**rsLQR Algorithm Description**

It is essential to have a clear understanding of underlying memory allocation and algorithm’s data flow in order to re-implement the algorithm on GPU and find the possible ways to adjust the algorithm to the GPU features and take advantage of even more potential parallelism. In the following section we provide a graphic visualization of the different structs used in rsLQR file and the logic behind it. We proceed with a more detailed description of the algorithm , solve.c file in particular, and visualization of the data flow.

1. **Structs in rsLQR**

* BinaryNode (from binary\_tree.h)
* OrderedBinaryTree (from binary\_tree.h)
* NdLqrCholeskyFactors (from cholesky\_factors.h)
* CholeskyInfo (from linalg.h)
* LQRData (from lqr\_data.h)
* LqrProblem (from lqr\_problem.h)
* Matrix (from matrix.h) // do not need this class in Python as it is supported by numpy library no need
* NdFactor (from nddata.h)
* NdData (from nddata.h)
* NdLqrProfile noneed
* NdLqrSolver (from solver.h)

In our implementation in Python we omitted CholeskyInfo and Matrix structs (or in Python language ‘classes’) as they could be replaced with linalg library. Hence we will not provide graphical visualization of these structs.

All structs can be divided into two groups: structs that are inside the NdLqrSolver object or structs that build up the LqrProblem object.

