USING GPUS AND THE PARAMETERIZATION METHOD FOR RAPID SEARCH AND REFINEMENT OF CONNECTIONS BETWEEN TORI IN PERIODICALLY PERTURBED PLANAR CIRCULAR RESTRICTED 3-BODY PROBLEMS

Bhanu Kumar∗, Rodney L. Anderson†, and Rafael de la Llave‡

When the planar circular restricted 3-body problem is periodically perturbed, most unstable periodic orbits become invariant tori. However, 2D Poincaré sections no longer work to find their manifolds’ intersections; new methods are needed. In this study, we first review a method of restricting the intersection search to only certain manifold subsets. We then implement this search using Julia and OpenCL, representing the manifolds as triangular meshes and gaining a 30x speedup using GPUs. We finally show how to use manifold parameterizations to refine the approximate connections found in the mesh search. We demonstrate the tools on the planar elliptic RTBP.

INTRODUCTION

Numerous prior studies have used the stable and unstable manifolds of unstable planar periodic orbits, both around libration points as well as at resonances, as an efficient tool for multi-body mission design in the planar circular restricted 3-body problem (PCRTBP). For instance, Anderson and Lo1,2 studied intersections of the manifolds of resonant periodic orbits in the Jupiter-Europa system as a mechanism of resonance transitions. The book of Koon et al.3 describes the Poincaré section method of finding intersections of manifolds between L1 and L2 libration point orbits, and shows how to use the resulting regions to construct trajectories with arbitrary itineraries between the different realms of the PCRTBP model. As the phase space is 4-dimensional, fixing an energy level restricts the dynamics to a 3D submanifold, and the Poincaré section further reduces the dimensionality of the system to 2D. Since the manifolds of the periodic orbits are 2D cylinders in the full phase space, taking the Poincaré section reduces the manifolds to 1D curves in the section, so the problem of finding connections between periodic orbits reduces to finding the intersection of two 1D manifold curves in a 2D plane.

While this method of intersecting 1D curves is useful in the 4-dimensional phase space of the PCRTBP, it is not applicable to systems with higher dimensional phase spaces. For instance, in the 6D phase space of the spatial CRTBP, fixing an energy level and taking a Poincaré section still results in a 5D energy level and 4D section to be explored. In the case of a time-varying periodic perturbation of the PCRTBP, the phase space becomes 5-dimensional, considering time as a state

∗PhD Candidate, School of Mathematics, Georgia Institute of Technology, 686 Cherry St. NW, Atlanta, GA 30332.
†Technologist, Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109.
‡Professor, School of Mathematics, Georgia Institute of Technology, 686 Cherry St. NW, Atlanta, GA 30332.
© 2020. All rights reserved.
variable. As energy is no longer conserved in non-autonomous systems, taking a Poincaré section again leaves us with a 4D space to be explored. Furthermore, in these systems, unstable periodic orbits are no longer the main dynamical structures of interest, as 2D manifolds of periodic orbits do not generically intersect each other in a 5D phase space or energy level; instead, unstable quasi-periodic orbits and their manifolds are the objects of sufficient dimensionality such that one can expect intersections.

In prior work, we described how most PCRTBP unstable periodic orbits will persist as 2D unstable quasi-periodic orbits in the 5D phase space of periodically-perturbed PCRTBP models. By considering stroboscopic maps instead of the continuous-time flow, we can reduce the dimensionality of the system by 1 so that these quasi-periodic orbits become invariant 1D tori (circles) in the 4D stroboscopic map phase space \((x, y, p_x, p_y)\); these invariant circles have 2D cylindrical stable and unstable manifolds as the PCRTBP unstable periodic orbits did. However, due to the absence of an energy integral, manifold intersections in the perturbed system will occur at isolated points, rather than along continuous trajectory curves. Hence, a different method of computing homoclinic and heteroclinic connections in the full 4D phase space is required. The purpose of this study is to develop new and computationally fast methods and tools for this problem.

In this paper, after defining some models and maps, we start by briefly recalling the results of our previously developed parameterization method for the accurate computation of invariant tori and their manifolds in periodically-perturbed PCRTBP models. Next, we review the method of layers for restricting the homoclinic and heteroclinic connection search to appropriate subsets of the two manifolds of interest. Once these subsets are identified, we can computationally represent each manifold as a 2D mesh of points in the 4D stroboscopic map phase space. With these meshes representing the manifolds, we next develop a heavily parallel algorithm for detecting and computing intersections of these meshes in 4D space; we then describe how to implement this method using the Julia programming language and OpenCL, taking advantage of the capabilities of modern graphics processing units (GPU) to massively speed up the algorithm execution time. Finally, we take the approximate intersections of manifolds computed from the mesh intersection search, and show how to use our manifold parameterizations to refine the intersections to high precision. We demonstrate the use of our methods by applying them to the search for heteroclinic connections between resonances in the Jupiter-Europa planar elliptic RTBP.

