Recent advances in the HPMPC and BLASFEO software packages

Gianluca Frison

Syscop group retreat

6 September 2016

HPMPC

- library for High-Performance implementation of solvers for MPC
- the QP solver is a Riccati based IPM
- linear algebra tailored for small-scale performance, hand optimized for many computer architectures
- outperforming similar solvers (e.g. FORCES) thanks to much better compuational performance

HPMPC ⇒ HPMPC + BLASFEO

- HPMPC: big software library (about 370k lines of code)
- split the library (work in progress...)
  - HPMPC: optimization algorithms for MPC
  - BLASFEO: linear algebra for embedded optimization

HPMPC

Improve reliability:

- more accurate solution
- possibly at the expense of a small preformance loss

Investigated techniques:

- in IPM, compute search direction step v.s. 'iterate'
- Riccati recursion as factorization of the KKT matrix: iterative refinement

Search direction in IPM

Given the QP

$$\begin{aligned} \min_x \quad & \tfrac{1}{2} x^T H x + g^T x \\ s.t. \quad & Ax = b \\ & Cx \geq d \end{aligned}$$

the KKT conditions are

$$\begin{aligned} Hx + g - A^T \pi - C^T \lambda &= 0 \\ Ax - b &= 0 \\ Cx - d - t &= 0 \\ \lambda^T t &= 0 \quad \Rightarrow \quad \Lambda T e = 0 \\ (\lambda, t) &\geq 0 \end{aligned}$$

The first 4 conditions are a system of nonlinear equations $f(y) = 0$.

Search direction in IPM

Search direction as Newton method step on the KKT conditions

$$\nabla f(y_k)\Delta y = -f(y_k)$$

giving

$$\begin{bmatrix} H & -A^T & -C^T & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & T_k & \Lambda_k \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \pi \\ \Delta \lambda \\ \Delta t \end{bmatrix} = - \begin{bmatrix} r_H \\ r_A \\ r_C \\ r_T \end{bmatrix}$$

with the residuals at the RHS

$$\begin{bmatrix} r_H \\ r_A \\ r_C \\ r_T \end{bmatrix} = \begin{bmatrix} Hx_k - A^T\pi_k - C^T\lambda_k + g \\ Ax_k - b \\ Cx_k - t_k - d \\ \Lambda_k T_k e \end{bmatrix}$$

Search direction in IPM

Rewritten as augmented system

$$\begin{bmatrix} H + C^T T_k^{-1} \Lambda_k C & -A^T \\ -A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \pi \end{bmatrix} = - \begin{bmatrix} r_H + C^T T_k^{-1}(r_T + \Lambda_k r_C) \\ -r_A \end{bmatrix}$$

where the RHS expression is

$$- \begin{bmatrix} (H + C^T T_k^{-1} \Lambda_k C) x_k - A^T \pi_k + (g - C^T(\lambda_k + T_k^{-1} \Lambda_k d)) \\ b - A x_k \end{bmatrix}$$

It is possible to compute directly the iterate $\tilde{y}_{k+1} = y_k + \Delta y$ as

$$\begin{bmatrix} H + C^T T_k^{-1} \Lambda_k C & -A^T \\ -A & 0 \end{bmatrix} \begin{bmatrix} \tilde{x}_{k+1} \\ \tilde{\pi}_{k+1} \end{bmatrix} = \begin{bmatrix} g - C^T(\lambda_k + T_k^{-1} \Lambda_k d) \\ b \end{bmatrix}$$

and the step in the search direction step as $\Delta y = \tilde{y}_{k+1} - y_k$

Search direction in IPM

- the direct computation of Δy requires the computation of residuals at the RHS ($\mathcal{O}(n^2)$ flops)
- the computation of Δy from $\tilde{y}_{k+1}$ does not require the computation of residuals at the RHS ($\mathcal{O}(n)$ flops)
- the procedures are equivalent in exact arithmetic...
- ... but not on finite-precision arithmetic

Search direction in IPM

- suppose $y^* = 5.0$, your current iterate y_k has 5 digits of accuracy but the conditioning of the LHS matrix gives 3 digits of accuracy
- not using residuals, Δy is computed as

$$\Delta y = \tilde{y}_{k+1} - y_k = 5.00365958 - 5.00004213 = 0.00361745$$

so (for $\alpha \approx 1$) the next iterate actually loses accuracy!!!

$$y_{k+1} = y_k + \alpha \Delta y = 5.00004213 + \alpha 0.00361745 \approx 5.0036$$

