Implementation of a First-Order Quadratic Program Solver in C

Russell Burns[1]

The College of William and Mary, Williamsburg, Virginia 23185, USA

Glen Gimpl[2]

University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

Steven Williams[3]

Embry-Riddle Aeronautical University, Daytona Beach, Florida 32114, USA

James Yu[4]

Louisiana State University, Baton Rouge, Louisiana 70803, USA

This paper details a translation of a first order quadratic program (QP) solver from MATLAB to C. NASA could use this QP solver to generate online flight path trajectories for powered descent vehicles during landing. Over 12 weeks, the team designed, implemented, and tested two iterations of the QP solver for accuracy and runtime on 104 benchmark QP tests. The final iteration was 541.07% faster than the first, handling most tests in under one second. Additionally, it solved four more QP tests for $N \geq 1383$, and all outputs for cost and D_x matched the MATLAB reference values.

Nomenclature

$\boldsymbol{\rho}$	=	Step size matrix
$\boldsymbol{x}$	=	Objective variable
$\bar{\boldsymbol{x}}$	=	Transformed objective variable
$\varepsilon_{\text{prim}}$	=	Maximum error in primal solution
$\varepsilon_{\text{dual}}$	=	Maximum error in dual solution
ϵ_{abs}	=	Absolute tolerance
ϵ_{rel}	=	Relative tolerance
σ	=	ADMM sigma step
α	=	ADMM relaxation parameter
ρ	=	ADMM rho step
k_{update}	=	Update rho interval
max_{iters}	=	Maximum number of iterations
N	=	$\text{nnz}(\boldsymbol{A}) + \text{nnz}(\boldsymbol{B})$

[1] Undergraduate Degree, Engineering Physics and Applied Design
[2] Undergraduate Degree, Mathematical Physics
[3] Undergraduate Degree, Aerospace Engineering
[4] Undergraduate Degree, Mechanical Engineering

I. Introduction

Given $n, m \in \mathbb{R}$, $\boldsymbol{q} \in \mathbb{R}^n$, $\boldsymbol{P} \in \mathbb{R}^{n \times n}$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\boldsymbol{l}, \boldsymbol{u} \in \mathbb{R}^m$, with $m > n$, Oxford University's Operator Splitting Quadratic Program (OSQP) solver is a first-order method that solves QPs of the form [1]:

$$\begin{aligned} &minimize\ \frac{1}{2}\boldsymbol{x}^T\boldsymbol{P}\boldsymbol{x} + \boldsymbol{q}^T\boldsymbol{x} \\ &subject\ to\ \boldsymbol{l} \preceq \boldsymbol{A}\boldsymbol{x} \preceq \boldsymbol{u} \end{aligned} \tag{1}$$

where $\boldsymbol{x}^T$ denotes the vector transpose of $\boldsymbol{x}$, and $\boldsymbol{l} \preceq \boldsymbol{A}\boldsymbol{x} \preceq \boldsymbol{u}$ asserts that every entry of the vector $\boldsymbol{A}\boldsymbol{x}$ is greater than its corresponding element in vector $\boldsymbol{l}$, and less than its corresponding element in vector $\boldsymbol{u}$. The minimized expression is called the cost function. First order QP solvers can be sensitive to ill-conditioning and lack adaptive measures to improve convergence speed. OSQP addresses these issues by performing step-size tuning and input matrix pre-processing. OSQP can also solve initial value problems. In general, regardless of the heuristics employed, first order QP solvers are not as accurate as second order solvers.

Optimization problems of this form arise in several disciplines, including engineering, finance, and operations research [1]. QPs are also used during rocket and drone landings by treating their solutions as model predictive control (MPC) inputs [2]. MPCs predict the behavior of linear, time-invariant, dynamical systems over a finite time window. Oxford's OSQP has features such as solution polishing and infeasibility detection, which are unnecessary for solving known MPC problems. Hence, a lite version of OSQP, called OSQP_lite was developed in MATLAB. OSQP_lite solves QPs of form (1), outputting a state vector and residual vectors. This version served as the reference implementation. Two QPs served as initial input references: a simple 4x2 problem and a powered descent vehicle problem.

Duality is an important concert for understanding a convex optimization algorithm's efficacy. Convex optimization problems, by nature, have two different solution approaches. There are the "primal" and "dual" approaches, which oppositely attempt a solution via minimization or maximization. How closely both approaches align at each iteration is important, as the answer to one provides an upper or lower solution bound for the other. This algorithm's per-iteration solution bounds, referred to as residuals, take the form [1]:

$$\boldsymbol{residual_{prim}} = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{z} \quad \& \quad \boldsymbol{residual_{dual}} = \boldsymbol{P}\boldsymbol{x} + \boldsymbol{q} + \boldsymbol{A}'\boldsymbol{y} \tag{2}$$

where $\boldsymbol{y}, \boldsymbol{z} \in \mathbb{R}^n$, and $\boldsymbol{y}$ is the Lagrange multiplier associated with the constraint $\boldsymbol{A}\boldsymbol{x} - \boldsymbol{z}$. OSQP's second output stores per-iteration residual vectors within $\boldsymbol{rprimlog} \in \mathbb{R}^{m \times (\text{max\_iters})}$ and $\boldsymbol{rduallog} \in \mathbb{R}^{n \times (\text{max\_iters})}$. The algorithm terminates when the infinity norm of each residual vector is within its respective maximum error threshold ε. The maximum error threshold is equal to solution precision tolerance. The final difference between residuals is called the duality gap. A duality gap of zero defines strong duality, which indicates feasible inputs and a rigorous algorithm.

