable framework for a practical, high-performance, multi-level controller complete with a command-line server for runtime interaction. Our primary goal is a framework that is straightforward to reimplement on new systems while being safe (and thread-safe) and testable. It should be suited to incremental development techniques such as Kent Beck's Test Driven Development [1] and therefore should be modular. Polymorphism should be leveraged so that the high-level supervisory programs can modify the controller behaviour at runtime by *hot-swapping* modules.

Currently in alpha-testing, our code base is applied to a semi-autonomous robot called virtualME by CrossWing Inc. [2] for navigation. This is a continuation of our work presented in [3] and [4].

Funding provided by the NSERC's Engage program.

T. A. V. Teatro (email: timothy.teatro@uoit.ca) and J. M. Eklund (email: mikael.eklund@uoit.ca) are from the Department of Electrical, Computer and Software Engineering, University of Ontario Institute of Technology, Oshawa, ON L1H 7K4

At the next stage of development the core framework will be isolated, packaged with a few examples as starting templates, and published under an open-source license.

The framework should work well for any type of iterative control process, and maybe more. However, we have started with NMPC control and so this paper describes the architecture as it relates to that specific controller.

Software packages exist for modelling and optimizing which are commonly used for constructing NMPC and related controllers. (For examples, see YALMIP or CVX Matlab toolboxes). The authors feel that the contribution of this work is not in competition with those offerings. Rather, this architecture will be most useful to people who need the flexibility that comes with hand tailored computational methods and tight integration with other components of a running platform or embedded system. We leave it to the user to provide code for modelling an minimization so that performance is in their hands.

II. (Nonlinear) Model Predictive Control

Model Predictive Control (MPC) was originally developed for process control [5], [6]. The method involves using a mathematical model to predict the consequences of a control plan over a finite time interval called the *preview horizon*. That is, for a given control plan $\boldsymbol{u}$, the model will predict a state trajectory $\boldsymbol{x}$. In discrete terms, the horizon is the interval of integers $[1, N]$. The model is a recurrence relation of the form $\boldsymbol{x}_k = \boldsymbol{f}(\boldsymbol{x}_{k-1}, \boldsymbol{u}_{k-1})$ for $k \in [1, N]$. That relation along with techniques from mathematical optimization are used to find a control plan $\boldsymbol{u}^*$ that minimizes a cost functional J which depends on the control input and predicted state trajectory:

$$\boldsymbol{u}^* = \underset{\{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots \boldsymbol{u}_N\}}{\arg\min} J(\boldsymbol{x}, \boldsymbol{u}), \tag{1}$$

The optimization is performed online. The individual iterations are open-loop: no state data is used except for the initial condition $\boldsymbol{x}_0$. The loop is closed when information from the plant's state estimator is used to seed the forecasting model in each iteration. This relationship is illustrated in Figure 1.

The overall state trajectory is formed by executing the first element(s) of each of the optimal plans before the calculation is repeated using up-to-date state estimates. This process is summarized in Algorithm 1.

The algorithm can be applied to nonlinear models directly, and is then referred to as Nonlinear Model Predictive Control (NMPC) [7], [8], [5]. In most applications of this technique to nonlinear systems (for example, [9], [3], [10], [11], [12], [13], [14]), the minimization problem 1 is approached by solving the Euler-Lagrange differential equations using a variational

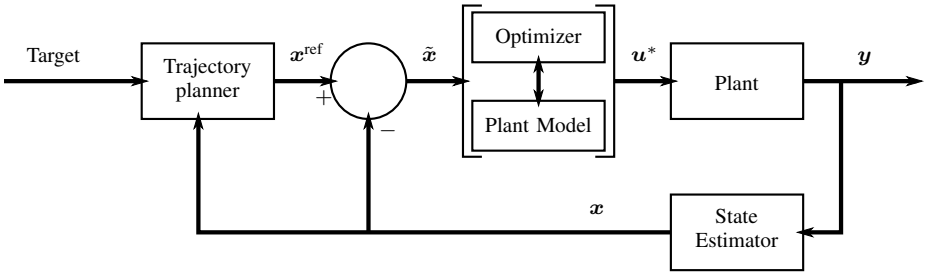

Fig. 1. A block diagram for a system with (N/)MPC controller. The trajectory planner sets a reference path x^{ref} based on a given target. The error signal $\tilde{x}$ is used in the control block which orchestrates the optimization calculation to get the optimal control plan u^*. The state is estimated and fed back to the planner, closing the loop.