MODELS AND EQUATIONS OF MOTION

Planar Circular Restricted 3-body Problem

The planar circular restricted 3-body problem (PCRTBP) models the motion of a spacecraft under the gravitational influence of two large bodies which revolve in a circular Keplerian orbit around their barycenter. The large bodies are called the primary body of mass \(m_1\) and secondary body of mass \(m_2\) (together referred to as the primaries), with mass ratio \(\mu = \frac{m_2}{m_1 + m_2}\). After normalizing units, the distance between \(m_1\) and \(m_2\) becomes 1, \(G(m_1 + m_2) = 1\), and their period of revolution becomes \(2\pi\). Using the usual synodic coordinate system with the primaries on the \(x\)-axis and origin at their barycenter, the equations of motion are Hamiltonian of form

\[
\begin{align*}
\dot{x} &= \frac{\partial H_0}{\partial p_x}, & \dot{y} &= \frac{\partial H_0}{\partial p_y}, & \dot{p}_x &= -\frac{\partial H_0}{\partial x}, & \dot{p}_y &= -\frac{\partial H_0}{\partial y} \\
H_0(x, y, p_x, p_y) &= \frac{p_x^2 + p_y^2}{2} + p_x y - p_y x - \frac{1 - \mu}{r_1} - \frac{\mu}{r_2}
\end{align*}
\]
where \( r_1 = \sqrt{(x+\mu)^2 + y^2} \) is the distance from the spacecraft to \( m_1 \) and \( r_2 = \sqrt{(x-1+\mu)^2 + y^2} \) is the distance to \( m_2 \). We assume that the spacecraft moves in the same plane as the primaries. The Hamiltonian in Equation (2) is autonomous and is hence an integral of motion.

**Periodic Perturbations of the PCRTBP**

Many effects not modeled in the PCRTBP act in a time-periodic manner on the spacecraft; these can be represented as periodic perturbations of the PCRTBP. Many such perturbations are also Hamiltonian themselves; in this case, the equations of motion are given by

\[
\begin{align*}
\dot{x} &= \frac{\partial H_\varepsilon}{\partial p_x}, \quad \dot{y} = \frac{\partial H_\varepsilon}{\partial p_y}, \quad \dot{p}_x = -\frac{\partial H_\varepsilon}{\partial x}, \quad \dot{p}_y = -\frac{\partial H_\varepsilon}{\partial y}, \\
\dot{\theta}_p &= \Omega_p
\end{align*}
\]

where \( \theta_p \in \mathbb{T} \) is an angle considered modulo \( 2\pi \), \( H_0 \) is the PCRTBP Hamiltonian from Equation (2), \( H_1 \) is the time-periodic perturbation satisfying \( H_1(x, y, p_x, p_y, \theta_p; 0) = 0 \), and \( \varepsilon > 0 \) and \( \Omega_p \) are the perturbation parameter and perturbation frequency, respectively. \( \varepsilon \) signifies the strength of the perturbation, with \( \varepsilon = 0 \) being the unperturbed PCRTBP, and \( \Omega_p \) is a known constant frequency. The perturbation from \( H_1 \) is \( 2\pi/\Omega_p \) periodic, with \( \theta_p \) being the perturbation phase angle. In general, the Hamiltonian in Equation (4) is not an integral of motion when \( \varepsilon \neq 0 \).

**Planar Elliptic Restricted 3-body Problem**

There are many periodically perturbed PCRTBP models that are of interest, such as the bicircular problem, the quasi-bicircular problem, and the Hill restricted 4-body problem. Another well-known periodically-perturbed PCRTBP model is the planar elliptic restricted 3-body problem (PERTBP), which we use in this study for numerical demonstration of our tools. In the PERTBP, \( m_1 \) and \( m_2 \) revolve around their barycenter in an elliptical Keplerian orbit of nonzero eccentricity \( \varepsilon > 0 \). All other assumptions are the same as the PCRTBP. The length unit is defined such that the \( m_1-m_2 \) orbit semi-major axis is 1, and the period of the primaries’ orbit remains \( 2\pi \). This implies that the perturbation frequency \( \Omega_p = 1 \); hence, we can take \( \theta_p = t \) modulo \( 2\pi \).

We use the same PERTBP model used by Hiday-Johnston and Howell, except for a coordinate change from position-velocity to position-momentum coordinates and a restriction of the dynamics to the \( xy \)-plane. Again, the coordinate system is defined with the primaries on the \( x \)-axis and origin at their barycenter. However, the distance from \( m_1 \) to \( m_2 \) is now time-periodic; \( t = 0, 2\pi, \ldots \) are the times of periapse. This is different from the well-known pulsating coordinates of Szebehely. The equations of motion are given by Equation (3) with time-periodic Hamiltonian

\[
H_\varepsilon(x, y, p_x, p_y, t) = \frac{p_x^2 + p_y^2}{2} + n(t)(p_x y - p_y x) - \frac{1 - \mu}{r_1} - \frac{\mu}{r_2}
\]

where \( r_1 = \sqrt{(x+\mu(1-\varepsilon \cos E(t)))^2 + y^2} \) and \( r_2 = \sqrt{(x-(1-\mu)(1-\varepsilon \cos E(t)))^2 + y^2} \). \( n(t) \) and \( E(t) \) are the \( 2\pi \)-periodic angular rate and eccentric anomaly of the elliptical \( m_1-m_2 \) orbit, and can be computed using the usual Kepler’s problem methods as in Bate, Mueller, and White. The momenta are related to velocity by \( p_x = \dot{x} - n(t)y \) and \( p_y = \dot{y} + n(t)x \).
BACKGROUND

Stroboscopic Maps

The quasi-periodic orbits of interest in periodically-perturbed PCRTBP models lie on 2D unstable invariant tori in the 5D extended phase space \((x, y, px, py, \theta_p)\). These invariant tori can be parameterized as the image of a function of two angles \(K_2 : T^2 \rightarrow \mathbb{R}^4 \times T\). Any quasi-periodic trajectory \(x(t)\) lying on this torus can be expressed as

\[
x(t) = K_2(\theta, \theta_p) \quad \theta = \theta_0 + \Omega_1 t, \quad \theta_p = \theta_{p,0} + \Omega_p t
\]