- using residuals, we have directy Δy with 3 digits of accuracy

$$\Delta y = -0.00004215$$

and then (for $\alpha \approx 1$) the next iterate has about 8 digits of accuracy

$$y_{k+1} = y_k + \alpha \Delta y = 5.00004213 - \alpha 0.00004215 \approx 4.99999998$$

Search direction in IPM

- in IPM, 3 digits of accuracy in the step Δy are enough (there is a safety factor of about 0.995 anyway to keep $(\lambda, t) > 0$)
- but conditioning gets increasingly worse at late IPM iterations
- idea: compute $\tilde{y}_{k+1}$ at early IPM iterations (possibly in single precision), use residuals close to solution (few iterations: region of quadratic convergence) for high-accuracy solution
- issue: switch point depends on conditioning of the system

Iterative refinement

- idea: use residual computation also in the solution of the equality-constrained QP giving the search direction
- may help if the system is badly conditioned and gives only a couple of digits of accuracy (e.g. late IPM iterations)
- e.g. iterative refinement in the solution of $M\Delta y = m$

factorize M
compute solution $\Delta y = M^{-1} m$
for $i = 1, 2, \dots, n_{\mathrm{ir}}$ **do**
    compute residuals $r_m = m - M\Delta y$
    solve for residuals $\delta y = M^{-1} r_m$
    update solution $\Delta y = \Delta y + \delta y$
end for

Partial condensing

- finally (being) embedded in the high-level HPMPC interface
  - invisible to the user, only one new argument N_p
- allows for arbitrary values for the new horizon length $1 \leq N_p \leq N$ (i.e. also different block sizes)
- uses the N^2 n_x^3 condensing algorithm (best choice for free x_0)
- recovers full space solution after QP solution (multipliers too)
- still work in progress:
  - general constraints to be done
  - atm the partial condensing happens in the feedback phase
  - needs extensive testing and debugging

BLASFEO

- BLAS For Embedded Optimization
- idea: take the linear algebra out of HPMPC, and make it available to implement other algorithms
- LA in HPMPC
  - focus on best possible performance for small matrices
  - use panel-major matrix format
  - main loop of each LA kernel is the `gemm` loop
  - LA kernels written as C function with intrinsics
- LA in BLASFEO
  - trade-off between performance and code size
  - focus on code reuse
  - use panel-major matrix format
  - LA kernels coded in assembly using custom function calling convention

Function calling convention in X86_64

- In Linux and Mac
  - first 6 arguments passed in GP registers (rdi, rsi, rdx, rcx, r8, r9)
  - the other arguments passed on the stack, one evey 64-bit (regardless the data type)
  - GP registers rbx, rbp, r12, r13, r14, r15 have to be saved on the stack and restored by the called function
  - the other GP registers can be freely modified
  - no arguments can be passed on the FP registers
  - the upper 256-bit of the FP registers must be set to zero before returning to the caller function
- On Windows, only the first 4 arguments are passed in GP registers
- not suitable to efficiently code small functions working on FP:
  - large overhead (lot of stuff to be saved on the stack)
  - FP registers can not be used to pass arguments

Function calling convention in BLASFEO

- LA kernels with same interface as in HPMPC
- but implemented calling many 'lightweight' functions (procedures) with local scope and custom calling convention
  - no use of stack
  - content of GP registers rdi, rsi, rdx, rcx, r8, r9 is untouched
  - int and pointers passed in GP registers r10, r11, r12, r13, 14, r15, also used for local int and pointers operations
  - first $n = 4$, 8 or 12 FP registers used as accumulation registers
  - remaining $(16 - n)$ FP registers used for local FP operations
- suitable to efficiently and modularly code LA kernels
  - procedures have very small overhead (about the same as 2 unconditional jumps - one for `call` and one for `ret`)
  - a procedure codes for an 'atomic' operation on FP registers
  - same procedure called by many LA kernels

Macro use in BLASFEO

- procedures can be easily replaced by macros
  - trade-off between code size and number of `call` and `ret` (and taget address misprediction)
- 3 levels of macros use
  - level 0: all procedures, no macros
  - level 1: `gemm` procedure, all others macros
  - level 2: no procedures, all macros
- trade-off small performance loss (1-2%) with substantial code size reduction (getting larger as more LA kernels are implemented)

BLASFEO

- still work in progress as well
- atm only LA routines needed for Riccati and condensing
- atm 4 architectures (plus generic code)
  - Intel Haswell 64-bit
  - Intel Sandy-Bridge 64-bit
  - Intel Core 64-bit
  - AMD Bulldozer 64-bit
- next ARMv8A ?
- code showcase