Algorithm 1 A general statement of the Model Predictive Control algorithm

1: $u \leftarrow$ initialized value
2: **while** not sufficiently close to target **do**
3: $x_0 \leftarrow$ current state estimate
4: $x^{\text{ref}} \leftarrow$ desired path from path planner
5: $u \leftarrow \arg\min_{\{u_1, u_2, \ldots u_N\}} J(x, u)$
6: Execute u_1^*
7: **end while**

expansion to construct a gradient based iterative search. The scheme usually involves application of Lagrange multipliers to constrain solutions to those obeying the system dynamics.

III. The Architecture

We have separated the problem into six key objects:

1) NMPC Kernel
2) Model
3) Minimizer
4) Path planner
5) Logger
6) Executor

Though more detailed descriptions of these objects are to be found in the sub-headings of this section, for now it suffices to say that these objects assume the responsibilities that their names suggest. The problem space is naturally parted along these lines which leads to computer code that is intuitive and straightforward to develop and maintain. Each module may be individually developed and unit tested.

In ideal use, initializing the controller is a matter of instantiating a series of mutually compatible objects and entering the NMPC main loop as per Algorithm 1:

```
model = instance of sub-type of NmpcModel
minimizer = instance of sub-type of NmpcMinimizer
planner = instance of sub-type of PathPlanner
logger = instance of sub-type of Logger
kernel = instance of NmpcKernel
do                          ▷ // The NMPC loop as in Algorithm 1
   kernel.nmpcStep(planner.getSeed())
   executor.run(...)
while planner.isContinuing()
```

where the *seed* contains the current state estimate of the robot and the reference trajectory for tracking.

To the largest practical extent, these objects should interact through narrow interfaces. This decreases coupling and leads to less fragile code. Developing child classes to implement those interfaces is a task which is suited to incremental development techniques and unit testing. Writing code to implement an interface also helps the programmer by documenting the behaviour each class must implement. Each interface should outline clear responsibilities for its sub-classes and be independent where possible.

When implementation is separated from interface, dynamic polymorphism allows the user to *hot-swap* modules to modify behaviour of the algorithm. For example, if a particular child of `NmpcMinimizer` is having difficulty, or is too slow given specific conditions then it can be swapped for one more appropriate to the task.

A. NmpcModel

The class `NmpcModel` provides an interface for the `NmpcKernel` to operate. In each new application of the framework, the programmer should inherit from `NmpcModel` at which point they are free to implement the interface however they wish.

The interface is illustrated in Figure 2. A child of `NmpcModel`

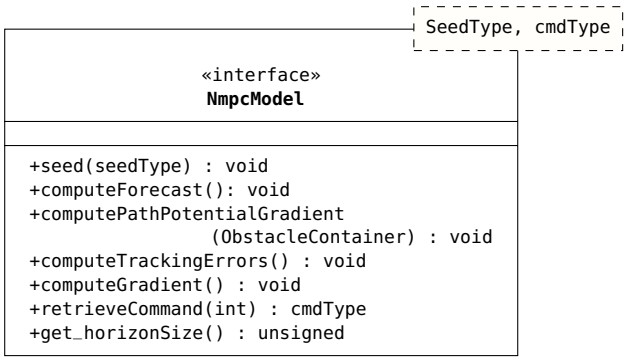

Fig. 2. Class diagram for `NmpcModel` interface.

contains the predicted state trajectory, tracking error, control plan, and gradient information for the horizon. Its methods provide the means by which the gradient vector for the optimizer is produced from the seed with all intermediate steps including the model based forecast/prediction. When all of the data in `NmpcModel` correspond to the seed information, we call the model *self-consistent*. For example, when the model is given

a new seed, but the gradient and forecast data still correspond to the previous seed, then the model is *self-inconsistent*. When the appropriate methods are called so that the forecast and gradient are properly computed for the new seed, then the model is then again in a self-consistent state.