The team tackled numerous obstacles spanning all stages of the development process. First, the team needed to decide how to implement matrix datatypes and linear algebra operations in C. The team initially used the Neat Matrix Library (NML), an open-source linear algebra library, as it provided most of the data handling required to implement OSQP in C. The team wrote the rest of the required operations using NML's datatypes as storage and organized them under the eponymously named "Gruven" library. The team named this first version OSQP_NML. After timing OSQP_NML against OSQP_lite, the team determined that optimization was necessary. The team refactored the entire algorithm by substituting all NML datatypes and operations for Basic Linear Algebra Subprograms (BLAS) datatypes and operations. BLAS encompasses a library of low-level, fundamental operations on vectors and matrices based on FORTRAN implementations, and its C wrapper is called CBLAS [3]. The team named this second version OSQP_BLAS. Both versions included accessibility features that (1) allow the user to call each solver from MATLAB, and (2) allow each solver to accept pre-formatted matrix .txt inputs. The team enabled MATLAB invocation by converting each OSQP version into a .mex file via MATLAB's C API. This feature facilitates easy access to the algorithm for those more comfortable in a MATLAB environment. OSQP's standalone development was important for enabling streamlined diagnostics, thus motivating the need for the algorithm to also read in matrices from text files.

The team benchmarked and unit tested both solver versions using POST2 within the NASA Langley EDL (Entry, Descent & Landing) Cluster. This cluster is a shared lab space consisting of many powerful lab machines. POST2 (Program to Optimize Simulated Trajectories II) is an ITAR restricted code base which is used for point mass trajectories for powered and un-powered vehicles. POST2 also has the capabilities to run 6DOF and closed loop GNC

trajectories [4]. The team benchmarked against 104 QPs from the Maros and Meszaros (MM) set [5]. The MM set is a comprehensive collection of standard convex QP examples.

II. Adaptive Step Sizing

The number of iterations to reach a solution depends on the diagonal step-size matrix $\boldsymbol{\rho}$. With a fixed $\boldsymbol{\rho}$, any QP solver may converge slowly. Adaptively changing the diagonal elements of $\boldsymbol{\rho}$ can greatly improve solver performance for a given QP, so OSQP offers an adaptive step-size function named update_rho [1]. This function modifies the step-size matrix every k_{update} iterations, which is fixed to 25 by default. This function is costly, requiring an LDL^T factorization, hence why it is not run each iteration. By updating the step-size throughout runtime, the solver can converge with less iterations and perform less operations overall. Eventually, for convergent inputs, the norms of the residuals r_{prim}^k and r_{dual}^k respectively decrease below $\varepsilon_{\text{prim}}$ and $\varepsilon_{\text{dual}}$, thus terminating the solver's execution [1].

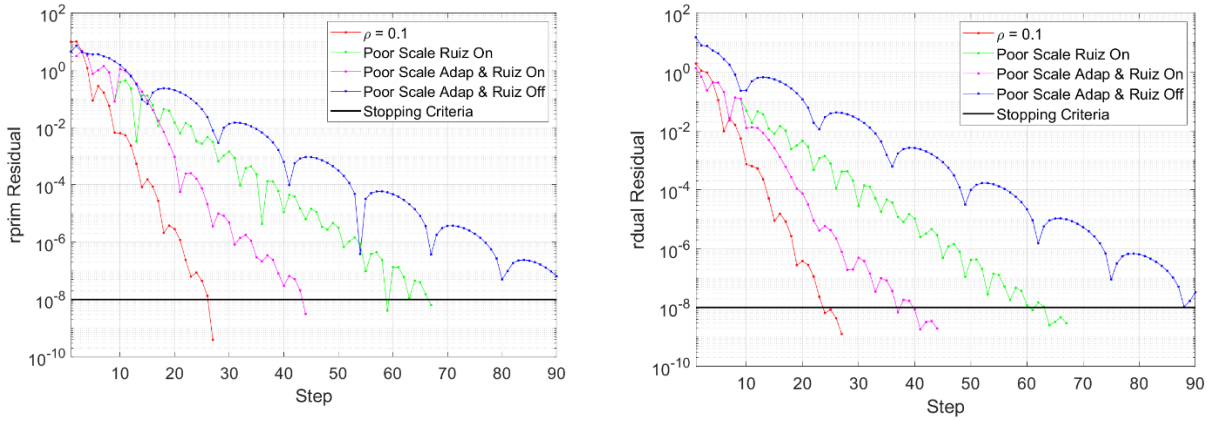

Figure 1. A simple QP solved in MATLAB using OSQP_lite with small $\boldsymbol{\rho} = 0.1$, poorly scaled $\boldsymbol{\rho} = 5$, and adaptive $\boldsymbol{\rho}$

Fig. 1 illustrates the importance of an adaptive $\boldsymbol{\rho}$, as even the initial choice of $\boldsymbol{\rho}$ has a large effect on the number of iterations it takes to solve the program. This figure shows the primal (left) and dual (right) residuals for a simple QP test. Each line represents the same problem with different solver settings. In the QP test case above, the precision tolerance is 1e-8.

III. Input Matrix Pre Processing

Ill-conditioned inputs to a QP solver drastically increase the number of iterations required to reach a solution when compared to well-conditioned inputs [1]. Ruiz Equilibration is a pre-processing algorithm that improves the conditioning of the inputs and normalizes the cost function so that solvers need less iterations to converge. OSQP implements this conditioning in a function named ruiz. Ruiz Equilibration performs a change-of-variables on the original problem form, shown in Eq. (1), resulting in a well-conditioned problem form, shown in Eq. (3):

$$\begin{aligned} &\textit{minimize } \frac{1}{2}\boldsymbol{x}^T\boldsymbol{cDPD}\bar{\boldsymbol{x}} + \boldsymbol{cq}^T\boldsymbol{D}\bar{\boldsymbol{x}} \\ &\textit{subject to } \boldsymbol{El} \preceq \boldsymbol{EAD}\bar{\boldsymbol{x}} \preceq \boldsymbol{Eu} \end{aligned} \tag{3}$$

where $\boldsymbol{c} \in \mathbb{R}^n, \boldsymbol{E}$ and $\boldsymbol{D}$ are diagonal matrices with positive entries determined via the Karush-Kuhn-Tucker (KKT) first-order derivative tests [1]. KKT transforms inputs $\boldsymbol{P}, \boldsymbol{A}, \boldsymbol{l},$ and $\boldsymbol{u}$ into $\boldsymbol{DPD}, \boldsymbol{EAD}, \boldsymbol{El}$, and $\boldsymbol{Eu},$ respectively.

