Quasilinear Chebyshev-Picard Iteration Method for Indirect Trajectory Optimization

Thomas Antony* and Michael J. Grant†
Purdue University, West Lafayette, IN 47907

Trajectory optimization plays a key role in conceptual design and control of complex dynamical systems. There are two main classes of methods that are used for solving trajectory optimization problems, namely, direct and indirect methods. Direct methods transcribe these problems into large parameter optimization problems that are then solved using an optimization method such as Sequential Quadratic Programming (SQP). This in contrast to indirect methods that use optimal control theory based on calculus of variations and convert the optimization problem into a nonlinear boundary value problem.

The Quasilinear Chebyshev Picard Iteration (QCPI) method builds on prior work and uses Chebyshev Polynomial series and the Picard Iteration combined with the Modified Quasilinearization Algorithm. The method is developed specifically to utilize parallel computational resources for solving large Two-Point Boundary Value Problems (TPBVP). The capabilities of the numerical method are validated by solving some representative nonlinear optimal control problems. The results demonstrate that QCPI is capable of leveraging parallel computing architectures and shows great potential for exploiting highly parallel architectures such as GPUs.

I. Introduction

The use of emerging parallel computational architectures is one way to accelerate numerical solution of large boundary value problems. There have been different approaches for parallelizing the numerical methods underlying indirect [1], [2] as well as direct methods [3], with varying degrees of success. Past work[1] showed that while it is indeed possible to accelerate the numerical methods used for solving BVPs associated with indirect methods, there are challenges once the problems get larger. This points to a need to develop a BVP solver that is inherently parallel and can efficiently exploit parallel computational resources for solving large-dimensional problems.

Graphics processing units (GPUs) were originally designed to be used as dedicated processors for rendering three-dimensional graphics on computers. Therefore GPUs are specialized in efficiently running compute-intensive, highly parallel operations, especially matrix operations that are required for rendering 3D graphics. Conventional computer processors were designed with more transistors dedicated to data caching and flow control, leading to very small latencies as opposed to high throughput. GPUs, in contrast have slower memory access and allows parallel execution of thousands of threads of execution, with some limitations. Hence, a GPU is especially suited to problems which can be expressed as large numbers of data-parallel computations [4], with a high ratio of arithmetic operations to memory operations. These operations should ideally be independent of each other and require very little cross-communication.

In a prior work, a highly parallel indirect optimization strategy for the rapid design of optimal trajectories was developed[1]. The multiple shooting method was used to develop this custom algorithm, *bvpgpu*, that ran very efficiently on a GPU. It was demonstrated that indirect optimization methods can be used to rapidly solve complex optimization problems by utilizing this GPU-accelerated multiple shooting method as shown by the benchmarks in Fig. 1. The benchmarks involved solving maximum terminal energy trajectories for a hypersonic vehicle with varying combinations of initial and terminal point constraints as well as path constraints. The test problem in this case was relatively small in terms of number of dimensions (6-24 states). These benchmarks showed a speedup of 2x-4x by using a GPU-based shooting method instead of *bvp4c*.

However, the multiple shooting method is still not "parallel enough" to scale well as the BVPs become large. This is the case when solving multi-vehicle problems or when solving the problem has multiple path constraints[5]. This challenge arises partly because the method was not originally formulated with the express purpose of utilizing parallel

*Autonomous Vehicle Engineer, Smart Ag, 2710 South Loop Drive, Ames, IA 50010, AIAA Member.

†Adjunct Assistant Professor, School of Aeronautics and Astronautics, 701 West Stadium Ave., Purdue University, West Lafayette, IN 47907, AIAA Senior Member

computational architectures. In order to obtain the 2x-4x speed-up seen in the benchmarks, the multiple shooting method had to be reformulated to make it more parallel in nature. This motivates the need for developing a numerical method that is inherently parallel and is designed specifically to exploit parallel computing architectures. This work describes the design of a new, scalable, highly parallel numerical method which advances the state-of-the-art for solving large nonlinear boundary value problems.

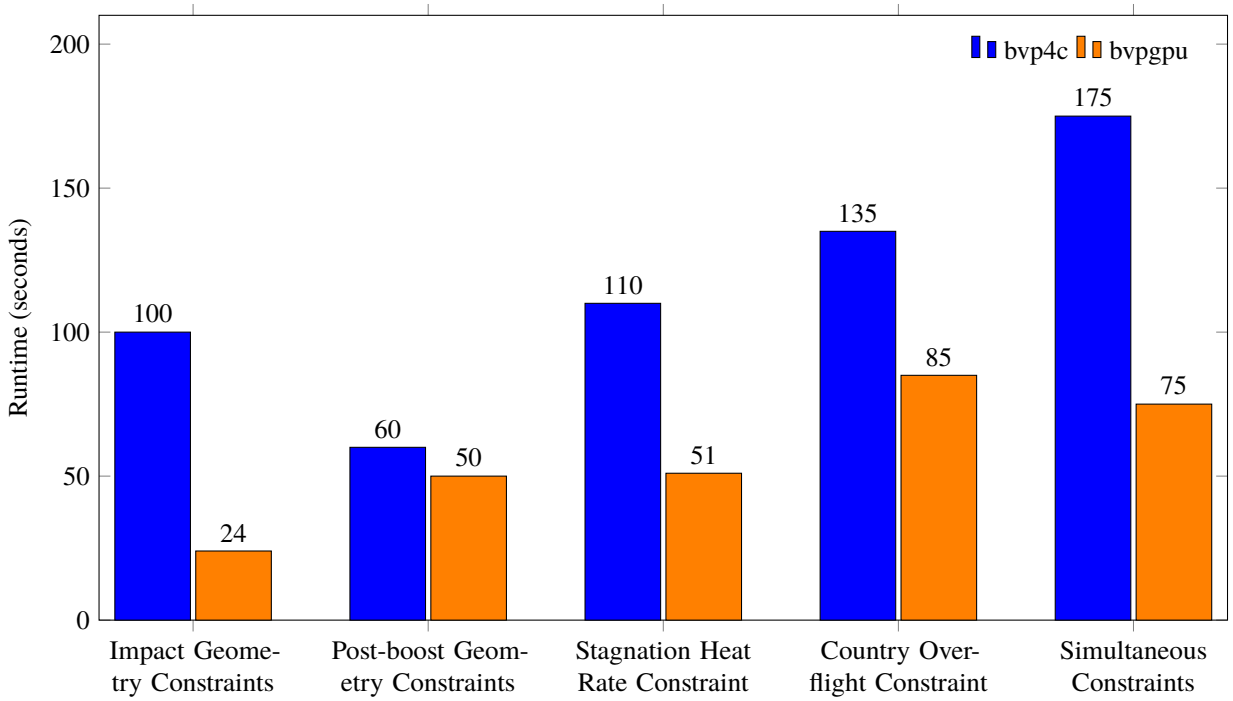


Fig. 1 Benchmarks – *bvp4c* vs. *bvpgpu* [1]

One of the key requirements for creating a fast numerical solver is to leverage modern computing architectures which are trending towards highly parallel systems in place of large monolithic processors[6–8]. Numerical methods designed with characteristics capable of exploiting these parallel processors will be faster than those which only utilize a single processor [1, 2, 9]. The goal of this work is to design a numerical method for solving boundary value problems that can exploit these architectures, thereby enabling indirect methods to rapidly solve more complex, highly constrained trajectory optimization problems.

II. Background

A. Picard Iteration

The starting point for the numerical method developed in this paper is the Picard iteration, also known as the Picard–Lindelöf theorem[10]. It is a method that was originally used to prove the existence and uniqueness of solutions to first-order differential equations with a given set of initial conditions.

$$\begin{aligned} y' &= f(t, y(t)), \;\; y(t_0) = y_0 \\ \phi_0 &= y_0 \\ \phi_{k+1} &= y_0 + \int_{t_0}^{t} f(s, \phi_k(s))ds, \text{ for iteration } k \end{aligned} \tag{1}$$

The Picard-Lindelöf Theorem shows that this series summation (ϕ_n) in Eq. (1) converges to $y(t)$ at the limit [10]. An example of the application of this theorem for a simple first-order initial value problem (IVP), $y' = f(t, y(t)) =$

$-y; y(t_0) = 1.0$, is shown in Eq. (2).

$$y' = f(t, y(t)) = -y \tag{2a}$$
$$\phi_0 = y_0 = 1.0 \tag{2b}$$
$$\phi_1 = y_0 + \int_{t_0}^{t} -1 \, ds = 1 - t \tag{2c}$$
$$\phi_2 = y_0 + \int_{t_0}^{t} (-1 + s) \, ds = 1 - t + \frac{t^2}{2} \tag{2d}$$
$$\phi_3 = y_0 + \int_{t_0}^{t} (-1 + s) \, ds = 1 - t + \frac{t^2}{2} - \frac{3t^3}{6} \tag{2e}$$

It can be seen that this series converges to the analytical solution of the system at the limit as: $y(t) = 1 - t + \frac{t^2}{2} - \frac{3t^3}{6} + \ldots = \exp(-y)$. This iteration forms the core of the Modified Chebyshev Picard Iteration algorithm.