It is intended as a promise for the kernel *and* the minimizer. The kernel will make use of `seed(·)` and `retrieveCommand(·)` and perhaps the getter for the horizon size. The remaining methods are promised to the minimizer.

The `computeForecast()` method triggers the prediction step where the seeded state is propagated forward in time by the system dynamics to form a state-space trajectory.

The `computePotentialGradient(·)` computes $\partial\Phi/\partial\boldsymbol{x}$, which is necessary for gradient based optimization techniques. The potential gradient is to be distinguished from the business of `computeGradient()` which computes $\partial J/\partial\boldsymbol{u}$ for the gradient based minimum search.

The `retrieveCommand()` method takes an integer, say n and returns the n^{th} command in the current horizon. The `commandType` is just an abstraction for the data structure required for the executor do its job.

Since the methods are largely niladic and return void, it follows that a concrete `NmpcModel` child will have the relevant data in its membership. The minimizer (and logger) will need access to those members, which are not part of the interface. This means that the logger and minimizer must be programmed specifically toward a particular child of `NmpcModel`. This is not generally a problem. Most `NmpcModel`s programmed for a given system (and have the same state and control spaces), even if they contain different implementations will share the same data. It is not difficult in most cases to have compatibility between loggers, minimizers and multiple models for a given system.

One might notice that the `NmpcModel` can be pared down by removing the methods `computeForecast()` and `computePotentialGradient()` from the interface, making them private and simply calling them in `computeGradient()`. This is correct and doing so would have the added benefit of reducing the possibility of finding `NmpcModel` in a self-inconsistent state. However, for now, they remain as part of our interface to facilitate unit testing and for the clarity of self-documentation that they bring.

B. NmpcMinimizer

A child of `NmpcMinimizer` is compatible with a particular child of `NmpcModel`. It implements the interface for the kernel to invoke the minimization. The minimization is typically an iterative process where the model is repeatedly given a candidate control plan and brought to self-consistency until the ℓ^2-norm of the gradient is sufficiently small.

The minimizer interface is lean and is illustrated in Figure 3.

The kernel will call `solveOptimalControlHorizon` at appropriate times and the minimizer must operate leaving the model in a self-consistent state such that the control plan is optimal in the sense of Eq. (1). The return value is a code, ideally from an `enum` should notify the kernel of events such as hitting a maximum iteration limit or failed bounds checking.

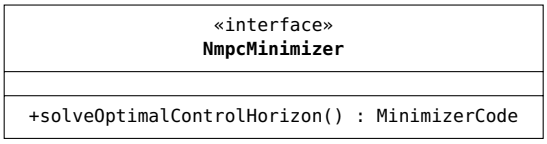

Fig. 3. Class diagram for `NmpcMinimizer` interface.

Since our implementation is in C++, we chose to use a return code as a lighter alternative to exceptions.

C. NmpcKernel

The `NmpcKernel` wraps most of the steps in the NMPC iteration. It passes the seed to the model and triggers the minimizer. It provides a high-level interface that receives a seed for each NMPC step and dispenses individual commands from the solved horizon and keeps account of them, preventing the user from collecting more than a horizon's worth of commands.

The kernel is a concrete class since it has not yet been useful to promise its interface. It aggregates the model, the minimizer, the planner and the logger as shown in Figure 4.