IV. Linear Algebra Libraries

1. *Neat Matrix Library (NML)*

NML is a user-friendly, zero-dependency linear algebra library written in C [6]. It supports elementary operations such as matrix addition, multiplication, and row/column swaps. It can also perform LU and QR decomposition, find inverses, determinants, and solve linear systems of equations using simple methods. NML uses the "NML_mat" struct to store matrix data, including the number of rows and columns, a Boolean for if the matrix is square, and all its defined entries. NML offered the team an intuitive set of functions to use while coding OSQP_NML, and debugging

was relatively simple since the library is open source. However, the naïveté of NML comes at the cost of the methods being slow. NML_mat structs use nested arrays to store data, which takes more memory than unnested array implementations. Most fundamental operations are entirely unoptimized—matrix multiplication, for example, does not have a check to handle large inputs with, say, the Strassen algorithm. NML also lacks any functions that (1) calculate matrix and vector infinity norms, (2) perform element-wise multiplication, division, and indexing, (3) create a diagonal matrix from a vector, (4) replace a matrix column with a vector, or (5) calculate the number of non-zero entries in a matrix, all of which were necessary for the initial OSQP translation from MATLAB to C.

2. BLAS and CBLAS

BLAS has several, sometimes competing implementations, such as IntelOneMKL and OpenBLAS. The EDL Cluster setup for the team simply uses the reference BLAS available on netlib.org, specifically with its CBLAS wrapper. Though the reference implementation is unoptimized [7], its assembly language coded subprograms would save time regardless [8]. With this information in mind, the team sought out to rewrite OSQP_NML using BLAS functions and logic. The goal was to reduce the amount of memory management performed mid-execution through optimizing the implementation of various methods, in addition to minimizing calls to memory-related functions directly. Implementation changes from OSQP_NML to OSQP_BLAS included reducing the number of matrix concatenation operations, altering the application of temporary matrices, and the near elimination of structs.

The first step in the development process was to review the functions within the Gruven library. Many of them were obsolete since BLAS provided workarounds. For example, infinity norms can be calculated using cblas_idamax to find maximum magnitude elements, as opposed to the manual parsing done in Gruven using the team's own functions. After recoding the lower-level helper functions, the team converted the function blocks for update_rho, ruiz, and the main function itself. The largest optimization the team made increases memory reusage by allocating all memory for matrices, intermediate and permanent, at the start of the solution process. The temporary matrices are only freed once the solver has terminated. This method of memory management made debugging more difficult; since memory was reused within each execution, the source of leaks and corruptions were difficult to trace.

V. MEX Conversion

A MEX file is compiled and executed from MATLAB by targeting a C program written with MATLAB's C API. Successful compiling enables the user to easily define and send in a .mat to either C version of OSQP. The team performed a successful MEX conversion for OSQP_NML. A mex function was written for OSQP_BLAS, but the team could not properly compile it with the Cluster's BLAS libraries. The MATLAB C API uses the mxArray data structure, which contains much less information than the NML_mat struct. So, for OSQP_NML, the team had to write a wasteful conversion function to turn input mxArrays into input NML_mat structs, and another function to turn output NML_mats into output mxArrays. The MATLAB C API contains functions to turn mxArrays into double pointer arrays, which is the datatype that BLAS uses. Thus, conversion was also necessary between MATLAB and OSQP_BLAS, though it is natively supported. Because of this support, the MEX implementation for OSQP_BLAS handles data much smoother than the OSQP_NML implementation. Since NML is zero-dependency, the act of compiling the MEX file was trivial. When compiling the MEX file for OSQP_BLAS, needs to correctly include the statically built BLAS library located in their working directory. This inclusion was not successful due to several undefined errors—the team looked at issues ranging from memory discrepancies between compiler and library to simple syntax differences, to no avail. The program includes README instructions regarding possible solutions.

VI. .txt Conversion

The MATLAB interface, while useful, lacks robust diagnostics. OSQP's standalone development is essential to enabling streamlined benchmarking and unit testing, so the team explored the algorithm's ability to accept simple input formats. Obvious choices included .xls, .xlsx, .csv, and .txt. The team initially decided they needed to be able to store matrices as large as 20,000 x 20,000; XLS and XLSX were immediately ruled out, as XLS and XLSX only support up to 65,536 x 256 and 1,048,576 x 16,384, respectively. CSV and TXT files can store as much data as the computer has as available storage. However, Excel can only display up to the size of an XLSX, so CSVs are not optimal. Notepad does not have this issue with displaying TXTs. Though Notepad only creates columns up to the width of the screen, the data in a TXT can be custom formatted to any number of rows or columns and scroll through its entirety. Additionally, one can append matrix dimensions to the top of the TXT and interpret them with a *scanf* command in C. The NML library reads TXTs into an NML_mat, and the team used a similar process for OSQP_BLAS.