(6)

where \(\theta_0\) and \(\theta_{p,0}\) are determined from the initial condition. \(\theta_p\) and \(\Omega_p\) are the perturbation phase and frequency defined earlier, respectively. Defining the stroboscopic map \(F : \mathbb{R}^4 \times T \rightarrow \mathbb{R}^4 \times T\) as the time-\(2\pi/\Omega_p\) mapping of extended phase space points by the equations of motion, we have

\[
F(K_2(\theta, \theta_p)) = K_2(\theta + \omega, \theta_p), \quad \text{where} \quad \omega = 2\pi\Omega_1/\Omega_p
\]

(7)

since the angle \(\theta_p\) increases by \(2\pi\) in the time \(2\pi/\Omega_p\). Since the value of \(\theta_p\) does not change under the map \(F\), fix a value of \(\theta_p\) and define \(K(\theta) = K_2(\theta, \theta_p)\) (without loss of generality, we choose \(\theta_p = 0\) in this study). Then, Equation (7) becomes

\[
F(K(\theta)) = K(\theta + \omega)
\]

(8)

Ignoring the invariant \(\theta_p\) component of the extended phase space and making a slight abuse of notation, we have \(F : \mathbb{R}^4 \rightarrow \mathbb{R}^4\) and \(K : T \rightarrow \mathbb{R}^4\). Equation (8) implies that \(K\) is an invariant 1D torus of \(F\). Hence, basing our study on the stroboscopic map \(F\) is more efficient than solving for tori invariant under the flow of the ODE, since we reduce the phase space dimension from 5D to 4D and the dimension of the unknown invariant tori from 2D to 1D. This is the approach we take in the remainder of this study.

Parameterization Methods for Tori, Bundles, and Manifolds

In this section, we summarize the results of the parameterization methods developed in Kumar, Anderson, and de la Llave,\(^4\) which themselves were inspired by methods described in Haro et al.\(^12\) and Zhang and de la Llave.\(^13\) With the stroboscopic map \(F\) defined, we first wish to find solutions \(K(\theta)\) of Equation (8). The rotation number \(\omega = 2\pi\Omega_1/\Omega_p\) is generally known; for instance, this is the case whenever the invariant torus being solved for comes from a known PCRTBP periodic orbit.

The quasi-Newton method developed in our previous study\(^4\) for solving equation (8) adds an extra equation to be solved. In particular, in addition to \(K(\theta)\), we simultaneously solve for matrix-valued periodic functions \(P(\theta), \Lambda(\theta) : T \rightarrow \mathbb{R}^{4 \times 4}\) satisfying

\[
DF(K(\theta))P(\theta) = P(\theta + \omega)\Lambda(\theta)
\]

(9)

\(P(\theta)\) and \(\Lambda(\theta)\) are the matrices of bundles and of Floquet stability, respectively. For each \(\theta \in T\), the columns of \(P(\theta)\) are comprised of the tangent, symplectic conjugate center, stable, and unstable directions of the torus at the point \(K(\theta)\), in that order, while \(\Lambda\) has the form

\[
\Lambda(\theta) = \begin{bmatrix} 1 & T(\theta) & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \lambda_s & 0 \\ 0 & 0 & 0 & \lambda_u \end{bmatrix}
\]

(10)
where \( T : \mathbb{T} \to \mathbb{R} \) and \( \lambda_s, \lambda_u \in \mathbb{R} \) are constants with \( \lambda_s < 1 \) and \( \lambda_u > 1 \). As it turns out, solving simultaneously for \( K, P, \) and \( \Lambda \) not only gives more information, but actually has lower computational complexity than other methods which solve for \( K \) alone.\(^4\)

Once the \( F \)-invariant tori and their center, stable, and unstable torus bundles are computed, we next wish to accurately compute stable and unstable manifolds of the tori under \( F \). As the invariant tori are 1D, their stable and unstable manifolds will be 2D and topologically equivalent to \( \mathbb{T} \times \mathbb{R} \). We can hence parameterize the manifolds as the image of a function \( W(\theta, s) \) satisfying

\[
F(W(\theta, s)) = W(\theta + \omega, \lambda s)
\]  

(11)

where \((\theta, s) \in \mathbb{T} \times \mathbb{R}\), and \( \lambda \) is the stable \( \lambda_s \) or unstable \( \lambda_u \) multiplier from \( \Lambda \), depending on which manifold we are solving for. To solve equation (11), we express \( W(\theta, s) \) as a Fourier-Taylor series

\[
W(\theta, s) = \sum_{k \geq 0} W_k(\theta)s^k = K(\theta) + \sum_{k \geq 1} W_k(\theta)s^k
\]  

(12)

where \( K(\theta) \) is the invariant circle whose manifold we are trying to compute. The \( s^0 \) term of \( W \) is \( W_0(\theta) = K(\theta) \), and the linear term \( W_1(\theta) \) is the stable or unstable bundle known from the third or fourth column of \( P \). The higher order \( W_k(\theta) \) terms can then be solved for recursively, as described in our previous work.\(^4\) Note that \( s = 0 \) corresponds to the base invariant torus.