B. Chebyshev Polynomials

Chebyshev polynomials[11] are a complete set of orthogonal polynomials that are commonly used for function approximation. There are two kinds of Chebyshev polynomials. For convenience, Chebyshev polynomials of the first kind are referred to as simply Chebyshev polynomials in this work. These polynomials are defined through a recurrence relation:

$$T_0(x) = 1 \tag{3}$$
$$T_1(x) = x \tag{4}$$
$$T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x) \tag{5}$$

where T_k represents the k-th order Chebyshev polynomial. They may also be computed using a trigonometric relation, $T_k(x) = \cos(k \arccos x)$ where $x \in [-1, 1]$. Chebyshev polynomials $T_k(x)$ up to $k = 5$ are shown in Figure 2.

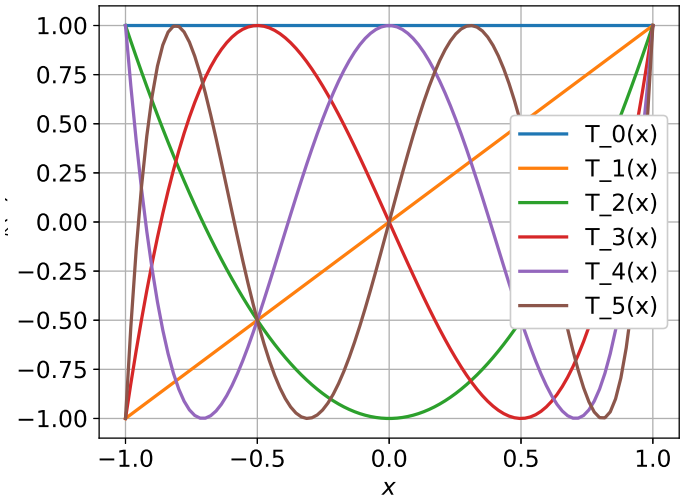


Fig. 2 Chebyshev Polynomials up to $k = 5$

The zeros of these polynomials are called Chebyshev-Gauss-Lobatto (CGL) nodes. The N+1 CGL nodes for an Nth order Chebyshev Polynomial can be calculated as:

$$x_k = \cos\left(\frac{k\pi}{n}\right), \qquad k = 0, 1, 2, ..., \ N+1 \tag{6}$$

When these nodes are used for polynomial interpolation, Runge's phenomenon is minimized, and the best function approximation under the minimax norm can be obtained [12, 13]. Ref. 12 shows that if a smooth function $f(\tau)$ is approximated by an N-th order Chebyshev polynomials as $f(\tau) \approx \sum_{k=0}^{''N} \alpha_k T_k(\tau)$, the coefficients α_k can be computed as:

$$\alpha_k = \frac{2}{N} \sum_{j=0}^{N}{}'' f(\tau_j) T_k(\tau_j), \qquad k = 0, 1, ..., N \tag{7}$$

The $''$ in the summation denotes that the first and the last terms in the summation are to be halved. The integral of Chebyshev polynomials is defined by:

$$\int T_k(x) = \frac{1}{2}\left(\frac{T_{k+1}}{k+1} - \frac{T_{k-1}}{k-1}\right) \tag{8}$$

The relation in Eq. (8) forms the basis for numerical methods that use Chebyshev polynomials to solve differential equations. There are many past works describing such methods that solve initial value problems and boundary value problems[14–20]. There are also some direct methods that use Chebyshev polynomials for solving optimal control problems [21–23]. Some of these methods that combine the Picard Iteration with Chebyshev polynomials to solve IVPs and BVPs are examined in the Section II.C.

C. Chebyshev-Picard Methods

Clenshaw's work in Ref. 15 is one of the first in literature that uses the Picard iteration combined with Chebyshev polynomials to create a practical numerical method for solving IVPs and BVPs. This method was later applied to several astrodynamics problems involving interplanetary trajectories by Feagin[24]. The suitability of this Chebyshev-Picard method for efficient implementation on parallel processors was examined by Shaver[25] by using it to create a parallel orbit propagation algorithm and a parallel orbit estimation algorithm. A vector-matrix formulation of the same algorithm was designed by Feagin in Ref. 26 but no experimental results were shown. This was the precursor to the method used in MCPI which forms part of the numerical method developed in this paper. Ref. 27 shows an implementation of the Chebyshev-Picard method on a vector computer. However, this implementation was in some cases slower than the scalar version of the code owing to some overheads and inefficiencies. The Modified Chebyshev Picard Iteration algorithm[9] built on these existing works and created a unified matrix-vector method for solving both IVPs and certain classes of BVPs. MCPI was shown to be capable of solving several important celestial mechanics problems. The original work also showed techniques for improving the convergence domain of Chebyshev-Picard methods.

1. Modified Chebyshev-Picard Iteration Method

The Modified Chebyshev Picard Iteration (MCPI) method is a numerical method for solving Initial Value Problems (IVPs) and certain classes of Boundary Value Problems (BVPs) without directly propagating the equations of motion or evaluating gradients[9]. MCPI is based on the Picard iteration method described in Section II.A. The algorithm represents the integrand in Eq. (1) as a weighted sum of Chebyshev polynomials of sufficiently high order. The integration step of Picard iteration is performed using the quadrature rule in Eq. (8). The coefficients of the Chebyshev polynomials representing the solution is solved for, based on the boundary conditions of the problem.

While this iteration can also be implemented using other orthogonal polynomial sets such as Legendre polynomials, this particular formulation of Chebyshev polynomials was chosen because it is possible to fit a function to these polynomials without solving a linear algebra problem[12]. As shown before in Eq. (7), the calculation of polynomial coefficients on a Chebyshev mesh for a given function is a long summation operation which can be reformulated into a simple matrix multiplication operation[9].

The MCPI algorithms for initial value problems (MCPI-IVP) and boundary value problems (MCPI-BVP) are very similar in their implementation. The simpler of the two, MCPI-IVP, using Chebyshev polynomial series of order N, is summarized below.

For a given dynamic system,

$$\dot{\mathbf{y}} = \Phi(\mathbf{y}, t) \qquad \mathbf{y}(0) = \mathbf{y}_0 \tag{9}$$

a scaled version, ϕ, is formulated which can be evaluated between -1 and 1.

$$\phi(\mathbf{y}, t) = \frac{T}{2}\Phi(\mathbf{y}, \frac{T}{2}\tau + \frac{T}{2}) \tag{10}$$

The solution is evaluated over Chebyshev-Gauss-Lobatto (CGL) mesh points of a given order, N, that are defined by:

$$\tau_j = \cos(j\pi/N), \qquad j = 0, 1, ..., N \tag{11}$$

The main steps in the algorithm are as follows. First, the dynamic equations are evaluated over the CGL mesh, and the coefficients for the Chebyshev polynomials, F_k, corresponding to these equations are calculated.

$$F_k = \frac{2}{N}\sum_{j=0}^{N} \phi(t, \mathbf{y}_k)\, T_k(\tau_j) \tag{12}$$

Each coefficient F_k is obtained through the summation of $N + 1$ terms, each involving the product of the scaled dynamic equations ϕ and the Chebyshev polynomial T_k evaluated at node τ_j. By applying the Picard iteration and the integration rule from Eq. (8) to Eq. (12), the dynamic equations are integrated to obtain the coefficients, β_k $(k = 0, 1, 2..., N)$, corresponding to the solution.

$$\beta_r = \frac{1}{2r}(F_{r-1} - F_{r+1}), \qquad r = 1, 2, ..., N-1 \tag{13}$$

$$\beta_N = \frac{F_{N-1}}{N} \tag{14}$$

$$\beta_0 = 2y_0 + \sum_{j=1}^{N}(-1)^{j+1}\beta_j \tag{15}$$

$$\mathbf{y}_{k+1}(\tau) = \sum_{j=0}^{N}\beta_j T_j(\tau) \tag{16}$$

The algorithm starts with an initial guess for the entire solution expressed on a Chebyshev-Gauss-Lobatto grid and continues until the change in $\mathbf{y}_k$ is less than a desired tolerance. The original author showed that the operations involved in calculating β_k, as well as computing the solution from β_k in each iteration can be expressed as a series of matrix-vector operations [9]. The steps in Eqs. (12)-(16) then condense to the following form:

$$\vec{\beta} = C_\alpha \phi(\mathbf{y}_k, \tau) + \mathbf{y}_0 \tag{17}$$

$$\mathbf{y}_{k+1} = C_x \vec{\beta} \tag{18}$$

C_x and C_α are constant matrices for a given order of Chebyshev polynomials. The overall structure of these matrices are given in the Appendix. These matrices can be pre-computed and cached before the iteration process begins, and hence the "integration" consists entirely of matrix-vector multiplication operations. Such operations are ideal for parallel implementation such as on a GPU or multi-core CPUs for accelerated processing. A GPU implementation of MCPI-IVP is shown in Ref. 28, where it was used for high-precision parallel orbit propagation.