VII. Benchmarking and Unit Testing Procedure

3. Benchmarking Constraints

Benchmarking is a critical aspect of the project, providing a quantifiable measure of success for the program. The primary goal of benchmarking is to evaluate the performance (runtime) of the QP solver under various scenarios. The team benchmarked the MM test set on OSQP_NML, OSQP_BLAS, and OSQP_lite. These test results allowed for direct comparison with Oxford's OSQP. The MM set is very thorough, encompassing a wide range of sparsity patterns and matrix sizes. Due to their size, nine QP tests from the original 140 were unable to be converted to .txt format. In order to time MM set performance, the team wrote a benchmarking program to run on the EDL Cluster. Lab 36, which has 112 threads and 192 GB of available memory, served as the machine of choice. On the Cluster, each QP test had its own folder containing the input matrices of each test case for the solver. A QP test case refers to a run of a QP test done on one of the three precision tolerances: 1e-3, 1e-5, or 1e-8. The team ran every test in that exact test case order.

Benchmark QP Test Parameters
$\epsilon_{abs} = \epsilon_{rel} = 10^{-3}$
$\epsilon_{pinf} = \epsilon_{dinf} = 10^{-4}$
$\rho = 0.1\ (adaptive)$
$\sigma = 10^{-6}$
$\alpha = 1.6$
$k_{update} = 25$
$max_{iters} = 4000$

Table 1. Two QP tests running three different accuracies and their runtime on OSQP_BLAS.

Test Name	**HS21**			**VALUES**		
Test Case	1.00E-03	1.00E-05	1.00E-08	1.00E-03	1.00E-05	1.00E-08
Time (s)	0.001156	0.001161	0.001231	3.919712	16.49957	40.93608
Standard Deviation	0.000934	0.000958	0.000953	0.009626	0.045423	0.088018

A preliminary benchmark ran OSQP_BLAS on nine QP tests of varying N, performing 100 executions for each test case to track standard deviation. Table 1 shows results for two of those tests; runtime standard deviation is clearly negligible for OSQP_BLAS. Thus, all final benchmarks used a sample size of one. Any test with a row or column count larger than 10,000 was disqualified. This constraint eliminated 27 tests, bringing the final total to 104 tests. The team defined $N = 1383$ as an important standard for the QP solver; $N = 1383$ for the sample powered descent problem.

For benchmarking, the team decided to allow 600 seconds maximum for each test case to converge. The team classified each overall test as either a runtime failure, partial runtime failure, iteration (iter) failure, partial iter failure, or a success. A runtime failure occurred when a 1e-3 test case passed 600 seconds, while an iter failure occurred when a 1e-3 test case did not converge within 4000 iters. A partial iter failure successfully converged for one or two of the three precision tolerances but failed to converge for the remainder. The same logic applies to the partial runtime failure.

4. GoogleTest

The Google Test is a C++ library utilized for comparative testing. When built, POST2 runs the GoogleTest platform for any unit tests written on the module. All unit testing in the POST2 simulation utilizes a C file for allocating memory and inputting variable values into a struct, followed by a C++ file to run the GoogleTest function using those values. The team decided to utilize this same testing platform to ensure accuracy of OSQP_NML and OSQP_BLAS against OSQP_lite. Two build scenarios utilized the GoogleTest: a test-suite-name (TSN) and a value-parameterized test-suite (VPTS). A single QP test scenario is what defines a test-suite-name, along with the test-name to be run. The simple single test will run anytime the POST2 project is built, like all other unit tests. A value-parameterized test-suite is what handles dynamic test fixtures, meaning the inputs change in value and size. POST2 uses this suite for an all-QP-test checkout of either C version. This suite also works when running two or more tests that operate on similar data [9]. The team chose the cost and the state vector $\boldsymbol{D_x}$ as the comparative output variables for every unit test.

```
1276  [----------] 27 tests from nasaQP/osqp_lite
1277  [ RUN      ] nasaQP/osqp_lite.nominal/0
1278  [       OK ] nasaQP/osqp_lite.nominal/0 (3022 ms)
1279  [ RUN      ] nasaQP/osqp_lite.nominal/1
1280  [       OK ] nasaQP/osqp_lite.nominal/1 (229 ms)
1281  [ RUN      ] nasaQP/osqp_lite.nominal/2
1282  [       OK ] nasaQP/osqp_lite.nominal/2 (3940 ms)
1283  [ RUN      ] nasaQP/osqp_lite.nominal/3
1284  [       OK ] nasaQP/osqp_lite.nominal/3 (88 ms)
1285  [ RUN      ] nasaQP/osqp_lite.nominal/4
1286  [       OK ] nasaQP/osqp_lite.nominal/4 (469 ms)
```

Figure 2. VPTS Scenario for OSQP_BLAS Unit testing Results

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from nasaQP/osqp_lite
[ RUN      ] nasaQP/osqp_lite.nominal/0
DUAL1
../artemis_src/tools/osqp_lite//test/qp_nasa_unittest.cpp:93: Failure
The difference between test_cost[0][0] and osqp_out_vect.cost is 1.9999999999638023, which exceeds tolerance, where
test_cost[0][0] evaluates to 6.2202509770000001,
osqp_out_vect.cost evaluates to 8.2202509769638024, and
tolerance evaluates to 1e-08.
../artemis_src/tools/osqp_lite//test/qp_nasa_unittest.cpp:97: Failure
The difference between test_dX[0][jj] and osqp_out_vect.D_x[jj] is 1.999999999677198, which exceeds tolerance, where
test_dX[0][jj] evaluates to 2.0052447409999998,
osqp_out_vect.D_x[jj] evaluates to 0.0052447413228017999, and
tolerance evaluates to 1e-08.
[  FAILED  ] nasaQP/osqp_lite.nominal/0, where GetParam() = "DUAL1" (79 ms)
[----------] 1 test from nasaQP/osqp_lite (79 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (79 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] nasaQP/osqp_lite.nominal/0, where GetParam() = "DUAL1"