**Using Parameterizations for Manifold Globalization**

We have found that the torus stable and unstable manifold parameterizations \( W(\theta, s) \) described in the previous section approximate the manifolds very accurately, but only within some finite range of \( s \) values. To find this range of values, we first choose an error tolerance, say \( E_{tol} = 10^{-5} \) or \( 10^{-6} \). We then find the largest \( D \in \mathbb{R}^+ \) such that for all \( s \) satisfying \( |s| \leq D \),

\[
\max_{\theta \in \mathbb{T}} \| F(W(\theta, s)) - W(\theta + \omega, \lambda s) \| < E_{tol}
\]  

(13)

We refer to the set \((-D, D)\) as the fundamental domain of \( W(\theta, s) \). Once \( D \) is computed, we wish to compute manifold points \( W(\theta, s) \) for \( s \) values outside the fundamental domain. This step is called globalization. Repeatedly applying equation (11), we have that \( F^k(W(\theta, s)) = W(\theta + k\omega, \lambda^k s) \), where superscript \( k \) refers to function composition. From this, we can derive two relations:

\[
W(\theta, s) = F^k(W(\theta - k\omega, \lambda^{-k} s))
\]  

(14)

\[
W(\theta, s) = F^{-k}(W(\theta + k\omega, \lambda^k s))
\]  

(15)

These equations can be used to evaluate \( W(\theta, s) \) for \( s \) values outside the fundamental domain. If \( W \) is an unstable manifold with \|\lambda\| > 1, then take \( k \geq 0 \) such that \|\lambda^{-k}s\| < D \) and evaluate Equation (14). Similarly, if \( W \) is a stable manifold with \|\lambda\| < 1, then take \( k \geq 0 \) such that \|\lambda^k s\| < D \) and evaluate Equation (15). The mapping by \( F^k \) or \( F^{-k} \) is just computed using numerical integration.

**THE METHOD OF LAYERS FOR RESTRICTING THE CONNECTION SEARCH**

After computing the stable and unstable manifolds of unstable invariant tori (invariant circles) in a periodically-perturbed PCRTBP, a natural next step is to search for heteroclinic connections between them. Henceforth, let \( W^u_1(\theta_u, s_u) \) and \( W^s_2(\theta_s, s_s) \) represent the unstable and stable manifolds of
stroboscopic map invariant circles 1 and 2, respectively. Heteroclinic connections from circle 1 to
circle 2 occur when the images of \( W_1^u \) and \( W_2^s \) intersect in \((x, y, p_x, p_y)\) space. This means we need to
find \((\theta_u, s_u)\) and \((\theta_s, s_s)\) such that
\[
W_1^u(\theta_u, s_u) = W_2^s(\theta_s, s_s)
\] (16)
In order to solve equation (16), it would help to be able to restrict our solution search to only
certain regions of the \((\theta_u, s_u, \theta_s, s_s)\) space. It is to this end that we define the concept of layers; the
following discussion is conceptually the same as that from our previous study, although we have
changed some definitions to improve the notation of the results.

Let \( \lambda_u \) and \( \lambda_s \) be the multipliers for the internal dynamics on \( W_1^u \) and \( W_2^s \), respectively. Let
\((-D_u, D_u)\) and \((-D_s, D_s)\) be the fundamental domains of the parameterizations of \( W_1^u \) and \( W_2^s \),
respectively. Now, define subsets \( U^+_n, U^-_n \) and \( S^+_n, S^-_n \) of \( W_1^u \) and \( W_2^s \) as follows:
\[
U^+_n = \{ W_1^u(\theta, s) : (\theta, s) \in \mathbb{T} \times [D_u \lambda_u^{n-1}, D_u \lambda_u^n] \} 
\] (17)
\[
U^-_n = \{ W_1^u(\theta, s) : (\theta, s) \in \mathbb{T} \times [-D_u \lambda_u^{n-1}, -D_u \lambda_u^n] \} 
\] (18)
\[
S^+_n = \{ W_2^s(\theta, s) : (\theta, s) \in \mathbb{T} \times [D_s / \lambda_s^{n-1}, D_s / \lambda_s^n] \} 
\] (19)
\[
S^-_n = \{ W_2^s(\theta, s) : (\theta, s) \in \mathbb{T} \times [-D_s / \lambda_s^{n-1}, -D_s / \lambda_s^n] \} 
\] (20)
where \( n \in \mathbb{Z} \). Finally, define \( U_n = U^+_n \cup U^-_n \) and \( S_n = S^+_n \cup S^-_n \). We refer to the subsets \( U_n \)
and \( S_n \) as layers, and to \( U^+_n, S^+_n \) and \( U^-_n, S^-_n \) as positive and negative half-layers, respectively. In
our experience, \( W_1^u(\theta_u, s_u) \) and \( W_2^s(\theta_s, s_s) \) do not intersect for \(|s_u| < D_u\) and \(|s_s| < D_s\); this can
usually be seen from plotting the projections of the manifolds for these \( s \)-values in \((x, y, p_x)\) space.
Hence, if \( W_1^u \) and \( W_2^s \) intersect, it must be that \( U_{n_1} \) intersects \( S_{n_2} \) for some \( n_1, n_2 \in \mathbb{Z}^+ \).

