**ndlqr\_Solve()**

Next we will analyze the main function for rsLQR which is ndlqr\_Solve(). We start with the abstract overview from Jackson’s paper[[1]](#footnote-1) and continue with the more detailed explanation, drawing connections between the paper and the github source. We attach the diagram for better understanding of the data flow which allows us to think about GPU implementations further.

1. **Absctract Overview**

*The abstract overview of the whole rsLQR is available on* [*https://github.com/bjack205/rsLQR/blob/main/docs/rslqr.dox*](https://github.com/bjack205/rsLQR/blob/main/docs/rslqr.dox) *and on* [*rslqr.pdf (bjack205.github.io)*](https://bjack205.github.io/papers/rslqr.pdf)*.*

To remind you by the time we call ndlqr\_Solve() we already have all our data in NdlqrSolver, where Q,R diagonal matrices are stored in separate array, the original data is in NdData Data and the solution/right hand side vector is in NdData soln, as we go through ndlqr\_solve() we are filling the NDData fact with factorization variables.

|  |  |
| --- | --- |
| for k=1:N do | For each timestep from 1 to Nhorizon |
| SolveLeaf(k) | Solves all the ‘independent blocks’ at the lowest level diagonal blocks, by timestep.  In terms of LQR problem calculates:  [Qk-1 Ak-T  Qk-1 qk  Rk-1 Bk-T  Rk-1 rk]  For each index k.[[2]](#footnote-2)  In terms of Shur compliments calculates:  *D* and *E* matrices |
| For j = 0: depth :  L= PowerOfTwo(depth-level-1)  Parfor i=0:L, p= j:K+1 :  FactorInnerProducts(i,j,p) | Once we have *D, E* matrices we can compute /bar b for level j, leaf i and parent level p. |
| For i= 0:L :  MatrixCholeskyFactorizeWithInfo(i,j) | Compute Cholesky Factorizatoin of bBar with index i, level j |
| For i=0:L, p = j:K+1  Solve Bbar(i,j,p) | After obtaining Cholesky Factorization of Bbar we can solve for y for all leaves I and upper levels p. |
| For k = 0:N, p = j:K+1  UpdateSchur(k,j,p) | Last step is to calculate x and z by updating a/bar,c/bar,D/bar, E/bar. |

1. **SolveLeaf() in details**

@param: solver; an initialized rsLQR solver

@param: index; knot-point index

@return 0 if successful[[3]](#footnote-3)

|  |  |
| --- | --- |
| **Lines 11-21:** Declaring the variables. | Initializing:  NdFactor\* C, NdFactor\* F (factorization), NdFactor\* z (soln vector), Matrix\* Q, Matrix\* R, CholeskyInfo\* Qchol=Null, CholeskyInfo\* Rchol = 0.  We will use these variables to manipulate Solver data. |
| **Lines 23-59:** Solves the independent blocks when knot-point index = 0 (first time step)  **Lines 23-44:** Manipulates NdFactor F and computes R | First, we extract the NdFactor at time step 0 from NdData data, NdData fact and NdData soln. Then we are solving the block system of equation for NdFactor F (Factorization of the original data). We also get R from solver and find Cholesky factorization for R. |
| **Lines 44-59:** Manipulates NdFactor z and computes Q | After solving for R, now we solve for Q by working with NdFactor z (soln vector). The last lines of this block of code is computing Cholesky factorization of Q. |
| **Lines 60**-**87:** SolveLeaf when knot-point index is != 0, N | The loop for index for 0<index<N, is following the same logic with a few changes:   * We get Cholesky Factorization of Q first as it is the same procedure index <N and index = N * Besides solving for F.input = Q\Cu (line 41 and 85), we are also solving for F.state = Q\Cx |
| **Lines 88-92:** SolveLeaf for the last time step. | Finally when solving for *k = nhorizon* we solve *z.state =* *Q\zx (z.state)*  This is the only term that is computed at the last time step. |
| **Lines 92-102:** Solve for the terms from the dynamics of the previous time step. | Solving for the terms from the dynamics (A,B) of the previous time step.  *By the end of SolveLeaf for each time step we have terms Q,R from the Cost function and A,B from the dynamics function.* |

Solver

[][

NDfactor\* C

NDfactor from NdData data

Ndlqr\_GetNdFactor(solver->data,0,j,&C)

Figure 1 Data flow from Solver to SolveLeaf vars

NDfactor from NdData fact

NDfactor\* F

Ndlqr\_GetNdFactor(solver->fact,0,0,&F)