 1 FAILED TEST
```

Figure 3. TSN Scenario for OSQP_BLAS Unit testing Results

Figures 2 and 3 show the output when building POST2 with a project. During the VPTS GoogleTest displayed in Fig. 2, POST2 unit tested 27 QP tests—all of them passed. VPTS builds in POST2 are a developmental feature that would not be of interest to a normal user unless they intend to unit test the outputs of another set of QPs. During the TSN GoogleTest displayed in Fig 3, both cost and $\boldsymbol{D_x}$ evaluate to outside of the 1e-8 precision tolerance. By default, a POST2 build runs a TSN GoogleTest on OSQP_BLAS with the DUAL1 QP from the MM set.

VIII. Benchmarking and Unit Testing Results

5. OSQP_NML

Table 2. OSQP_NML Benchmarking Results

Class	Success	Iter Failure	Partial Iter Failure	Runtime Failure	Partial Runtime Failure
# of tests	29	12	3	60	1
Percentage	27.62	11.43	2.86	57.13	0.95

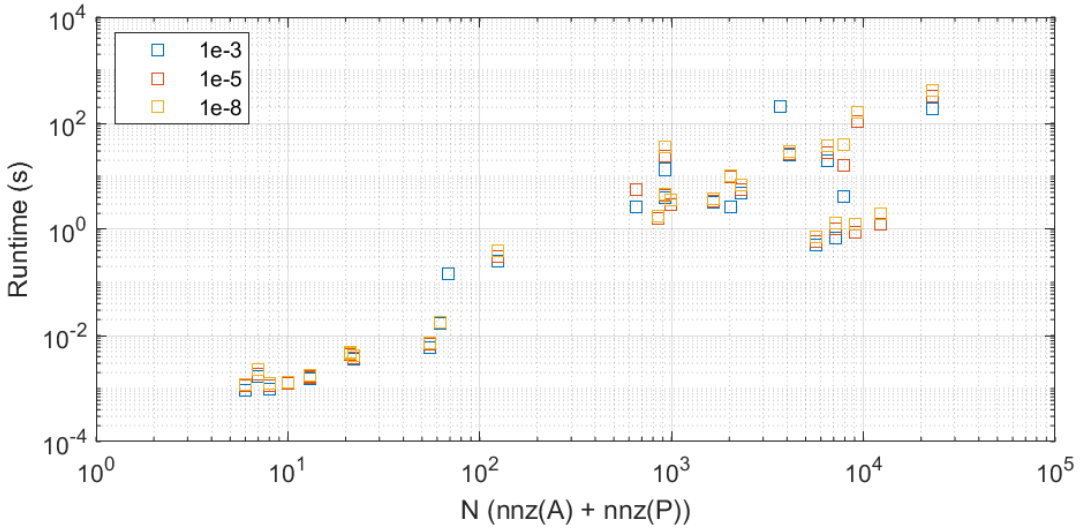

Figure 4. Computation Time vs Problem Dimension for OSQP_NML

Table 3. Number of Tests Solved Across Differing Classes and Ranges for OSQP_NML.

Range or Class	N >= 1383	t < 0.1 (s)	t > 10 (s)
# Tests Solved	13	13	6

6. OSQP_BLAS

Table 4. OSQP_BLAS Benchmarking Results

Class	**Success**	**Iter Failure**	**Partial Iter Failure**	**Runtime Failure**	**Partial Runtime Failure**
# of tests	27	23	5	48	2
Percentage	25.71	21.90	4.76	45.71	1.90

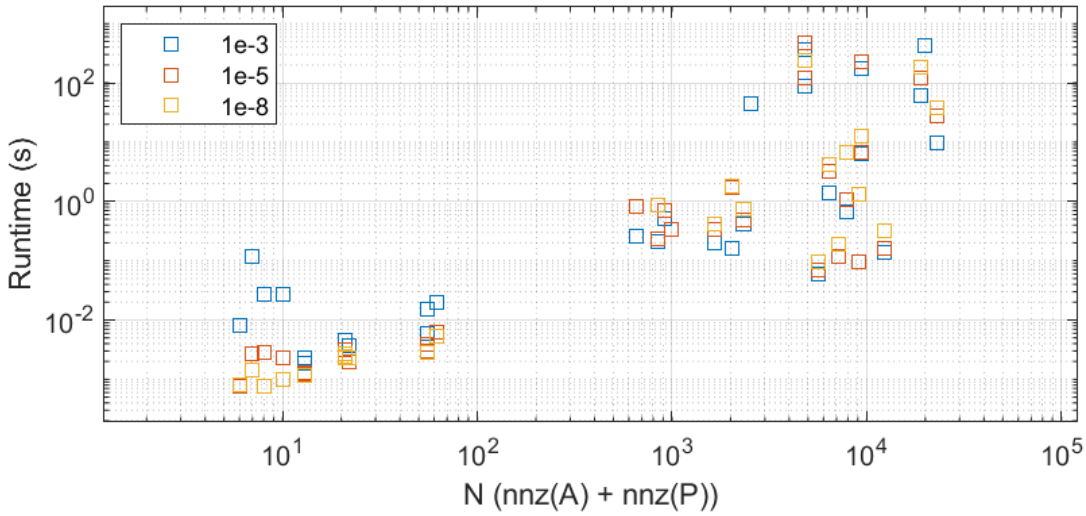

Figure 4. Computation time vs problem dimension for optimized second iteration of code.

Table 5. Number of tests solved for differing classes and ranges for OSQP_BLAS.

Range or Class	**N >= 1383**	**t < 0.1 (s)**	**t > 10 (s)**
# Tests Solved	17	14	6

7. Unit Testing and Benchmarking Analysis

For OSQP_NML, most test classes were runtime failures. These results motivated the development of OSQP_BLAS. The team hoped OSQP_BLAS's optimization would solve some of those runtime failures and fill in the gaps in the middle of OSQP_NML's performance plot. OSQP_NML and OSQP_BLAS respectively solved 92 and 93 tests cases, 79 of which they both solved. On average for these shared cases, OSQP_BLAS was 541.07% faster than OSQP_NML. OSQP_NML solved six tests constituting eight test cases that OSQP_BLAS failed on iters. OSQP_BLAS solved a single test case that OSQP_NML failed on iters. OSQP_NML did not solve any test cases that OSQP_BLAS failed on runtime. OSQP_BLAS solved four tests constituting 10 test cases that OSQP_NML failed on runtime. OSQP_BLAS partially failed on iters for a single test that OSQP_NML failed on runtime. Overall, OSQP_BLAS solved four more tests with an $N \geq 1383$ than OSQP_NML. The results show no change the number of tests solved in greater than 10 seconds, though OSQP_BLAS solved one more in under 0.1 seconds. Overall, OSQP_BLAS handled tests with large a N slightly worse than OSQP_NML. While it had fewer runtime failure test classes than OSQP_NML, it also failed on tests the team expected it to solve. The runtime gain was very large, allowing the algorithm to solve a few new problems of N between 10^3 and 10^4, though OSQP_BLAS did not fill the desired graphical gap. Unit testing was successful—every OSQP_BLAS and OSQP_NML benchmark success test case matched the values for cost and $\boldsymbol{D_x}$ that OSQP_lite outputted at the same precision tolerance.

IX. Conclusion