The most important property of these layers is that \( F(U_n) = U_{n+1} \) and \( F(S_n) = S_{n-1} \); more
generally, \( F^k(U_n) = U_{n+k} \) and \( F^k(S_n) = S_{n-k} \) for all \( k \in \mathbb{Z} \). This allows us to restrict our
heteroclinic connection search to only certain pairs of layers of \( W_1^u \) and \( W_2^s \). To see this, suppose
we are searching for a heteroclinic connection which comes from layer \( U_{n_1} \) intersecting layer \( S_{n_2} \)
at \( x \in \mathbb{R}^4 \). Then, since \( F(U_n) = U_{n+1} \) and \( F(S_n) = S_{n-1} \), we have that \( F(x) \) is in the intersection
of \( U_{n_1+1} \) and \( S_{n_2-1} \). More generally, for all \( k \in \mathbb{Z} \), we have that
\[
F^k(x) \in U_{n_1+k} \cap S_{n_2-k} \tag{21}
\]
Now, if \( n_1 \) and \( n_2 \) are both odd or both even, using \( k = \frac{n_2 - n_1}{2} \) in equation (21) gives us \( F^k(x) \in \)
\( U_{\tilde{n}} \cap S_{\tilde{n}} \), where \( \tilde{n} \) \( \triangleq \frac{n_1+n_2}{2} \). On the other hand, if \( n_1 \) and \( n_2 \) are of opposite parity, setting
\( k = \frac{n_2 - n_1 + 1}{2} \) in equation (21) gives us \( F^k(x) \in U_{\tilde{n}} \cap S_{\tilde{n}+1} \), where \( \tilde{n} \) \( \triangleq \frac{n_1+n_2+1}{2} \).

When searching for the heteroclinic trajectory which arises due to the manifolds’ intersection
at \( x \), it is enough to find any point on the orbit of \( x \) under the map \( F \), including \( F^k(x) \) from the
preceding analysis. Based on the above discussion, it is clear that we will find the point \( F^k(x) \) if we
look for intersections of pairs of layers of form \((U_n, S_n)\) or \((U_n, S_{n-1})\) for \( n \in \mathbb{Z}^+ \) (as mentioned
earlier, our experience is that the manifolds do not intersect for \(|s_u| < D_u\) and \(|s_s| < D_s\), so we
only consider positive \( n \)). Since \( x \) was an arbitrary heteroclinic point, if we search for intersections
of pairs of layers of the form just presented above, we will find all possible heteroclinic trajectories.

As a final note, it is easy to see that if \( U_{n_1} \) intersects \( S_{n_2} \), this implies that the time of flight of the
resulting heteroclinic connection from the fundamental domain of one torus manifold to the other
is \( 2\pi(n_1 + n_2) / \Omega_p \); this is because \( n_1 + n_2 \) mappings by \( F \) are required. Hence, the layer indices
can be thought of as a proxy for the connection trajectory time of flight.
RAPID GPU-ASSISTED SEARCH FOR MANIFOLD INTERSECTIONS

With methods of computing manifolds and restricting the connection search to certain layer pairs now developed, we now seek to develop computationally fast methods of finding intersections of the manifold layers. The manifolds being dealt with are geometric objects in 4D space, so by discretizing the manifolds and applying methods inspired by those from computer graphics collision detection algorithms, we are able to very rapidly search a pair of manifolds for intersections. The algorithms are massively sped up by taking advantage of the huge number of threads available on modern graphics processing units (GPUs). We start this section with a brief overview of GPU computing capabilities and paradigms, followed by a description of the manifold discretization. After this we give the full explanation and demonstration of our intersection search algorithm.

An Overview of GPU Computing

Graphics processing units (GPUs) are special-purpose computer processors originally designed for executing computations required for 3D graphics. More recently, as the capabilities of GPUs have grown, it has become possible to use GPUs for many other computational tasks as well. GPUs excel at tasks with large and highly parallel computational requirements, as the GPU processes blocks of many elements in parallel using the same program. While a single-program multiple-data (SPMD) programming model is supported on GPUs, due to the lock-step execution of the program on multiple data elements, it is necessary to evaluate both sides of any code branches for all elements in a block of data. Hence, GPUs are best suited to straight-line programs which are mostly written in a single-instruction, multiple data (SIMD) style. Flow control should be kept to a minimum.

There are two main toolkits used for programming GPUs directly. The most commonly used is Nvidia CUDA, which works only with Nvidia GPUs. The other is OpenCL, which is an open standard which provides a programming language and APIs that can be used with a variety of SIMD-capable devices, including both AMD and Nvidia GPUs as well as CPUs. OpenCL implementations exist for many different platforms, including Windows, MacOS, and Linux x64. The basic programming concepts of CUDA and OpenCL are very similar; as we used OpenCL in this study, we briefly go over the OpenCL model here, covering the concepts used in this work.

In OpenCL, the fundamental task is the programming of kernels. A kernel is a program which executes on the OpenCL device, like a GPU. When a kernel is submitted for execution, a 1D, 2D, or 3D space of indices is defined, called an NDRange (we only use a 1D NDRange in this study); each index corresponds to a separate execution of the kernel. These kernel instances are referred to as work items, with each work item identified by a unique global ID (a nonnegative integer if NDRange is 1D) corresponding to an index in NDRange. Each work item executes the same kernel, but can access different data in memory. Work items are grouped into work groups; all work items in a work group execute concurrently. Finally, the work groups taken together form the NDRange. The key advantage of GPUs is the massive number of threads they have, usually on the order of thousands, which allows them to execute large numbers of work items in parallel.