The BVP version of the algorithm specifies a different update equation for each state depending on whether it is constrained at the initial point, terminal point, or both[29]. If both the initial and terminal values are given for a state, the update equations for β are changed as shown below:

$$\beta_r = \frac{1}{2r}(F_{r-1} - F_{r+1}), r = 2, 3, ..., N-1 \tag{19a}$$

$$\beta_N = \frac{F_{N-1}}{N} \tag{19b}$$

$$\beta_0 = y_0 + y_f - 2(\beta_2 + \beta_4 + \beta_6 + ...) \tag{19c}$$

$$\beta_1 = \frac{y_f - y_0}{2} - (\beta_3 + \beta_5 + \beta_7 + ...) \tag{19d}$$

An astrodynamics trajectory problem was solved in the original work using this method, and its performance and solution quality were compared to that of a direct pseudospectral method. Significant speedups were obtained over direct methods, and it was also shown that the method can derive huge benefits from implementation on GPU computing architectures.

However, there is a significant drawback when it comes to using this algorithm for solving BVPs arising in trajectory optimization problems. The MCPI-BVP formulation assumes that every state in the problem has at least one boundary condition defined for it. If the BVP does not define a boundary condition for a particular state, a boundary condition has to be derived for it from other domain-specific information available in the problem, if any. This is only possible for a limited class of problems such as the astrodynamics problems demonstrated in the original implementation[9]. The algorithm also assumes that the boundary conditions are simple equality constraints at the initial and terminal points. This was the motivation for the development of a more generalized MCPI-BVP algorithm that is capable of handling general nonlinear boundary conditions such as those encountered in optimal control problems.

D. A Generalized MCPI-BVP Algorithm

A more generalized formulation of the MCPI-BVP method is required for solving nonlinear BVPs arising in optimal control problems. Initially, a version of MCPI was created which linearized the boundary conditions and tried to compute for the polynomial coefficients by solving a linear system[30]. The original MCPI algorithm was formulated by assuming a set of fixed boundary conditions of the form shown in Eq. (20) and solving a linear system analytically to obtain the expressions for the Chebyshev coefficients β_k.

$$\vec{y}(t_0) = \vec{y}_0, \quad \vec{y}(t_f) = \vec{y}_f \tag{20}$$

In order to formulate a more generalized version of this method, it is necessary to start with a more generic boundary condition function such as the one in Eq. (21) for a two-point boundary value problem.

$$\begin{aligned} &\vec{b}\left(\vec{y}(t_0), \vec{y}(t_f)\right) = 0 \\ &\text{Subject to: } \frac{d\vec{y}}{dt} = \vec{f}(t, y(t)) \end{aligned} \tag{21}$$

The boundary conditions are linearized in Eq. (22) and combined with the expressions for initial and terminal states, Eq. (23), in order to obtain the Chebyshev coefficients (β_k) of the solution as shown in Eq. (24).

$$\begin{aligned} \vec{b} \approx \quad & \mathcal{M} \times (\vec{y}(t_0) - \vec{y}_0) + \mathcal{N} \times (\vec{y}(t_f) - \vec{y}_f) + b(\vec{y}_0, \vec{y}_f) \\ \text{where } \mathcal{M} = & \frac{\partial \vec{b}}{\partial \vec{y}_0}, \ \mathcal{N} = \frac{\partial \vec{b}}{\partial \vec{y}_f} \end{aligned} \tag{22}$$

$$\begin{aligned} x(t_0) = & \frac{\beta_0}{2} - \beta_1 + \sum_{k=2}^{N} (-1)^{k+1} \beta_k \\ x(t_f) = & \frac{\beta_0}{2} + \beta_1 + \sum_{k=2}^{N} \beta_k \end{aligned} \tag{23}$$

$$\begin{aligned} \vec{x}(\tau) \approx & \sum_{k=0}^{N} \overrightarrow{\beta}_k T_k(\tau), \quad \text{where} \\ \beta_r = & \frac{1}{2r}(F_{r-1} + F_{r+1}), \ r = 1, 2, ..., N-1 \\ \beta_N = & \frac{F_{N-1}}{2N} \\ \left[\frac{\mathcal{M}+\mathcal{N}}{2}, \mathcal{N}-\mathcal{M}\right] \begin{pmatrix} \overrightarrow{\beta_0} \\ \overrightarrow{\beta_1} \end{pmatrix} = & \Big(\mathcal{M} \times (\vec{y}_0 - \sum_{k=2}^{N} (-1)^{k+1} \beta_k) \\ & + \mathcal{N} \times (\vec{y}_f - \sum_{k=2}^{N} \beta_k) - \vec{b}(\vec{y}_0, \vec{y}_f) \Big) \end{aligned} \tag{24}$$

Eq. (24) outlines one way to incorporate non-linear boundary conditions into the MCPI algorithm. In Ref. 30, this algorithm was demonstrated using the Brachistochrone problem. However, it was found that it was not capable of solving problems with more numerical sensitivity such as hypersonic optimal control problems. This prompted the search for a different approach to solving boundary value problems that complements the drawbacks of MCPI-BVP and can be combined with MCPI to create a more general numerical method.

E. Modified Quasi-Linearization Algorithm

The method of particular solutions for solving linear two-point boundary value problems is described by Miele in Ref. 31. The boundary-value problem is solved by linearly combining several particular solutions of the original differential system. This method was further expanded to include some classes of nonlinear problems in Ref. 32 with nonlinear dynamic equations. The modified quasi-linearization algorithm (MQA)[33, 34] is a further refinement of the method of particular solutions that allows for nonlinear boundary conditions at the terminal point. Ref. 35 explores the use of MQA for solving optimal control problems. In the modified quasi-linearization algorithm, the known initial conditions and guesses for the unknown initial states are used to generate a reference solution using numerical integration. Then, small, linearly independent perturbations of the unknown states are also propagated using a numerical integrator. The resulting perturbations at the terminal point are used to compute corrections for the entire trajectory until all the boundary conditions are satisfied.

Consider a nonlinear dynamic system with n states as follows:

$$\dot{\mathbf{y}} = \phi(\mathbf{y}, t), \qquad 0 \leq t \leq t_f \tag{25}$$

with the initial conditions,

$$b_{0j}(\mathbf{y}(t_0)) = 0 \qquad j = 1, 2, ..., p \tag{26}$$

and terminal conditions,

$$b_{fj}(\mathbf{y}(t_f)) = 0 \qquad j = 1, 2, ..., q \tag{27}$$

Taking a first order approximation of b_f,

$$\frac{\partial b_f(\mathbf{y}(t_f))}{\partial \mathbf{y}} \Delta\mathbf{y}(t_f) + b_f(\mathbf{y}(t_f)) = 0 \tag{28}$$

Let $\mathbf{A}_j(t)$ denote the perturbations from the reference solution for a small perturbation in a free initial state. $\mathbf{A}_j(t_f)$ is computed for $q + 1$ perturbed initial conditions to form the linear combination:

$$\mathbf{A}(t) = \sum_{j=1}^{q+1} k_j \mathbf{A}_j(t) \tag{29}$$

Ref. 32 shows that this linear combination satisfies the system in Eq. (28). Therefore the coefficients k_j can be computed by solving the following linear system:

$$\sum_{j=1}^{q+1} k_j = 1 \qquad \psi_{\mathbf{y}}(\mathbf{y}(t_f)) \sum_{j=1}^{q+1} k_j \mathbf{A}_j(t) + \psi(\mathbf{y}(t_f)) = 0 \tag{30}$$

The correction for the solution is computed as shown in Eq. (31). This correction is applied not just to the initial state, but to the entire trajectory. This feature makes it apt for inclusion into a method like MCPI where, unlike a shooting method, the entire trajectory is approximated at all times.

$$\Delta y(t) = \alpha \sum_{j=1}^{q+1} k_j \mathbf{A}_j(t) \quad \text{where } 0 \leq \alpha \leq 1 \tag{31}$$

The step-size, α can be determined by a one-dimensional line-search of the performance index, P, the cumulative error in the differential equations and the boundary conditions, defined in Eq. (32). Ref. 32 proves that the use of this

performance index gives the algorithm its *descent property*: If the step-size, α, is sufficiently small, the reduction in P is guaranteed. The search is started with $\alpha = 1.0$ and continues until $P(\alpha) < P(0)$.

$$P(\alpha) = \int_0^T (\dot{\mathbf{y}} - \phi)^T (\dot{\mathbf{y}} - \phi)\, dt + b_f^T b_f + b_0^T b_0 \tag{32}$$

A recent work [36] examined the use of the method of particular solutions (MPS)[32] along with MCPI for computing perturbed orbits of orbital debris by solving Lambert's problem. The current work is focused on expanding this to include the ability to solve optimal control problems using a hybrid method that uses both MCPI and MQA, called the Quasi-Linear Chebyshev-Picard Iteration (QCPI) algorithm.