The team successfully wrote two versions of a lite QP solver in C that NASA could use to generate online flight path trajectories for powered descent vehicles during landing. The team benchmarked and unit tested both versions to ensure accuracy. Making up for lacking robustness with its speed, OSQP_BLAS is now incorporated into POST2. One could improve the performance diagnostics of this solver by tracking runtime on a GPU, as it is less noisy than the CPUs used and would measure runtimes better. One could also track the solver's duality gap for all test cases, as the team did not consider this metric. There are several ways one could improve the runtime of this algorithm. For example, switching its BLAS library from the reference implementation to OpenBLAS or OneIntelMKL. These optimized versions take advantage of parallelism, allowing the CPUs to break down matrix and vector operations in kernels that run simultaneously [10]. The EDL Cluster has OpenBLAS readily available, so a user would need to rebuild their working directory and replace the functions to try it. The team also has concerns about OSQP_BLAS's handling of sparse matrices. Native OSQP stores matrices in compressed sparse column (CSC) format [1]. Implementing this technique and using the CSparse library [11] for LDL factorization could offer large performance gains. Conversely, one could further mimic OSQP's implementation by installing QDLDL, the free open-source LDL factorization routine Oxford used [1]. The benefits of using QDLDL are the same as CSparse. Lastly, in practice, precision tolerance is a dynamic quantity that should be put into the form using the equations on page (9) of [1]. Overall, while the ability to solve form (1) optimization problems is an asset to NASA, it is also a relatively new one. Industry continually shows the utility of those QP solutions. This OSQP and its extensively documented development can and should serve as a blueprint for future iterations to keep NASA on the cutting edge.

Acknowledgements

The GNC team would like to acknowledge and thank our project mentor, Jing Pei. His willingness to introduce us to challenging problems and guide us through them has been an invaluable learning experience thus far. Thanks as well to Rafael Lugo—he was very helpful in improving our understanding of OSQP's implementation and enabling use of the Cluster. Kudos to William Colson for his perspective on the Cluster's two versions of BLAS. Last, but certainly not least, the GNC team extends our deepest gratitude to Eloisa Carrasco and Elizabeth Corry for their insight on the structure and standards of this paper. Eloisa Carrasco was also a great resource when developing the unit testing programs.

References

[1] Stellato, B., Banjac, G., Goulart, P., Bemporad, A., and Boyd, S. "OSQP: An operator splitting solver for quadratic programs," *Mathematical Programming Computation*, Vol. 12, No. 4, 2020, pp. 637-672. https://doi.org/10.48550/arXiv.1711.08013.

[2] Stellato, B., *Learning for Fast and Robust Real-Time Optimization* Available: https://stellato.io/assets/downloads/presentations/2022/berkeley.pdf.

[3] Burkardt, J., "CBLAS demonstrate the use of the C translation of the blas," *CBLAS - Demonstrate the use of the C Translation of the BLAS* Available: https://people.math.sc.edu/Burkardt/cpp_src/cblas/cblas.html.

[4] Brauer, G. L., Cornick, D. E., Habeger, A. R., Peterson, F. M., and Stevenson, R., *Program to optimize simulated trajectories (post). volume 3: Programmer's Manual - NASA Technical Reports Server (NTRS)* Available: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19750024075.pdf.