The OpenCL memory model has various categories of memory stored on the device, accessible to different parts of the program. Private memory is accessible only to a single work item, and local memory is accessible to all work items in the same work group. We do not use local memory in the programs written for this study. Finally, global memory is accessible to all work items. The use of global memory can be a problem if two work items try to access the same memory location at the same time; it is for this purpose that atomic functions are useful. These functions receive a pointer...
__kernel void findall(__global const char *in_array, __global uint *true_idxs, __global int *curr_idx) {
    unsigned int gid = get_global_id(0);
    if (in_array[gid] == 1) {
        uint old_val = atomic_inc(curr_idx);
        true_idxs[old_val] = gid + 1;
    }
}

Figure 1. Example of OpenCL findall kernel

to a 32 bit integer or floating point number stored in global or local memory, modify the value there, and return the old value. The key is that atomic functions execute so that when one work item is carrying out an atomic operation at a pointer location, the other threads must wait for that work item to complete the operation. Some such functions are atomic add, subtract, increment, min, and max.

An example of a simple OpenCL kernel that we use later is given in Figure 1. This program takes as input an array of zeros and ones (denoted as in_array) and finds the in_array entries which have a value of 1, saving their indices in the array true_idxs. It also takes a pointer curr_idx to an integer which is initialized to 0 before the kernel is run. The first line after the kernel declaration stores the work item global ID as gid. Then, the kernel checks if the gid entry of in_array is 1, so that each work item checks a different entry of in_array. If the gid entry is true, an atomic increment is applied to curr_idx, with the pre-increment value of curr_idx stored in old_val; each work item stores its value of old_val in private memory, separately from all other work items. curr_idx is in global memory, so it is shared between all work items. Hence, the first work item to apply atomic_inc sets its private old_val to 0 and the shared curr_idx to 1, the next work item to apply atomic_inc sets its old_val to 1 and curr_idx to 2, and so on. This way, each work item with in_array[gid] equal to 1 gets a unique consecutive value of old_idx. Finally, the gid values with in_array[gid] equal to 1 are stored in the true_idxs array at the indices old_idx by the corresponding work items (we add 1 to gid since we later use the true_idxs array in Julia, which has one-based indexing).

Discrete Mesh Representation of Manifolds

In the background section, we described how it is possible to compute Fourier-Taylor parameterizations $W(\theta, s)$ of the stable and unstable manifolds of stroboscopic map invariant circles. We also gave Equations (14) and (15) demonstrating how to use the parameterizations to compute $W(\theta, s)$ for $s$-values outside the fundamental domain $(-D, D)$. Now, we wish to use these tools to generate a discrete representation of the globalized manifold that can be used for computations and analysis.

When we compute functions of $\theta$, such as $K(\theta), P(\theta), \Lambda(\theta)$, or the manifold $s^k$ coefficients $W_k(\theta)$, we represent them on the computer as arrays of function values at $N$ evenly spaced $\theta$ values $\theta_i = 2\pi i/N, i = 0, 1, \ldots, N - 1$. By also taking a grid of $2K + 1$ evenly spaced $s$-values $s_k = kD/K, k = -K, -K+1, \ldots, 0, 1, \ldots, K$ from $-D$ to $D$, we end up with a set of $N(2K + 1)$ ordered pairs $(\theta_i, s_k)$. Using our Fourier-Taylor parameterizations to compute and store $W(\theta_i, s_k)$ for each $i = 0, \ldots, N - 1$ and $k = -K, \ldots, K$, we get a set of $N(2K + 1)$ points on the manifold.
Once the initial grids of values $W(\theta_i, s_k)$ have been stored, comes the step of numerically globalizing the manifolds. We describe the case of an unstable manifold with multiplier $\lambda > 1$; the stable case is very similar except for the use of $F^{-1}$ and a few sign changes. For this, fixing $k$ and applying Equation (14) to $W(\theta_i, s_k)$ for all $i = 0, 1, \ldots, N - 1$ gives $N$ manifold points $F^n(W(\theta_i, s_k)) = W(\theta_i + n\omega, \lambda^n s_k)$. These are all at the same $s = \lambda^n s_k$ value but are at $\theta$ values shifted from the $\theta_i$. We want all of our manifold points to have the same $\theta$ values; hence, to shift them back, we use a fast Fourier transform (FFT) based translation algorithm.

Suppose one has a periodic function $a(\theta)$ whose values are stored at $\theta_i = 2\pi i/N, i = 0, 1, \ldots, N - 1$. Then, we can take the FFT of this array of function values to find the first $N$ Fourier coefficients $\hat{a}(i)$, $i = 0, 1, \ldots, N - 1$. Finally, using the usual formula relating $a(\theta_i)$ to $\hat{a}(i)$, we have

$$a(\theta_i) = \frac{1}{N} \sum_{k=0}^{N-1} \hat{a}(k)e^{jk\theta_i} \rightarrow a(\theta_i + \rho) = \frac{1}{N} \sum_{k=0}^{N-1} [\hat{a}(k)e^{jk\rho}]e^{jk\theta_i}$$

(22)

where $j$ denotes $\sqrt{-1}$ in Equation (43). Hence, given the $a(\theta_i)$ values, to find the values $a(\theta_i + \rho)$, one takes the FFT, multiplies the $k$th Fourier coefficient by $e^{jk\rho}$, and takes the inverse FFT. By using this algorithm with $\rho = -n\omega$, given the $N$ values $W(\theta_i + n\omega, \lambda^n s_k)$ found earlier, we can find $W(\theta_i, \lambda^n s_k)$. We now compute and store the points $W(\theta_i, \lambda^n s_k)$, along with the corresponding $s$-values, for $n = 1, \ldots, n_{max}$, up to some $n_{max} \in \mathbb{Z}^+$. We do this for all $k = -K, \ldots, K$.