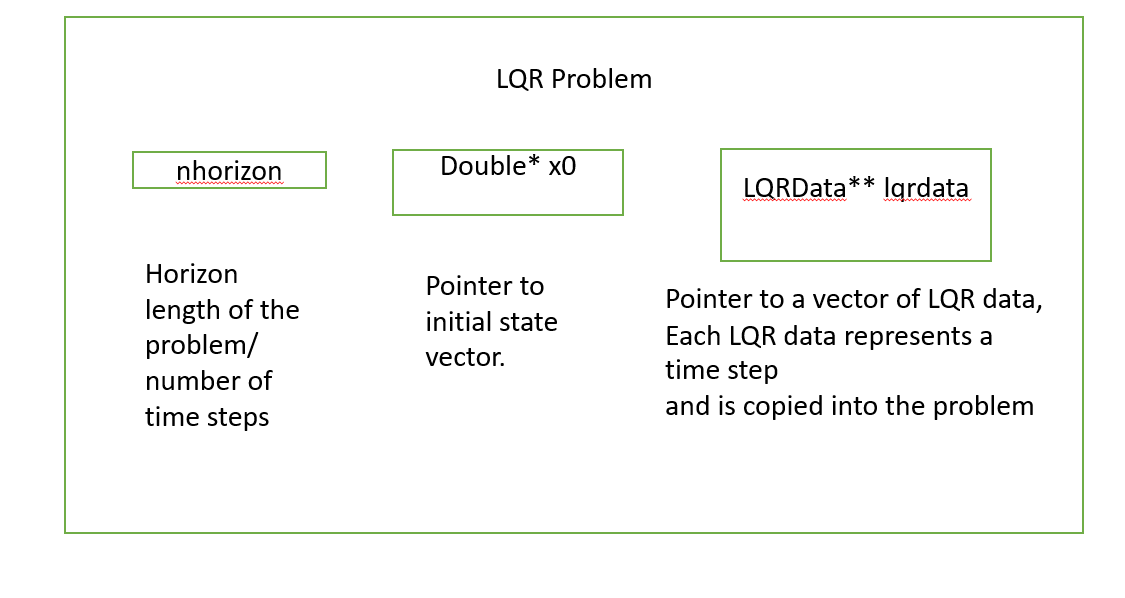


Figure LqrProblem picture

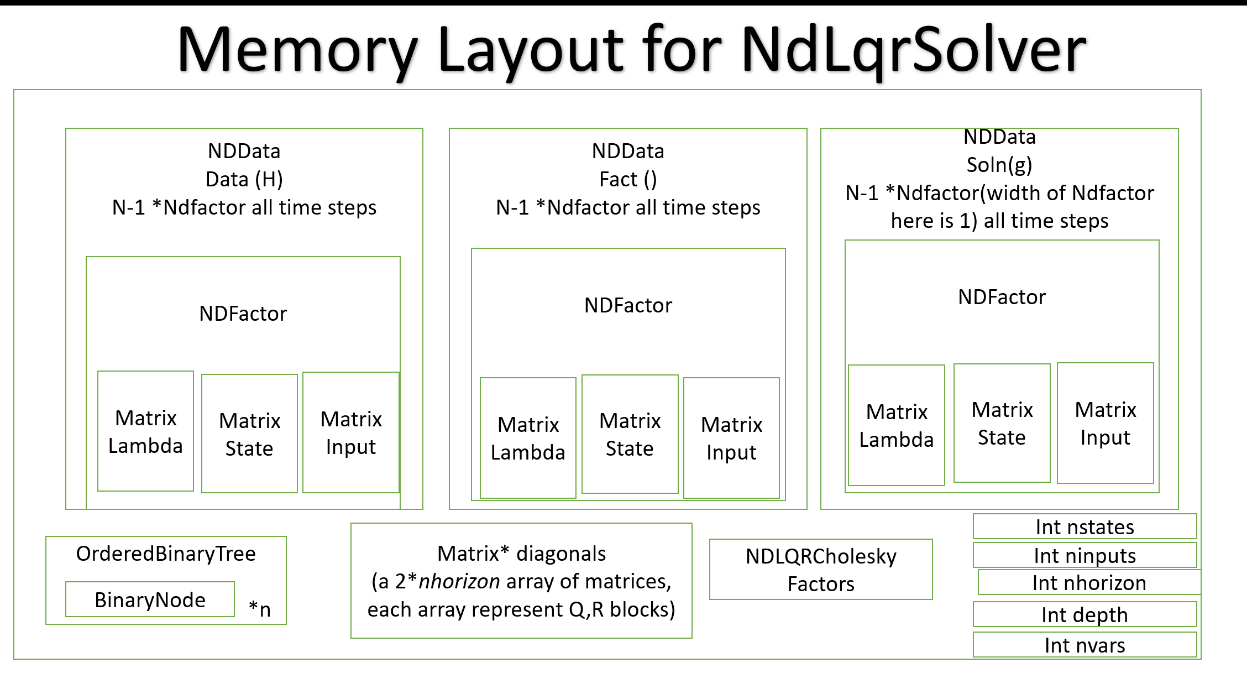


Figure 2 NdLqrSolver components

We use LqrProblem struct to initialize our problem (*ndlqr\_ReadLQRProblemJSONFile*) and upload our State, Control, and Dual variables. Then we initialize Solver with *ndlqr\_InitializeWithLqrProblem*, which already stores all the necessary initial information, and we proceed to solve the Matrix factorization using solver struct. We will look now into each struct separately building from bottom up.

* 1. **NdFactor**

Timeline

Description automatically generated

Figure NdFactor visualization

We use NdFactor to store a memory for a single time step in one chunk and take advantage of local memory. Each NdFactor has 3 matrices: dual variables, state variables and the control variables that come from the Dynamic and Cost functions. Each NdFactor represent one time step and Solver stores an array of NdFactors. NdFactor is also the main building block for NdData.

* 1. **NdData**

A picture containing diagram

Description automatically generated

Figure 4 NdData visualization

NdData is an array of memory blocks that can represent both the KKT matrix and the right-hand side vector. When storing the right-hand side vector the width of the NdFactor is equal to 1. It also stores general information about the KKT matrix/solution vector: number of states, number of segments, etc.

NdData Data stores the original data matrix in NdlqrSolver, the factorization of this matrix is stored in NdData Fact, we compute it as we try to solve the problem and use it at the end to find the solution vector. NdData soln(g) stores the right-hand side vector in the beginning of the algorithm and should store the solution upon the algorithm completion.

* 1. **NdLqrCholeskyFactors**

Diagram

Description automatically generated

Figure NdLqrCholeskyFactors visualization

Besides three NdData objects that store an array of NdFactors, Solver also has an object named NdLqrCholeskyFactors. This struct is used for storing the list of all cholinfo (the information for Cholesky Factorization) of the rsLQR solver. In the original code in C we have another class – cholinfo, which stores the Cholesky Decomposition itself, type of the matrix that was used (Upper/Lower),and information about what library was used during the decomposition. However, when implementing the code in python we substituted this class with convention always calling *scipy.linalg.Cholesky()* using Lower triangle matrix and storing a list of arrays (decompositions) in cholinfo.

* 1. **OrderedBinaryTree and BinaryNode**

Diagram

Description automatically generatedA picture containing text

Description automatically generated

Figure OrderedBinaryTree structure Figure BinaryNode structure

Another object that appears in NdLqrSolver is OrderedBinaryTree. This BinaryTree is built recursively by building the BinaryNode. This struct is a helper struct to make a transition between the hierarchial indexes and the knot-point indexes easier and faster. The tree doesn’t store anything in it besides the transition functions from one index to another.

* 1. **NdLqrSolver**

*See figure 1.*

Besides the solver variables that we already discussed (NdData, CholeskyFactors,BinaryTree…) NdLqrSolver also stores Q,R matrices in a separate array, as they can be computed independently. We also store general information about the problem s.t: ninputs,nstates…

Solver stores all the core information to solve the LQR problem and we will manipulate the original data in NdData data to solve the problem.

* 1. **LqrProblem and LqrData**

*See figure 2 for LqrProblem*  Diagram

Description automatically generated

Figure LqrData visualization

Finally, we have the structs LqrProblem and LqrData. LqrData stores the terms from the Cost function information and dynamics information. Basically it holds the data for a single time step of LQR problem and LqrProblem stores the the vector of LqrData’s. As mentioned above we initialize NdLqrSolver with LqrProblem variables.