[5] "The Maros and Meszaros convex quadratic programming test problem set," *Maros and Meszaros's Convex QP Test Problem Set* Available: https://www.cuter.rl.ac.uk/Problems/marmes.shtml.

[6] "nomemory/neat-matrix-library: nml" *Github* Available: https://github.com/nomemory/neat-matrix-library/blob/main/nml.c.

[7] "BLAS Frequently Asked Questions (FAQ)," *Blas (Basic Linear Algebra Subprograms)* Available: https://www.netlib.org/blas/.

[8] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic Linear Algebra Subprograms for Fortran Usage. ACM Trans. Math. Softw. 5, 3 (Sept. 1979), 308–323. https://doi.org/10.1145/355841.355847

[9] "Google/googletest: GoogleTest - Google Testing and mocking framework," *GitHub* Available: https://github.com/google/googletest/tree/main.

[10] "Parallelism," *Developer Reference for Intel® oneAPI Math Kernel Library - C* Available: https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-1/overview.html.

[11] Davis, T. A., *Direct Methods for Sparse Linear Systems*, Philadelphia: Society for Industrial and Applied Mathematics, 2006.

Appendix

Table 6. Results for MM Set on OSQP_BLAS Using "Benchmark QP Test Parameters"

QP Name	Status	Time (s) 1.00E-03	1.00E-05	1.00E-08	N
AUG2D	Disqualify	NaN	NaN	NaN	80000
AUG2DC	Disqualify	NaN	NaN	NaN	80400
AUG2DCQP	Disqualify	NaN	NaN	NaN	80400
AUG2DQP	Disqualify	NaN	NaN	NaN	80000
AUG3D	Runtime Failure	NaN	NaN	NaN	13092
AUG3DC	Runtime Failure	NaN	NaN	NaN	14292
AUG3DCQP	Runtime Failure	NaN	NaN	NaN	14292
AUG3DQP	Runtime Failure	NaN	NaN	NaN	13092
CONT-050	Runtime Failure	NaN	NaN	NaN	17199
CONT-100	Disqualify	NaN	NaN	NaN	69399
CONT-101	Disqualify	NaN	NaN	NaN	62496
CVXQP1_L	Disqualify	NaN	NaN	NaN	94966
CVXQP1_M	Runtime Failure	NaN	NaN	NaN	9466
CVXQP1_S	Partial Iters Failure	0.524067	0.705185	NaN	920
CVXQP2_L	Disqualify	NaN	NaN	NaN	87467
CVXQP2_M	Runtime Failure	NaN	NaN	NaN	8717
CVXQP2_S	Success	0.212758	0.232959	0.863944	846
CVXQP3_L	Disqualify	NaN	NaN	NaN	102465
CVXQP3_M	Runtime Failure	NaN	NaN	NaN	10215
CVXQP3_S	Partial Iters Failure	0.33185	0.328833	NaN	994
DPKLO1	Runtime Failure	NaN	NaN	NaN	1785
DTOC3	Disqualify	NaN	NaN	NaN	64989
DUAL1	Success	0.120525	0.120078	0.193194	7201
DUAL2	Success	0.094266	0.096617	1.337312	9112
DUAL3	Success	0.137591	0.159495	0.313002	12327
DUAL4	Success	0.059957	0.069214	0.095549	5673
DUALC1	Success	0.161267	1.702546	1.816707	2025
DUALC2	Success	0.195361	0.344942	0.410105	1659
DUALC5	Success	0.418732	0.489906	0.728619	2296
DUALC8	Runtime Failure	NaN	NaN	NaN	4096
GENHS28	Success	0.020143	0.006074	0.005322	62
GOULDQP2	Runtime Failure	NaN	NaN	NaN	2791
GOULDQP3	Runtime Failure	NaN	NaN	NaN	3838
HS118	Iters Failure	NaN	NaN	NaN	69
HS21	Success	0.008002	0.000776	0.000804	6
HS268	Success	0.005888	0.002913	0.002825	55
HS35	Success	0.001905	0.001291	0.001174	13

continued next page

QP Name	Status	Time (s)			N
		1.00E-03	1.00E-05	1.00E-08	
HS35MOD	Success	0.002251	0.001226	0.001134	13
HS51	Success	0.00453	0.002427	0.002345	21
HS52	Success	0.003092	0.003096	0.002663	21
HS53	Success	0.002753	0.00274	0.002383	21
HS76	Success	0.003704	0.001945	0.002351	22
HUES-MOD	Disqualify	NaN	NaN	NaN	40000
HUESTIS	Disqualify	NaN	NaN	NaN	40000
KSIP	Partial Iters Failure	417.733868	NaN	NaN	19938
LASER	Partial RT. Failure	172.320382	228.696863	NaN	9462
LISWET1	Disqualify	NaN	NaN	NaN	50004
LISWET10	Disqualify	NaN	NaN	NaN	50004
LISWET11	Disqualify	NaN	NaN	NaN	50004
LISWET12	Disqualify	NaN	NaN	NaN	50004
LISWET2	Disqualify	NaN	NaN	NaN	50004
LISWET3	Disqualify	NaN	NaN	NaN	50004
LISWET4	Disqualify	NaN	NaN	NaN	50004
LISWET5	Disqualify	NaN	NaN	NaN	50004