III. QCPI Algorithm Implementation

The Quasi-Linear Chebyshev-Picard Iteration (QCPI) method leverages MCPI and the modified quasi-linearization algorithm to solve nonlinear two-point boundary value problems such as those arising in trajectory optimization. It incorporates MCPI as the IVP integrator and uses MQA to update the solution.

The algorithm is designed to solve general nonlinear two-point boundary value problems of the following form:

$$\dot{\mathbf{x}} = \vec{\phi}(t, \mathbf{x}) \tag{33a}$$

$$\boldsymbol{b_0}(\mathbf{x}(0), 0) = 0 \tag{33b}$$

$$\boldsymbol{b_f}(\mathbf{x}(T), T) = 0 \tag{33c}$$

The solution is approximated using a Chebyshev Polynomial series of order N, with separate coefficients for each state. The algorithm consists of the following steps.

1) Define the matrices C_a and C_x as well as the independent variable mesh $\tau \in [-1, 1]$ for the given value of N. The structure of C_a and C_x are detailed in the Appendix.
2) For the initial state, x^0, The perturbed initial states, x_p are initialized as:

$$A_j = \delta_{ij} \Delta x_i, \quad \text{for } i, j = 1, 2..., n \tag{34}$$

$$\mathbf{x}_p(0) = [\mathbf{x}^0 + A_0, \mathbf{x}^0 + A_1, ..., \mathbf{x}^0 + A_n] \tag{35}$$

where δ_{ij} is the Kronecker delta function, and n is the number of ODEs in the BVP. $\mathbf{x}_p$ is a row vector of size n^2.

3) The initial guess matrix, x_{guess}, is initialized. This is done either using the value from a previous iteration or by calling a separate MCPI-IVP integrator to propagate the equations of motion with the actual initial state, x^0, along with the perturbed states, $x_p(0)$, for the given value of N. This combined state vector will now be denoted as X and contains the original state vector followed by perturbed state vectors.

$$X(0) = \begin{bmatrix} \mathbf{x}(0) & \mathbf{x}(0) + A_0 & \mathbf{x}(0) + A_1 & \dots & \mathbf{x}(0) + A_n \end{bmatrix} \tag{36}$$

4) Evaluate the dynamic equations of the BVP at every point of the CGL mesh, for both the original and the perturbed initial conditions. The results are stored in the matrix $\mathbf{\Phi}$ which has the same dimension as x_{guess}.

$$\Phi = w_1 \begin{bmatrix} \vec{\phi}(\tau_0, \mathbf{x}(\tau_0)) \\ \vec{\phi}(\tau_1, \mathbf{x}(\tau_1)) \\ \vdots \\ \vec{\phi}(\tau_N, \mathbf{x}(\tau_N)) \end{bmatrix}_{n^2 \times (N+1)} \qquad \text{where } w_1 = T/2 \tag{37}$$

The above computation assumes that the function $\vec{\phi}(\tau, x)$ returns a row vector of length n.

5) The derivative information in Φ is fit to a Chebyshev polynomial series of order N. By using the Matrix-Vector form described by Feagin[9, 26] and Bai[9], the computation of the polynomial coefficients representing the solution, β, consists of a simple matrix multiplication operation:

$$\beta = 2\vec{\chi}_0 + C_a \times \Phi_i \quad \text{where } \chi_0 = \begin{bmatrix} X(0) \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}_{(N+1)\times 1} \tag{38}$$

6) The solution and the guess for the next iteration, x_{new}, is obtained as:

$$x_{new} = C_x \times \beta \tag{39}$$

7) The change in x is calculated as $e = x_{new} - x_{guess}$. If the L2-norm of e is greater that the required tolerance, skip to step 13.
8) Once e is under the required tolerance, the integration is complete, and x_{new} contains dynamically feasible unperturbed and perturbed trajectories. The perturbations due to the change in each state is used to update the solution to satisfy the nonlinear boundary conditions. The Jacobian matrices for the initial and terminal boundary conditions are calculated using finite-difference methods. The residual error in the boundary conditions are also evaluated.

$$\mathbf{r}_0 = \boldsymbol{b_0}(0, \mathbf{x}(0)) \tag{40}$$
$$\mathbf{r}_f = \boldsymbol{b_f}(T, \mathbf{x}(T)) \tag{41}$$
$$\mathbf{b}_{0x} = \frac{\partial b_0}{\partial \mathbf{x}}|_{\mathbf{x}=\mathbf{x}(0)} \tag{42}$$
$$\mathbf{b}_{fx} = \frac{\partial b_f}{\partial \mathbf{x}}|_{\mathbf{x}=\mathbf{x}(T)} \tag{43}$$

9) If the L2-norms of residual errors $\mathbf{r}_g$ and $\mathbf{r}_f$ are below the desired tolerance, the solution for the BVP has converged and the iteration process can be stopped.
10) The required correction for the solution to reduce the residual errors, is assumed to be a linear combination of all the initial perturbations in state, or $\Delta x = \sum_{j=0}^{n+1} K_j A_j(T)$, where $A_j(t)$ is the perturbation at t when starting with a perturbation in the j-th state. It is to be noted that A_0 is a zero-vector which is used to add an additional constraint on the coefficients, K_j, that $\sum_{j=0}^{n+1} K_j = 1$. The coefficients of this linear combination, $\mathbf{K}$, are calculated by solving the following linear system.

$$\begin{bmatrix} 1 & .... & 1 \\ \mathbf{b}_{0x} \times A_j(0) & & \\ \mathbf{b}_{fx} \times A_j(T) & & \end{bmatrix} \underbrace{\begin{bmatrix} K_1 \\ K_2 \\ \vdots \\ K_{n+1} \end{bmatrix}}_{\mathbf{K}} = \begin{bmatrix} 1 \\ -\mathbf{r}_0 \\ -\mathbf{r}_f \end{bmatrix} \tag{44}$$

11) The direction of the state-correction vector is given by $\Delta\mathbf{x}(t) = \sum_{j=0}^{n+1} K_j A_j(t)$. Ref. 31 shows that a sufficiently small step-size α applied to this direction vector will reduce the residual error, leading to convergence. α is found by performing a line-search on the performance index, $P(\alpha) = ||\boldsymbol{b_f}(x(T) + \alpha\Delta x(T))||_2 + ||\boldsymbol{b_0}(x(0) + \alpha\Delta x(0))||_2$, to find a value of α such that $P(\alpha) < P(0)$. This value is then selected as the step-size for the iteration.
12) The solution is updated using the correction vector as $\mathbf{x}(t) = x(t) + \alpha\Delta x(t)$.
13) x_{guess} is replaced with x_{new}. Repeat from Step 4 until convergence criteria is satisfied or maximum number of iterations exceeded.

The algorithm is summarized in the flowchart in Figure 3.