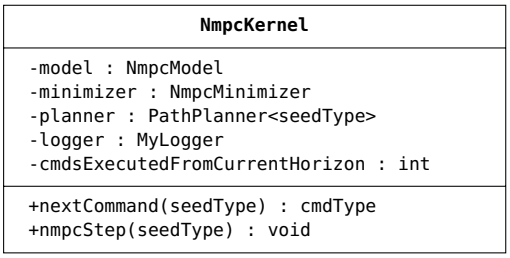

Fig. 4. Class diagram for `NmpcKernel` class.

Through the `nmpcStep(·)` method, it executes the logic for appropriately logging and handling the minimizer. The `nextCommand()` function dispenses commands for the use of the executor.

We have found it useful to make the `NmpcKernel` observable, in the sense of the GOF observer pattern [15]. This way the executor and any higher level logic can be made automatically aware of the presence of new solutions at the end of the `nmpcStep`.

D. Path Planner

The `PathPlanner` sits a level of abstraction higher than the `NmpcKernel`. It keeps track of the current state estimate and provides the seed for the kernel to pass down to the `NmpcModel`. The `PathPlanner` may halt the progress of the `NmpcKernel` by returning `false` from the method `isContinuing()`.

The `PathPlanner` provides the slender interface shown in Figure 5.

The `getSeed()` and `isContinuing()` methods are mainly used in the NMPC Loop as demonstrated in the first part of §III. The getter for the obstacles is used to provide the obstacles for the `computePathPotentialGradient(·)` member of `NmpcModel`.

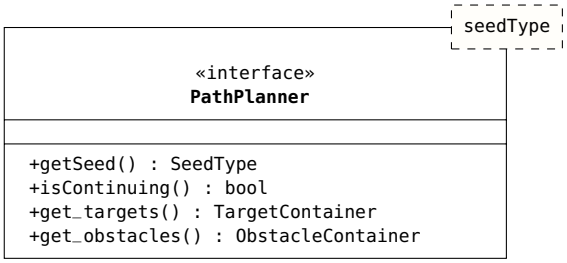

Fig. 5. Class diagram for `PathPlanner` class.

E. The Logger

The `Logger` is an optional but useful part of this apparatus. Its methods are called by the kernel at key points of the NMPC step so that information may be gathered and exported. For example, we have found it useful to write output to a simple JavaScript Object Notation (JSON) formatted text file at each step which is easily parsed for post-processing and analysis. In future work we plan to re-implement the interface to write high performance HDF5 data files which will allow us to efficiently store more data for more intensive studies.

F. The Executor

The Executor is in the abstraction layer which interfaces the product of the NMPC calculation with the level below the controller. This may be the actuators of the physical plant or a lower level of controller such as a PID controlling a servomotor. Since this is entirely application dependent, it is a topic not suited for a discussion of a generic framework. However, it is a vital component of each deployment of the framework.

IV. Success With The Current Code Base

Currently in alpha-testing, our code base is implemented in *C++* conforming to ISO/IEC 14882:2014 [16] and is applied to a semi-autonomous omni-wheeled robot called virtualME by CrossWing Inc. [2] as an NMPC controller with a high-level software supervisor.

The code includes a threaded TCP/IP command-line server to modify the list of targets and list of obstacles during run-time. The architecture allows for more flexibility than just those tasks, but it is sufficient for this application at this time.

The current size of the code base is roughly 4300 lines of code *not* including blank lines and comments. This does, however, include unit tests, integration tests, mock objects/stubs, analysis tools, visualization tools and some tools that are quite specific to the robotic application.

The code in its current application is available at http://www.github.com/timtro/vme-nmpc.

Early results are extremely promising. The framework offers the flexibility to support several simultaneous branches of research and development. Performance is not significantly different from our previous C99 implementations of the same controller, while being safer, more flexible and easier to develop. It is difficult, in general, to comparatively discuss performance since the flexibility of the framework requires users to provide their own code for the computationally intensive portions. We would mainly concern ourselves with overhead introduced by the architectural apparatus, which appears to be minimal.