LISWET6	Disqualify	NaN	NaN	NaN	50004
LISWET7	Disqualify	NaN	NaN	NaN	50004
LISWET8	Disqualify	NaN	NaN	NaN	50004
LISWET9	Disqualify	NaN	NaN	NaN	50004
LOTSCHD	Runtime Failure	NaN	NaN	NaN	72
MOSARQP1	Runtime Failure	NaN	NaN	NaN	8512
MOSARQP2	Success	89.007624	118.12944	235.297015	4820
POWELL20	Disqualify	NaN	NaN	NaN	40000
PRIMAL1	Success	1.382511	3.260168	4.159359	6464
PRIMAL2	Success	6.526522	6.75696	12.750759	9339
PRIMAL3	Success	9.601745	28.185454	38.091359	23036
PRIMAL4	Success	60.963329	120.158797	180.058542	19008
PRIMALC1	Partial Iters Failure	45.353647	NaN	NaN	2529
PRIMALC2	Iters Failure	NaN	NaN	NaN	2078
PRIMALC5	Iters Failure	NaN	NaN	NaN	2869
PRIMALC8	Iters Failure	NaN	NaN	NaN	5199
Q25FV47	Runtime Failure	NaN	NaN	NaN	130523
QADLITTL	Iters Failure	NaN	NaN	NaN	637
QAFIRO	Iters Failure	NaN	NaN	NaN	124
QBANDM	Runtime Failure	NaN	NaN	NaN	3023
QBEACONF	Iters Failure	NaN	NaN	NaN	3673

continued next page

QP Name	Status	Time (s)			N
		1.00E-03	1.00E-05	1.00E-08	
QBORE3D	Iters Failure	NaN	NaN	NaN	1872
QBRANDY	Iters Failure	NaN	NaN	NaN	2511
QCAPRI	Iters Failure	NaN	NaN	NaN	3852
QE226	Iters Failure	NaN	NaN	NaN	4721
QETAMACR	Runtime Failure	NaN	NaN	NaN	11613
QFFFFF80	Runtime Failure	NaN	NaN	NaN	10635
QFORPLAN	Iters Failure	NaN	NaN	NaN	6112
QGFRDXPN	Runtime Failure	NaN	NaN	NaN	3739
QGROW15	Runtime Failure	NaN	NaN	NaN	7227
QGROW22	Runtime Failure	NaN	NaN	NaN	10837
QGROW7	Iters Failure	NaN	NaN	NaN	3597
QISRAEL	Iters Failure	NaN	NaN	NaN	3765
QPCBLEND	Partial Iters Failure	0.257818	0.829277	NaN	657
QPCBOEI1	Iters Failure	NaN	NaN	NaN	4253
QPCBOEI2	Iters Failure	NaN	NaN	NaN	1482
QPCSTAIR	Partial RT. Failure	NaN	NaN	NaN	4790
QPILOTNO	Runtime Failure	NaN	NaN	NaN	16105
QPTEST	Success	0.027691	0.002254	0.000997	10
QRECIPE	Iters Failure	NaN	NaN	NaN	923
QSC205	Iters Failure	NaN	NaN	NaN	785
QSCAGR25	Runtime Failure	NaN	NaN	NaN	2282
QSCAGR7	Iters Failure	NaN	NaN	NaN	602
QSCFXM1	Runtime Failure	NaN	NaN	NaN	4456
QSCFXM2	Runtime Failure	NaN	NaN	NaN	8285
QSCFXM3	Runtime Failure	NaN	NaN	NaN	11501
QSCORPIO	Iters Failure	NaN	NaN	NaN	1842
QSCRS8	Runtime Failure	NaN	NaN	NaN	4560
QSCSD1	Runtime Failure	NaN	NaN	NaN	4584
QSCSD6	Runtime Failure	NaN	NaN	NaN	8378
QSCSD8	Runtime Failure	NaN	NaN	NaN	16214
QSCTAP1	Iters Failure	NaN	NaN	NaN	2442
QSCTAP2	Runtime Failure	NaN	NaN	NaN	10007
QSCTAP3	Runtime Failure	NaN	NaN	NaN	13262
QSEBA	Runtime Failure	NaN	NaN	NaN	6576

continued next page

QP Name	Status	Time (s)			N
		1.00E-03	1.00E-05	1.00E-08	
QSHARE1B	Iters Failure	NaN	NaN	NaN	1436
QSHARE2B	Iters Failure	NaN	NaN	NaN	873
QSHELL	Runtime Failure	NaN	NaN	NaN	74506
QSHIP04L	Runtime Failure	NaN	NaN	NaN	8548
QSHIP04S	Runtime Failure	NaN	NaN	NaN	5908
QSHIP08L	Runtime Failure	NaN	NaN	NaN	86075
QSHIP08S	Runtime Failure	NaN	NaN	NaN	32317
QSHIP12L	Runtime Failure	NaN	NaN	NaN	144030
QSHIP12S	Runtime Failure	NaN	NaN	NaN	44705
QSIERRA	Runtime Failure	NaN	NaN	NaN	9582
QSTAIR	Runtime Failure	NaN	NaN	NaN	6293
QSTANDAT	Runtime Failure	NaN	NaN	NaN	5576
S268	Success	0.015469	0.003815	0.002779	55
STADAT1	Runtime Failure	NaN	NaN	NaN	13998
STADAT2	Runtime Failure	NaN	NaN	NaN	13998
STADAT3	Disqualify	NaN	NaN	NaN	27998
STCQP1	Runtime Failure	NaN	NaN	NaN	66544
STCQP2	Runtime Failure	NaN	NaN	NaN	66544
TAME	Success	0.026913	0.002821	0.000756	8
UBH1	Disqualify	NaN	NaN	NaN	72012
VALUES	Success	0.667607	1.054153	6.843004	7846
YAO	Runtime Failure	NaN	NaN	NaN	10004
ZECEVIC2	Success	0.118319	0.002732	0.001439	7

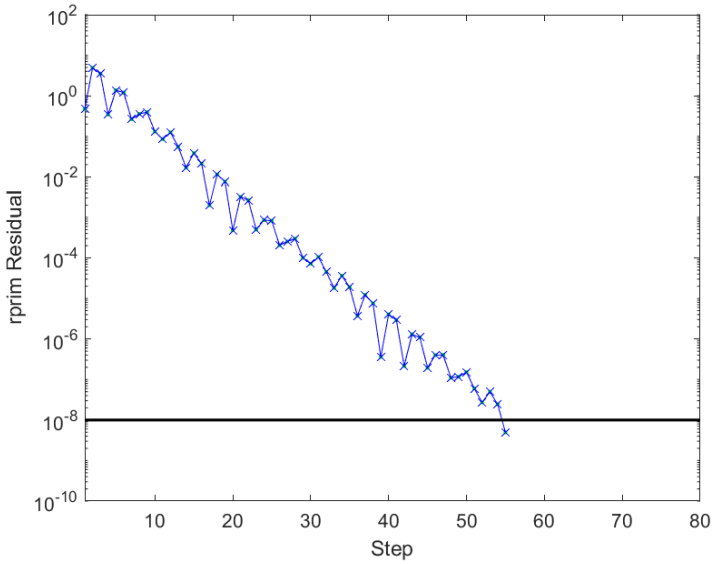

Figure 6. rprim Residual Graph for 4x2 Sample QP - max_{iters} = 80 on OSQP_BLAS