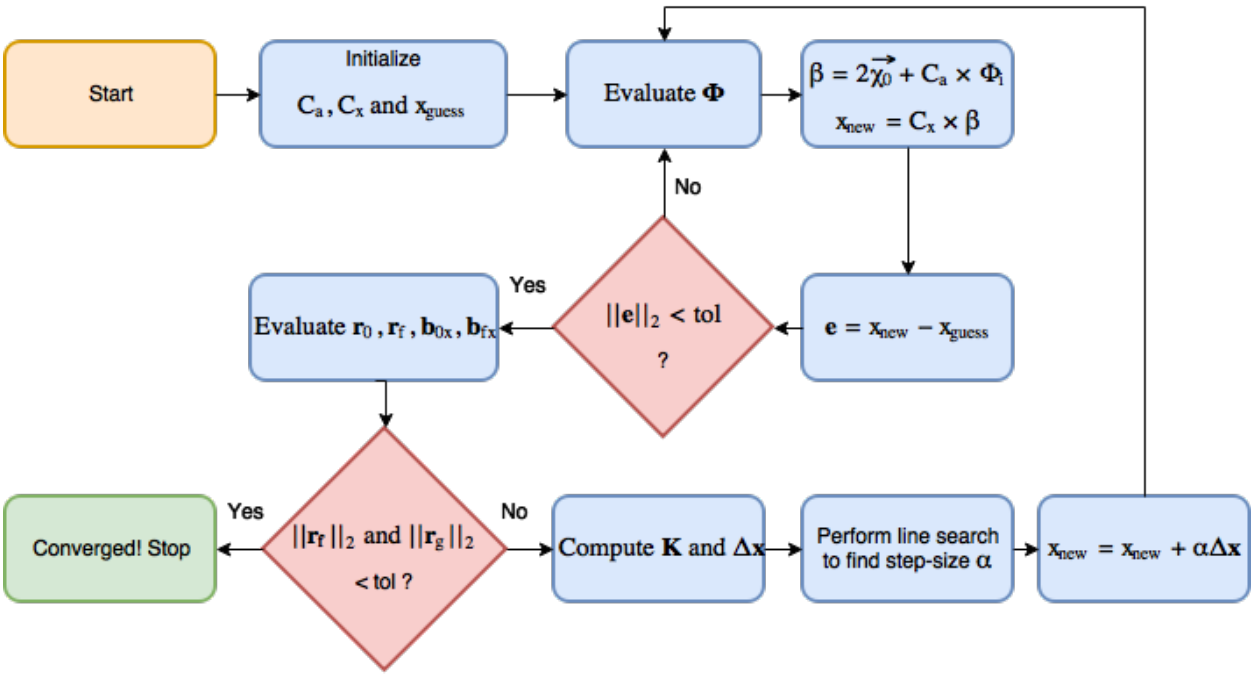


Fig. 3 QCPI Algorithm Implementation – Flowchart

IV. Acceleration using *Numba* Just-In-Time (JIT) Compiler

Parallelization of computational methods is generally a very time-consuming task that requires careful organization of data-parallel operations and creation of special data-structures required for exploiting parallel computation architecture. Prior work[1] explored in detail an efficient GPU implementation of the multiple shooting method in MATLAB. In contrast, QCPI is parallelized in a more automated manner using the JIT compiler *Numba*[37]. *Numba* helps speed up computation-heavy code written in the Python[38] programming language, by compiling it to high-performance native machine code with speeds comparable to C/FORTRAN without having to switch languages or Python interpreters. Numba is based on the LLVM (Low-Level Virtual Machine)[39] compiler which can inspect and analyze code on-the-fly and generate optimized native machine code. It is designed to work with multi-core CPUs or GPUs and can integrate directly with the Python scientific software stack such as NumPy[40] and SciPy[41].

Numba supports three different compiler modes:

- Python JIT mode which allows the use of Python data structures such as dictionaries and objects and is the slowest of all three. This option includes a compilation overhead the first time the code is executed.
- *nopython* JIT mode – this restricts the types of variables that can be included in a function. This mode can achieve performance close to C or FORTRAN native code. Since it is "just-in-time" compiled, there is an added overhead the first time the code is executed.
- Ahead-Of-Time mode – This compiles code into machine-specific binary ahead of time and can be used later without Numba.

For QCPI, the *nopython* JIT mode was used. Both of the JIT compilation modes also support automatic parallelization of certain types of loops, as long as the loop does not have cross-iteration dependencies. This is an extremely useful feature when calculating Jacobian matrices and when evaluating equations with multiple sets of perturbed states. Each individual iteration is run on separate CPUs in parallel at close to C/FORTRAN speeds by just adding some annotations to the Python code.

V. Validation

The QCPI solver is validated by testing it on some representative optimal control problems with known solutions. The results are compared to those obtained using a multiple shooting algorithm.

A. Classical Brachistochrone Problem

The classical Brachistochrone problem is the minimum-time problem described below in Eq. (45). It is used for validation as it is one of the simplest nonlinear optimal control problems with a known solution.

Problem Statement:

$$\text{Min } T \tag{45a}$$

$$\text{Subject to :}$$

$$\dot{x} = v\cos\theta \tag{45b}$$

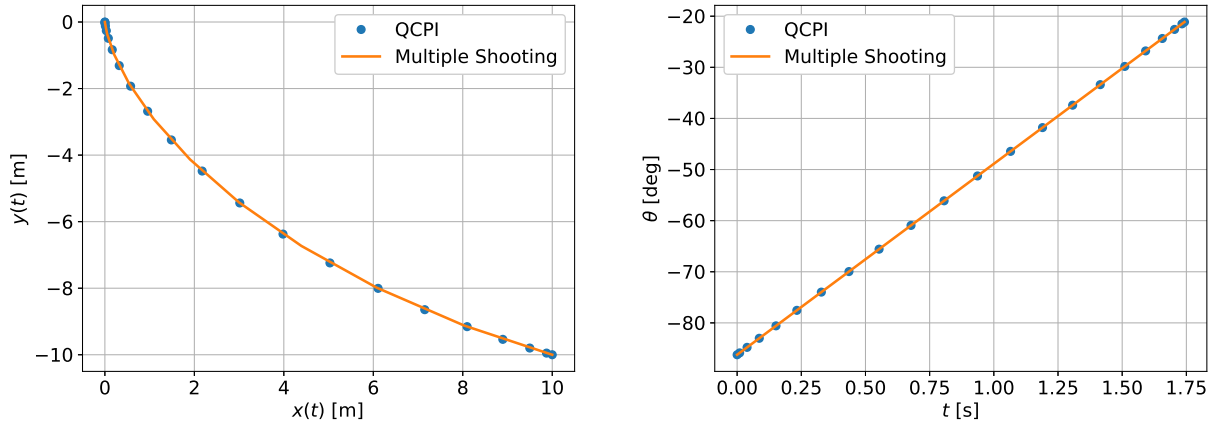
$$\dot{y} = v\sin\theta \tag{45c}$$

$$\dot{v} = g\sin\theta \tag{45d}$$

$$x(0) = y(0) = 0, x(T) = -y(T) = 10 \tag{45e}$$

$$g = -9.81 \tag{45f}$$

where θ is the control.

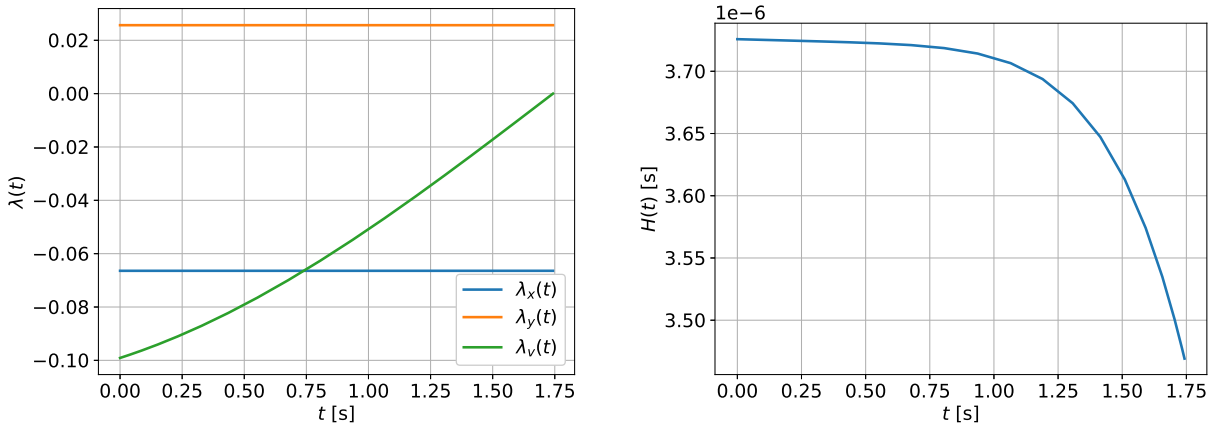


(a) Trajectory

(b) Control History

Fig. 4 QCPI Validation - Classical Brachistochrone Problem

The initial guess was created by propagating the equations of motion from the initial conditions with fixed initial values (=-0.1) for the costates for 0.1 seconds. The terminal conditions were updated to the desired value of $(x(T), y(T)) = (10, -10)$ over 11 continuation steps. The converged trajectory and control history are shown in Fig 4. The optimality of the solution is verified in Fig 5. The costate profiles in Figure 5(a) match the necessary conditions of optimality with λ_x and λ_y being constant, and $\lambda_v(T)$ being equal to zero as $v(T)$ is unconstrained. Fig 5(b) shows that the Hamiltonian remains very close to zero as it should for the optimal solution.