NDfactor from Nddata soln

NDfactor\* z

Ndlqr\_GetNdFactor(solver->soln,0,0,&z)

Matrix diagonals [2\*k]

[\

Matrix diagonals [2\*k+1]

Matrix Q

Matrix R

Figure 1 Data flow from Solver to SolveLeaf variables

Diagram

Description automatically generated Timeline

Description automatically generated with medium confidence



Figure 2 NdFactor F after lines 35-42 and NdFactor z after lines 48-56

1. **FactorInnerProduct() in detail**

@param: data ; The data for the original KKT matrix

@param: fact; The current data for the factorization

@param: index; Knot point index

@param: data\_level; Level index for the original data, or equivalently current level being processed

@param: fact\_level; level index for the factorization data, also parent level/upper level

@return 0 if successful[[4]](#footnote-4)

|  |  |
| --- | --- |
| **Lines 114-125:** Declaring and initializing variables | Initializing:  NdFactor\* C1 - extracting from solver.data at requested index and level  NdFactor\* F1 (factorization)- extracting from solver.fact at requested index and parent level of C1  NdFactor\* C2 – extracting the next NdFactor data from solver at index+1 and the same level as C1  NdFactor\* F2 - extracting from solver.fact at requested index+1 and parent level of C1/C2  Matrix\* S = Lambda (y) matrix of F2 or equivalently of solver.fact at index+1,parent level. |
| **Lines 125-134:** Matrix multiplications | Multiplying Lambda (y) and Input(x) matrices of C1 \*F1 and C2\*F2.  Storing the result in S which is the inner products neede at current level. |

Solver

[][

NDfactor\* C1

NDfactor from NdData data

Ndlqr\_GetNdFactor(solver->data,index,data\_level,&C1)

Figure 1 Data flow from Solver to SolveLeaf vars

NDfactor from NdData fact

NDfactor\* F1

Ndlqr\_GetNdFactor(solver->fact, index, fact\_level,&F1)

NDfactor from Nddata data

NDfactor\* C2

NDfactor from NdData fact

[\

NdFactor\* F2

Matrix S

Ndlqr\_GetNdFactor(solver->fact, index+1, fact\_level,&F2)

Ndlqr\_GetNdFactor(solver->data,index,data\_level,&C1)

*Figure 1 Data flow from Solver to SolveLeaf vars*

Ndlqr\_GetLambdaFactor(F2)

1. **ndlqr\_SolveCholeskyFactlr() in detail**

@param: Data for the factorization

@param: cholinfo; Cached Cholesky factorization for \f$ \bar{\Lambda}\_{k+1}^{(j)}

@param: index; knot point index

@param: level; level index for the level currently being processed by the upper-level solve.

@param upper\_level; level index for right hand side

This is a short function that extracts F.lambda from solver.fact at index+1, cur\_level of solve and puts it into Matrix Sbar. It also extracts G.lamda from solver.fact at index+1, upper\_level and puts it into matrix f. Finally it solves for yi by calling MatrixCholeskyWithInfo(Sbar,f,cholinfo)

1. **ndlqr\_UpdateShur() in detail**

@param: fact; NdData for the factorization data

@param: solnNdData for the factorization data or solution vector

@param: index; knot point index of the ‘separator’ at level @p level.

@param i: know point index to be processed

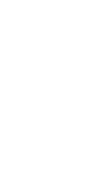


@param: level; level index for the level currently being processed .

@param upper\_level; level index of the upper level.

@param calc\_lambda; determined whether the lambda should be updated

|  |  |
| --- | --- |
| **Lines 158-1164:** Declaring and initializing variable | Initializing:  NdFactor\* f\_factor - extracting from solver.soln at requested @index+1 and @upper\_evel  NdFactor\* g - extracting from solver.soln at requested @index and @upper\_level  NdFactor\* F – extracting the next NdFactor fact from solver at @index and @level  Matrix\* f =f.\_factor.lambda |
| **Lines 165-170:** Matrix multiplications | Computing soln.lambda = - F.lambda. \* f  Soln.state = -F.state \* f  Soln.input = -F.input\*f |



1. [rslqr.pdf (bjack205.github.io)](https://bjack205.github.io/papers/rslqr.pdf) [↑](#footnote-ref-1)
2. Github documentation [↑](#footnote-ref-2)
3. Github documentation [↑](#footnote-ref-3)
4. Github documentation [↑](#footnote-ref-4)