The $s$ values $\lambda^n s_k$ for $n = 0, 1, \ldots, n_{max}$, $k = -K, \ldots, K$ for the computed points will form an unevenly spaced finite set ranging from $s = -\lambda^{n_{max}} D$ to $\lambda^{n_{max}} D$. Redefine $\{s_k\}$ now to be the set of all these $s$ values; for easier notation, we sort and reindex $\{s_k\}$ so that $k = 1, \ldots, M$ where $M$ is the length of $\{s_k\}$. Note that since $-D$ and $D$ were part of our initial grid of $s$ values, the $U_n$ layer boundaries (corresponding to points with $s = \pm \lambda^n D$) will be contained in our set $\{s_k\}$; this fact will be useful later. In our case, we stored the $x, y, p_x,$ and $p_y$ values of the computed manifold points in 4 separate $2D$ $N \times M$ arrays on the computer, so that the $(i + 1, k)$ entry of each array is the $x, y, p_x$, or $p_y$ coordinate of $W(\theta_i, s_k)$. Moving down a column of each array corresponds to increasing $\theta$ and constant $s$, and moving across a row corresponds to constant $\theta$ and increasing $s$.

With manifold points computed on a discrete grid of $(\theta, s)$ ordered pairs, we have the numerical results required in order to obtain the mesh representation of the manifold. We have the values of $W(\theta, s)$ at the points $\{(\theta_i, s_k)\}$ for $i = 0, \ldots, N - 1, k = 1, \ldots, M$. Consider the index $i$ to be
modulo $N$, so that $\theta_N = \theta_0$ and $\theta_{-1} = \theta_{N-1}$. To form the manifold mesh, connect $W(\theta_i, s_k)$ with $W(\theta_{i-1}, s_k)$, $W(\theta_{i+1}, s_k)$, and $W(\theta_i, s_{k+1})$ using line segments. If $k - 1$ or $k + 1$ is outside the range of allowed indices $1, \ldots, M$ (which is true if $k = 1$ or $k = M$, respectively), then omit the corresponding segment from the mesh. This yields a quadrilateral mesh representation of $W$, as is schematically illustrated on the left of Figure 2. We denote the $(i,k)$ quadrilateral to be that with vertex set $Q_{ik} = \{W(\theta_i, s_k), W(\theta_{i+1}, s_k), W(\theta_i, s_{k+1}), W(\theta_{i+1}, s_{k+1})\}$, where $i = 0, \ldots, N - 1$, $k = 1, \ldots, M - 1$ enumerate the $N(M - 1)$ quadrilaterals in the mesh.

As the vertices of a quadrilateral in 4D do not determine a plane, it is necessary to consider each quadrilateral as being composed of two triangles for later computational purposes. We split the $(i,k)$ quadrilateral into two triangles by connecting the vertex $W(\theta_i, s_k)$ with $W(\theta_{i+1}, s_{k+1})$. Hence, for each ordered pair $(i, k)$, $i = 0, \ldots, N - 1$, $k = 1, \ldots, M - 1$, we have two triangles. This gives a triangular mesh for $W$. The right of Figure 2 shows a schematic representation of the mesh construction, illustrating all the points which are connected to each other.

A 3D projection of an example globalized stable manifold (denoted $W^s$) of a 3:4 Jupiter-Europa PERTBP invariant circle is given in Figure 3; this figure was generated using MATLAB’s mesh function, which generates a quadrilateral mesh similar to the one described here.

**GPU-Accelerated Manifold Mesh Intersection Search**

Now that we have described how to construct the quadrilateral and triangular mesh representations of the manifolds, we start searching for heteroclinic connections. As defined earlier, let $W^u_1(\theta_u, s_u)$ and $W^s_2(\theta_s, s_s)$ represent the unstable and stable manifolds of stroboscopic map invariant circles 1 and 2, respectively. Let $\lambda_u$ and $\lambda_s$ be the multipliers for the internal dynamics and $D_u$ and $D_s$ be the fundamental domains of the parameterizations for $W^u_1$ and $W^s_2$, respectively. After computing and storing the vertices of the meshes for $W^u_1$ and $W^s_2$, the problem of finding heteroclinic connections becomes that of finding intersections of these two meshes in 4D space.

We know that we can restrict our attention to finding intersections of only certain layers of the manifolds; using the notation from earlier, we seek to find intersections of pairs of layers of the form $(U_n, S_n)$ or $(U_n, S_{n-1})$. Equivalently, we need to check if $U^+_n$ or $U^-_n$ intersect any of $S^+_n$, $S^-_n$, $S^+_{n-1}$, or $S^-_{n-1}$, for $n \in \mathbb{Z}^+$. Recalling that the $s$-values generated during manifold globalization include those for the boundaries of these layers and half-layers, it turns out that the half-layers just
correspond to easily identified subsets of the manifold meshes. For $U_n^+$, one simply takes the $W_n^+$ mesh vertices which satisfy $s_u \in [D_u \lambda_u^{n-1}, D_u \lambda_u^n]$. For $S_n^+$, take vertices of the $W_n^+$ mesh with $s_u \in [D_s / \lambda_s^{n-1}, D_s / \lambda_s^n]$. The negative half-layers are the same except for a change in the signs of $D_s$ and $D_u$. If the manifold coordinates are stored in four 2D arrays as described earlier, with each column containing the coordinates of all points for a given $s$ value, then the vertex set of a half-layer mesh is just comprised of points from a contiguous set of columns.