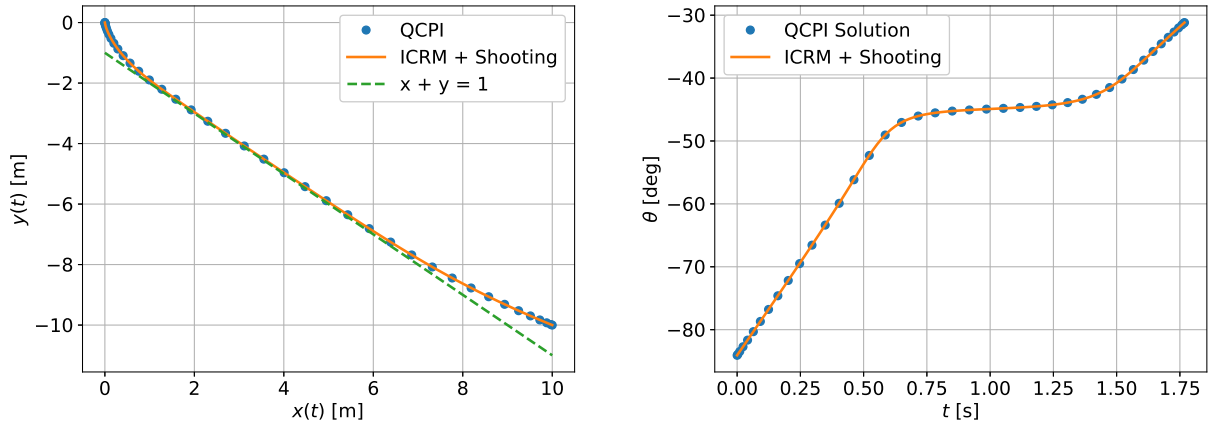


(a) Costates (b) Hamiltonian

Fig. 5 QCPI Validation - Classical Brachistochrone - Optimality Conditions

B. Constrained Brachistochrone Problem

QCPI is validated using a modified version of the classical Brachistochrone problem from the previous section with an added linear path constraint. The path constraint is regularized using the Integrated Control Regularization Method (ICRM) described in Ref. 5 and converted to a two-point boundary value problem. The regularization parameter, ϵ, which controls the fidelity of the path constraint when using ICRM, was set to 10^{-4}. The meaning of ϵ is explained in more detail in Ref. 5. The constraint is positioned such that the optimal trajectory is restricted from going as far down the y direction as it does in the unconstrained case. The optimal constrained trajectory follows the constraint where required and then goes back to a time-optimal path. The result is compared to that obtained using a shooting method solver in Figure 6 in order to validate it.



(a) Trajectory (b) Control History

Fig. 6 QCPI Validation - Constrained Brachistochrone Problem with $\epsilon = 10^{-4}$

C. Unconstrained Maximum Terminal Energy Hypersonic Trajectory

To demonstrate that QCPI can be applied to more complex aerospace problems, a scenario involving maximum terminal energy trajectories of a slender hypersonic vehicle is examined. The vehicle is assumed to be capable of angle-of-attack (AoA) control and having a peak L/D of around 2.4. The boundary conditions and the environment

parameters are listed in Tables 1 and 2 respectively.

Table 1 Boundary Conditions.

State	h	v	γ	θ
Staging (t=0)	80,000 m	4000 m/s	free	0 deg
Terminal (t=T)	15,000 m	free	free	5 deg

Table 2 Environment Parameters.

Parameter	Value
μ	$3.986 \times 10^{14}\,\mathrm{m^3\,s^{-2}}$
R_E	$6.3781 \times 10^{6}\,\mathrm{m}$
ρ_0	$1.2\,\mathrm{kg\,m^{-3}}$
h_s	$7500\,\mathrm{m}$

The problem is defined as follows:

$$\text{Max } v(T)^2 \tag{46a}$$

$$\text{Subject to :}$$

$$\dot{h} = v\sin\gamma \tag{46b}$$

$$\dot{\theta} = \frac{v\cos\gamma}{r} \tag{46c}$$

$$\dot{v} = \frac{-D}{m} - \frac{\mu\sin(\gamma)}{r^2} \tag{46d}$$

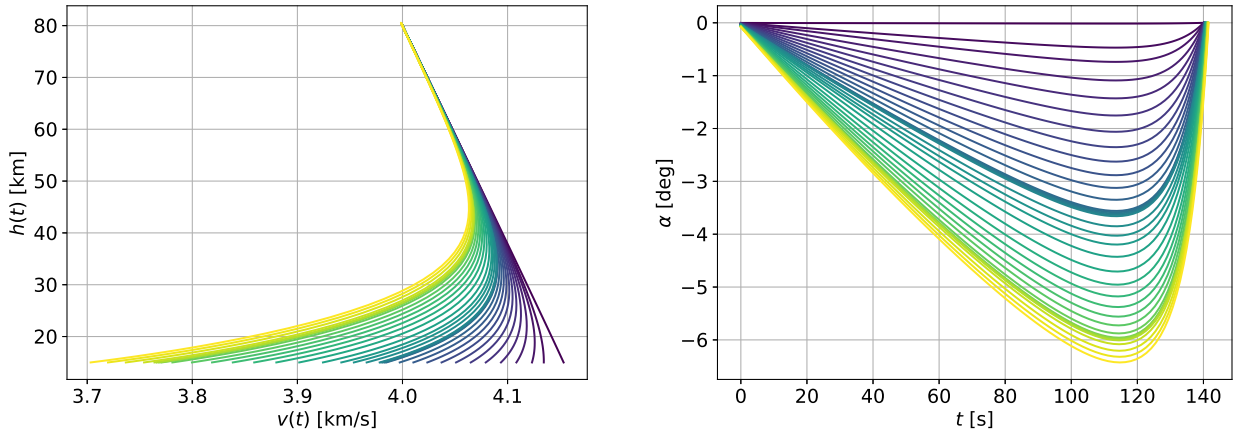
$$\dot{\gamma} = \frac{L}{mv} + \left(\frac{v}{r} - \frac{\mu}{vr^2}\right)\cos(\gamma) \tag{46e}$$

$$\begin{aligned} r &= R_E + h \\ D &= qC_D A_{ref} \\ L &= qC_L A_{ref} \\ q &= \frac{1}{2}\rho v^2 \\ \rho &= \rho_0 \exp(-h/h_s) \\ \\ C_L &= C_{L1}\alpha + C_{L0} \\ C_D &= C_{D2}\alpha^2 + C_{D1}\alpha + C_{D0} \end{aligned} \tag{46f}$$

The same problem was also solved using the shooting method. This is a particular example where QCPI in its current form performs worse than the shooting method even after parallelization. The reason for this is that in its current form, QCPI uses a CGL mesh of a fixed size and structure that is set before starting the solution process. The shooting method on the other hand, uses an adaptive numerical integrator method, Runge-Kutta-Fehlberg-45, which is able to adaptively select the mesh size based on the sensitivity of the dynamic equations. Due to this reason, regions of high numerical sensitivity towards the middle of the trajectory tends to cause the solver to diverge. Therefore, while both methods required the use of continuation, starting with a trivial initial guess, it is to be noted that the continuation strategy used for QCPI was different from that used for the shooting algorithm.

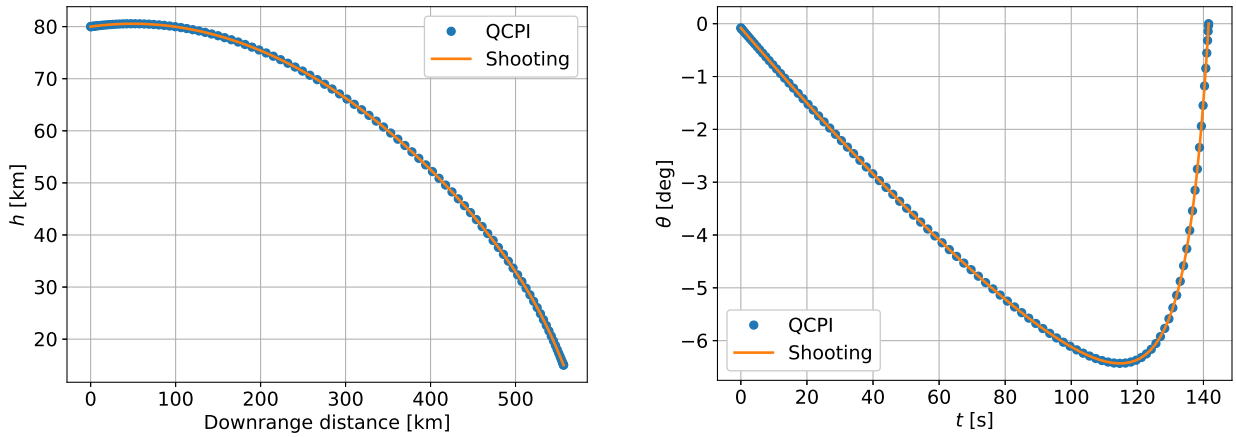
In case of the shooting algorithm, continuation was performed only on the boundary conditions on the states. However, for QCPI the problem initially had to be solved with the atmospheric density parameter, ρ_0, set to a very low value of 0.0012 kg/m^3 (0.1% of the actual value). Once the near-ballistic trajectory connecting the starting and ending points is solved, ρ_0 was increased up to its actual value of 1.2 kg/m^3. The majority of the time of the solution process is spent on changing ρ_0 to its actual value. This extra step is required due to the limitation of QCPI when dealing with problems with high-sensitivity regions near the middle of the trajectory as described in Section VI. By solving the trajectory first with very low atmospheric density, it is possible to avoid intermediate trajectories with high sensitivity regions. It is in fact, possible to solve such problems with the current implementation of QCPI if the number of nodes are significantly increased, at the cost of very high computation time.