V. Continuing goals

The project started with an NMPC controller built around a specific robot. It remains for us to isolate the core framework and make it stand alone.

It seems that without sacrificing anything, the remaining framework will suit linear-MPC and other iterative controllers.

In its final stages, we would like to polish and release the code under an open-source license along with documentation and examples with patterns for thread-safe use.

References

[1] K. Beck, *Test-driven Development: By Example*. Addison-Wesley Professional, 2003.

[2] (2011) Crosswing virtualME website. [Online]. Available: http://www.crosswing.com/

[3] T. A. V. Teatro, J. M. Eklund, and R. Milman, "Nonlinear Model Predictive Control for Omnidirectional Robot Motion Planning and Tracking With Avoidance of Moving Obstacles," *Can. J. Electr. Comput. Eng.*, vol. 37, no. 3, pp. 151–156, 2014.

[4] T. A. V. Teatro and J. M. Eklund, "Nonlinear model predictive control for omnidirectional robot motion planning and tracking," in *2013 26th IEEE Can. Conf. Electr. Comput. Eng.* IEEE, may 2013, pp. 1–4.

[5] S. Qin and T. A. Badgwell, "A survey of industrial model predictive control technology," *Control Eng. Pract.*, vol. 11, no. 7, pp. 733–764, jul 2003.

[6] J. Richalet, A. Rault, J. Testud, and J. Papon, "Model predictive heuristic control," *Automatica*, vol. 14, no. 5, pp. 413–428, sep 1978.

[7] R. Findeisen, L. Imsland, F. Allgower, and B. A. Foss, "State and Output Feedback Nonlinear Model Predictive Control: An Overview," *Eur. J. Control*, vol. 9, no. 2-3, pp. 190–206, jan 2003.

[8] D. Mayne, J. Rawlings, C. Rao, and P. Scokaert, "Constrained model predictive control: Stability and optimality," *Automatica*, vol. 36, no. 6, pp. 789–814, 2000.

[9] T. Teatro, P. McNelles, and J. M. Eklund, "A Formulation of Rod Based Nonlinear Model Predictive Control of Nuclear Reaction with Temperature Effects and Xenon Poisoning," in *23rd Int. Conf. Nucl. Eng.* Chiba, Japan: JSME, 2015.

[10] J. M. Eklund, J. Sprinkle, and S. Sastry, "Symmetric and Asymmetric Pursuit Evasion Games With Model Predictive Control." *IEEE Trans. Control Syst. Technol.*, vol. In Press, 2012.

[11] M. A. Abbas, R. Milman, and J. M. Eklund, "Obstacle avoidance in real time with Nonlinear Model Predictive Control of autonomous vehicles," in *Electr. Comput. Eng. (CCECE), 2014 IEEE 27th Can. Conf.*, may 2014, pp. 1–6.

[12] S. Petryna, J. M. Eklund, and G. Rohrauer, "Modelling a thermoelectric heat exchanger," in *Electr. Comput. Eng. (CCECE), 2013 26th Annu. IEEE Can. Conf.*, may 2013, pp. 1–6.

[13] F. Fahimi, "Non-linear model predictive formation control for groups of autonomous surface vessels," *Int. J. Control*, vol. 80, no. 8, pp. 1248–1259, aug 2007.

[14] G. J. Sutton and R. R. Bitmead, "Performance and Computational Implementation of Nonlinear Model Predictive Control on a Submarine," in *Nonlinear Model Predict. Control*, ser. Progress in Systems and Control Theory, F. Allgöwer, A. Zheng, and C. I. Byrnes, Eds. Birkhäuser Basel, 2000, vol. 26, pp. 461–472.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[16] "ISO/IEC 14882:2011 Information technology – Programming languages – C++," International Organization for Standardization, Geneva, Switzerland, ISO 14882:2014, 2014.