The evolution of the trajectory and control history with changing ρ_0 is shown in Figure 7. The control effort increases as the atmospheric density increases so that the vehicle can utilize lift to fly higher. There is also a significant decrease in the terminal velocity due to atmospheric drag.



(a) Energy Plot (b) Control History

Fig. 7 QCPI Validation - Unconstrained Hypersonic Trajectory Problem - Continuation in ρ_0



(a) Energy Plot (b) Control History

Fig. 8 QCPI Validation - Unconstrained Hypersonic Trajectory Problem

VI. Numerical Instability due to Fixed Mesh Size

The Quasilinear Chebyshev Picard Iteration algorithm has much in common with collocation methods. Like some collocation-based methods, QCPI represents the solution using an orthogonal polynomial series and uses quadrature rules to integrate the dynamic equations in the problem. This means that QCPI has some of the same drawbacks as these collocation based methods. The solution is represented on an uneven mesh of Chebyshev-Gauss-Lobatto (CGL) nodes as shown in Fig. 9. In this mesh, the nodes are clustered at the beginning and end of the trajectory with fewer nodes in the middle. This causes numerical instabilities when solving problems with dynamically sensitive regions in the middle of the trajectory. In shooting methods, the use of adaptive numerical integrators helps avoid this issue. In collocation-based solvers such as GPOPS, adaptive mesh refinement methods [42, 43] are used to dynamically change the node positions, usually by concatenating meshes of different sizes. This allows the solver to add extra nodes in regions where the trajectory is highly sensitive, thereby improving the numerical stability of the solver.

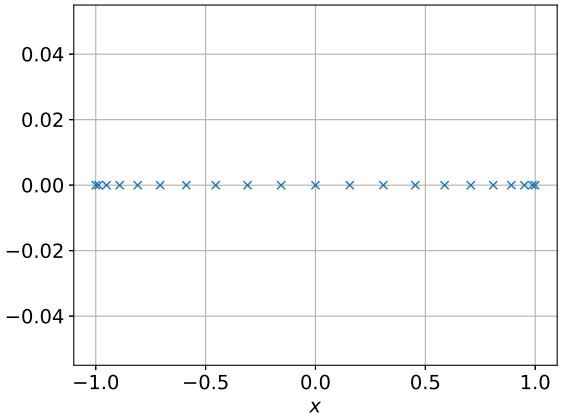


Fig. 9 CGL Nodes for Chebyshev Polynomial Series of Order, $N = 20$

QCPI in its current implementation, uses a fixed size grid that is specified a-priori. While it is still applicable to many nonlinear optimal control problems as illustrated in the previous section, this limits its use for problems with numerically sensitive regions in the middle of the trajectory. The challenges posed by this limitation and some strategies for mitigating them are explored further in Section VII.A.

VII. Future Work

A. Adaptive Grid & Mesh Refinement for QCPI

One of the major limitations of QCPI in its current form is that it uses a fixed-size mesh for representing the solution which has nodes clustered at the beginning and end of the trajectory. While this may be ideal in some cases, those problems that have highly sensitive dynamics towards the middle of the trajectory may be difficult to solve using the current implementation. This is very similar to the challenges encountered when using by collocation-based numerical methods such as *bvp4c* or direct solvers like GPOPS. These solvers use an adaptive mesh-sizing and mesh refinement strategy that adjusts the points where the nodes are clustered based on the sensitivity of the problem.

Some strategies for overcoming this limitation are described in Refs 44, 42, and 43. A similar strategy can be developed for QCPI that uses multiple sets of Chebyshev polynomial series starting and ending at points of high numerical sensitivity. Such an adaptive mesh refinement strategy would help greatly improve the numerical stability of QCPI and make it applicable to a wider range of nonlinear optimal control problems. Ref. 45 describes a method for choosing the order of the Chebyshev series based on the desired accuracy of the solution. This is another strategy that could be implemented in QCPI to make it more robust.

B. Parallel Implementation of QCPI

The work in this paper demonstrated the inherent parallelism of the QCPI algorithm using benchmarks on a multi-core computer. The various components of QCPI – independent evaluation of dynamic equations, matrix multiplication operations, and linear algebra, are all operations that can be very efficiently implemented on parallel processors. The MCPI algorithm that this method was built on was demonstrated to be very efficient at leveraging GPU processors[9, 28]. Similarly the QCPI can also greatly benefit from the highly parallel computing environment offered by GPUs and significantly accelerate the numerical solution of large-scale nonlinear boundary value problems.

VIII. Conclusion

The Quasilinear Chebyshev Picard Iteration (QCPI) method builds on prior work utilizing a Chebyshev Polynomial series and the Picard Iteration combined with the Modified Quasi-linearization Algorithm. The capabilities of this novel numerical method are validated by solving some representative nonlinear optimal control problems. Further benchmarking and analysis is required to compare the performance of QCPI to existing numerical methods.

Even with the limitations of its current implementation, QCPI has been demonstrated to be a viable, fast numerical method for solving a general class of nonlinear two-point boundary value problems. It advances the state-of-the-art in using indirect methods for solving non-trivial trajectory optimization problems.

Appendix

There are two matrices used in formulating the matrix-vector form of the Chebyshev-Picard iteration used by QCPI. The first, C_a, is used to compute the coefficients, $\vec{F}$, for an N-th order Chebyshev Polynomial series to a given function, $g(x)$, as follows:

$$\vec{F} = 2\vec{\chi}_0 + C_a \times \vec{g}(x) \tag{47}$$

where $\vec{g}(x)$ is $g(x)$ evaluated on an N-th order Chebyshev mesh as:

$$\vec{g} = [g(\tau_0), g(\tau_1), ..., g(\tau_N)]^T \tag{48}$$

and $\vec{\chi}_0$ is defined as:

$$\vec{\chi}_0 = \begin{bmatrix} 2x_0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix}^T \tag{49}$$

with x_0 being the initial value of x. C_a is defined as:

$$C_a \equiv RSTV \tag{50}$$

where

$$R = diag\left(\left[1, \frac{1}{2}, \frac{1}{4}, ..., \frac{1}{2(N-1)}, \frac{1}{2N}\right]\right) \tag{51a}$$

$$S = \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{2}{3} & \frac{1}{4} & \frac{-2}{15} & \cdots & (-1)^{N+1}\frac{1}{N-1} \\ 1 & 0 & -1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & -1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \tag{51b}$$

$$T = \begin{bmatrix} T_0(\tau_0) & T_0(\tau_1) & \cdots & T_0(\tau_N) \\ T_1(\tau_0) & T_1(\tau_1) & \cdots & T_1(\tau_N) \\ T_2(\tau_0) & T_2(\tau_1) & \cdots & T_2(\tau_N) \\ \vdots & \vdots & \vdots & \vdots \\ T_N(\tau_0) & T_N(\tau_1) & \cdots & T_N(\tau_N) \end{bmatrix} \tag{51c}$$

$$T_k(\tau_j) = \cos\left(k \arccos \tau_j\right) \tag{51d}$$

$$\tau_j = \cos\left(\frac{j\pi}{N}\right) \tag{51e}$$

$$V = diag\left(\left[\frac{1}{N}, \frac{2}{N}, \frac{2}{N}, ..., \frac{2}{N}, \frac{1}{N}\right]\right) \qquad \text{with } N+1 \text{ elements} \tag{51f}$$

$$\tag{51g}$$

The second matrix is C_x which is used to evaluate an N-th order Chebyshev series represented by its coefficients, $\vec{\beta}$ on a Chebyshev grid.

$$\vec{x} = C_x \times \vec{\beta} \tag{52}$$

C_x is defined as:

$$C_x \equiv TW \tag{53}$$

where

$$W = diag\left(\left[\frac{1}{2}, 1, 1, \cdots, 1, 1\right]\right) \qquad \text{with } N+1 \text{ elements} \tag{54}$$

Acknowledgments

This research was supported in part, by the Air Force Research Laboratory, under award number FA8651-16-2-0004. This work was also supported by Smart Ag Inc., Ames, IA.

References

[1] Antony, T., and Grant, M. J., "Rapid Indirect Trajectory Optimization on Highly Parallel Computing Architectures," *Journal of Spacecraft and Rockets*, Vol. 54, No. 5, 2017, pp. 1081–1091. doi:10.2514/1.A33755, URL https://doi.org/10.2514/1.A33755.

[2] Sparapany, M., "Towards the Real-Time Application of Indirect Methods for Hypersonic Missions," Master's thesis, Purdue University, West Lafayette, 2015.

[3] Fei, Y., Rong, G., Wang, B., and Wang, W., "Parallel L-BFGS-B Algorithm on GPU," *Computers & Graphics*, Vol. 40, 2014, pp. 1–9.

[4] NVIDIA Corporation, "Cuda C Programming Guide," , 2014. URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[5] Antony, T., and Grant, M. J., "Path Constraint Regularization in Optimal Control Problems using Saturation Functions," *AIAA Atmospheric Flight Mechanics Conference*, American Institute of Aeronautics and Astronautics, 2018. doi:10.2514/6.2018-0018, URL https://doi.org/10.2514/6.2018-0018.

[6] Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., and Glasco, D., "GPUs and the Future of Parallel Computing," *IEEE Micro*, Vol. 31, No. 5, 2011, pp. 7–17.

[7] Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., and Storaasli, O. O., "State-of-the-art in heterogeneous computing," *Scientific Programming*, Vol. 18, No. 1, 2010, pp. 1–33.

[8] Agarwal, V., Hrishikesh, M. S., Keckler, S. W., and Burger, D., "Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures," *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ACM, New York, NY, USA, 2000, pp. 248–259. doi:10.1145/339647.339691, URL http://doi.acm.org/10.1145/339647.339691.

[9] Bai, X., "Modified Chebyshev-Picard Iteration Methods for Solution of Initial Value and Boundary Value Problems," Ph.D. thesis, Texas A&M University, 2010.

[10] Coddington, E. A., and Levinson, N., *Theory of Ordinary Differential Equations*, Tata McGraw-Hill Education, 1955.

[11] Tchebychev, P. L., *Théorie des mécanismes connus sous le nom de parallélogrammes*, Imprimerie de l'Académie impériale des sciences, 1853.

[12] Fox, L., and Parker, I., "Chebyshev Polynomials in Numerical Analysis," , 1968.

[13] Mason, J. C., and Handscomb, D. C., *Chebyshev polynomials*, CRC Press, 2002.

[14] Clenshaw, C., "The numerical solution of linear differential equations in Chebyshev series," *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 53, Cambridge University Press, 1957, pp. 134–149.

[15] Clenshaw, C., and Norton, H., "The solution of nonlinear ordinary differential equations in Chebyshev series," *The Computer Journal*, Vol. 6, No. 1, 1963, pp. 88–92.

[16] Wright, K., "Chebyshev collocation methods for ordinary differential equations," *The Computer Journal*, Vol. 6, No. 4, 1964, pp. 358–365.

[17] Gottlieb, D., and Orszag, S. A., *Numerical analysis of spectral methods: theory and applications*, SIAM, 1977.

[18] Sezer, M., and Kaynak, M., "Chebyshev polynomial solutions of linear differential equations," *International Journal of Mathematical Education in Science and Technology*, Vol. 27, No. 4, 1996, pp. 607–618.

[19] Boyd, J. P., *Chebyshev and Fourier spectral methods*, Courier Corporation, 2001.

[20] Urabe, M., "Numerical solution of multi-point boundary value problems in Chebyshev series theory of the method," *Numerische Mathematik*, Vol. 9, No. 4, 1967, pp. 341–366.

[21] Vlassenbroeck, J., and Van Dooren, R., "A Chebyshev technique for solving nonlinear optimal control problems," *IEEE transactions on automatic control*, Vol. 33, No. 4, 1988, pp. 333–340.

[22] Elnagar, G. N., and Kazemi, M. A., "Pseudospectral Chebyshev optimal control of constrained nonlinear dynamical systems," *Computational Optimization and Applications*, Vol. 11, No. 2, 1998, pp. 195–217.

[23] Fahroo, F., and Ross, I. M., "Direct trajectory optimization by a Chebyshev pseudospectral method," *Journal of Guidance, Control, and Dynamics*, Vol. 25, No. 1, 2002, pp. 160–166.

[24] Feagin, T., "The numerical solution of two point boundary value problems using chebyshev series," Ph.D. thesis, Ph. D. dissertation, The Universtiy of Texas at Austin, Austin, TX, 1973.

[25] Shaver, J., "Formulation and evaluation of parallel algorithms for the orbit determination problem," Ph.D. thesis, 1980.

[26] Feagin, T., and Nacozy, P., "Matrix formulation of the Picard method for parallel computation," *Celestial Mechanics and Dynamical Astronomy*, Vol. 29, No. 2, 1983, pp. 107–115.

[27] Fukushima, T., "Vector integration of dynamical motions by the Picard-Chebyshev method," *The Astronomical Journal*, Vol. 113, 1997, p. 2325.

[28] Koblick, D., Poole, M., and Shankar, P., "Parallel High-Precision Orbit Propagation using the Modified Picard-Chebyshev Method," *ASME 2012 International Mechanical Engineering Congress and Exposition*, American Society of Mechanical Engineers, 2012, pp. 587–605.

[29] Bai, X., and Junkins, J. L., "Modified Chebyshev-Picard iteration Methods for Solution of Boundary Value Problems," *The Journal of the Astronautical Sciences*, Vol. 58, No. 4, 2011, pp. 615–642.

[30] Antony, T., and Grant, M. J., "A Generalized Adaptive Chebyshev–Picard Iteration Method for Solution to Two–Point Boundary Value Problems," *3rd Annual Meeting of the AFRL Mathematical Modeling and Optimization Institute*, 2015. URL `http://www.reef.ufl.edu/MMOI/MMOI2015_Book.pdf`.

[31] Miele, A., "Method of particular solutions for linear, two-point boundary-value problems," *Journal of Optimization Theory and Applications*, Vol. 2, No. 4, 1968, pp. 260–273. doi:10.1007/BF00937371, URL `https://doi.org/10.1007/BF00937371`.

[32] Miele, A., and Iyer, R., "General technique for solving nonlinear, two-point boundary-value problems via the method of particular solutions," *Journal of Optimization Theory and Applications*, Vol. 5, No. 5, 1970, pp. 382–399.

[33] Miele, A., Iyer, R. R., and Well, K. H., "Modified quasilinearization and optimal initial choice of the multipliers part 2—Optimal control problems," *Journal of Optimization Theory and Applications*, Vol. 6, No. 5, 1970, pp. 381–409. doi:10.1007/BF00932584, URL `https://doi.org/10.1007/BF00932584`.

[34] Miele, A., Mangiavacchi, A., and Aggarwal, A. K., "Modified quasilinearization algorithm for optimal control problems with nondifferential constraints," *Journal of Optimization Theory and Applications*, Vol. 14, No. 5, 1974, pp. 529–556. doi:10.1007/BF00932847, URL `https://doi.org/10.1007/BF00932847`.

[35] Gonzalez, S., and Rodriguez, S., "Modified quasilinearization algorithm for optimal control problems with nondifferential constraints and general boundary conditions," *Journal of Optimization Theory and Applications*, Vol. 50, No. 1, 1986, pp. 109–128. doi:10.1007/BF00938480, URL https://doi.org/10.1007/BF00938480.

[36] Woollands, R. M., Read, J. L., Macomber, B., Probe, A., Younes, A. B., and Junkins, J. L., "Method of Particular Solutions and Kustaanheimo-Stiefel Regularized Picard Iteration for Solving Two-Point Boundary Value Problems," *AAS/AIAA Spce Flight Meeting, Williamsburg, VA*, 2015.

[37] Lam, S. K., Pitrou, A., and Seibert, S., "Numba: A LLVM-based Python JIT Compiler," *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ACM, New York, NY, USA, 2015, pp. 7:1–7:6. doi:10.1145/2833157.2833162, URL http://doi.acm.org/10.1145/2833157.2833162.

[38] Rossum, G., "Python Reference Manual," Tech. rep., Amsterdam, The Netherlands, The Netherlands, 1995.

[39] Lattner, C., and Adve, V., "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, IEEE Computer Society, Washington, DC, USA, 2004, pp. 75–. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[40] Walt, S. v. d., Colbert, S. C., and Varoquaux, G., "The NumPy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, Vol. 13, No. 2, 2011, pp. 22–30.

[41] Jones, E., Oliphant, T., Peterson, P., et al., "SciPy: Open source scientific tools for Python," , 2001–. URL http://www.scipy.org/, [Online; accessed 10 Jan, 2018].

[42] Darby, C. L., Hager, W. W., and Rao, A. V., "An hp-adaptive pseudospectral method for solving optimal control problems," *Optimal Control Applications and Methods*, Vol. 32, No. 4, 2011, pp. 476–502.

[43] Darby, C. L., Hager, W. W., Rao, A. V., et al., "Direct trajectory optimization using a variable low-order adaptive pseudospectral method," *Journal of Spacecraft and Rockets*, Vol. 48, No. 3, 2011, p. 433.

[44] Shampine, L. F., Kierzenka, J., and Reichelt, M. W., "Solving boundary value problems for ordinary differential equations in MATLAB with bvp4c," *Tutorial notes*, Vol. 2000, 2000, pp. 1–27.

[45] Nacozy, P. E., and Feagin, T., "Approximations of Interplanetary Trajectories by Chebyshev Series," *AIAA Journal*, Vol. 10, No. 3, 1972, pp. 243–244. URL https://doi.org/10.2514/3.50087.