With the meshes for the half-layers identified, we finally arrive at the problem of searching for intersections of two half-layer meshes. Let us say that the mesh vertices corresponding to an unstable half-layer are $W^+_1(\theta_{u,i}; s_{u,k}), i = 0, \ldots, N_1 - 1, k = 1, \ldots, M_1$; for the stable half-layer mesh let the vertices be $W^+_2(\theta_{s,j}; s_{s,\ell}), j = 0, \ldots, N_2 - 1, \ell = 1, \ldots, M_2$. As was done in the previous section, define the quadrilateral vertex sets

$$
Q_{ik}^u = \{W^+_1(\theta_{u,i}; s_{u,k}), W^+_1(\theta_{u,i+1}; s_{u,k}), W^+_1(\theta_{u,i+1}; s_{u,k+1}), W^+_1(\theta_{u,i+1}; s_{u,k+1})\}
$$

$$
Q_{j\ell}^s = \{W^+_2(\theta_{s,j}; s_{s,\ell}), W^+_2(\theta_{s,j+1}; s_{s,\ell}), W^+_2(\theta_{s,j+1}; s_{s,\ell+1}), W^+_2(\theta_{s,j+1}; s_{s,\ell+1})\}
$$

where $i = 0, \ldots, N_1 - 1; j = 0, \ldots, N_2 - 1; k = 1, \ldots, M_1 - 1; \text{ and } \ell = 1, \ldots, M_2 - 1$ (again consider the indices $i$ and $j$ to be modulo $N_1$ and $N_2$, respectively).

For notational convenience, we also use $Q_{ik}^u$ and $Q_{j\ell}^s$ to refer to the quadrilaterals formed by the vertices contained therein. As described earlier, we consider each $Q_{ik}^u$ and $Q_{j\ell}^s$ to be comprised of two triangles, defined by vertex sets

$$
T_{ik}^{u1} = \{W^+_1(\theta_{u,i}; s_{u,k}), W^+_1(\theta_{u,i+1}; s_{u,k}), W^+_1(\theta_{u,i+1}; s_{u,k+1})\}
$$

$$
T_{ik}^{u2} = \{W^+_1(\theta_{u,i+1}; s_{u,k}), W^+_1(\theta_{u,i+1}; s_{u,k+1}), W^+_1(\theta_{u,i+1}; s_{u,k+1})\}
$$

$$
T_{j\ell}^{s1} = \{W^+_2(\theta_{s,j}; s_{s,\ell}), W^+_2(\theta_{s,j+1}; s_{s,\ell}), W^+_2(\theta_{s,j+1}; s_{s,\ell+1})\}
$$

$$
T_{j\ell}^{s2} = \{W^+_2(\theta_{s,j+1}; s_{s,\ell}), W^+_2(\theta_{s,j+1}; s_{s,\ell+1}), W^+_2(\theta_{s,j+1}; s_{s,\ell+1})\}
$$

Again for ease of notation, we also use $T_{ik}^{u1}, T_{ik}^{u2}, T_{j\ell}^{s1}$, and $T_{j\ell}^{s2}$ to refer to the plane triangles formed by the vertices contained therein. We have that $Q_{ik}^u = T_{ik}^{u1} \cup T_{ik}^{u2}$ and $Q_{j\ell}^s = T_{j\ell}^{s1} \cup T_{j\ell}^{s2}$.

Now, the problem of searching for intersections between the half-layers can be solved by checking whether any quadrilateral $Q_{ik}^u$ intersects any quadrilateral $Q_{j\ell}^s$, we say that $Q_{ik}^u$ intersects $Q_{j\ell}^s$ if any of the triangles $T_{ik}^{u1}$ or $T_{ik}^{u2}$ intersect either of $T_{j\ell}^{s1}$ or $T_{j\ell}^{s2}$. There are $N_1(M_1 - 1)$ quadrilaterals in the unstable manifold half-layer mesh, and $N_2(M_2 - 1)$ in the stable manifold half-layer mesh; hence, $N_1 N_2 (M_1 - 1)(M_2 - 1)$ pairs of quadrilaterals must be checked for intersection. For an example computation in the PERTBP computing intersections between manifolds of 3:4 and 5:6 resonant invariant circles, we had $N_1 = 1024, N_2 = 2048$, and $M_1 = M_2 = 35$, for a total of $2,424,307,712$ pairs of quadrilaterals to be checked; each quadrilateral pair check involves 4 triangle to triangle checks, which gives 9,697,230,848 pairs of triangles to check for our example. Given the massive number of identical checks to be done, it is clear that a GPU will be well suited to this application.

It is possible to determine whether two 2D triangles intersect in 4D by solving a $4 \times 4$ system of linear equations and checking whether the solution satisfies certain conditions, to be described in more detail later (see equation (31)). However, solving a different $4 \times 4$ system for each of billions of triangle pairs would be extremely computationally expensive, in addition to not being a suitable algorithm to implement on a GPU. However, when checking two half-layer meshes for intersection, it is expected that the vast majority of pairs of quadrilaterals will not intersect. Hence,
it is necessary to first have a computationally cheap and fast test which can reject most of the non-intersecting quadrilateral pairs. For this, we combine two well-known rejection tests from collision detection algorithms in computer graphics, both of which can be run on the GPU.

**Bounding Box Test**  The first test we use is a simple axis-aligned bounding box test.\(^{17}\) This test is generally used with geometric objects in 3D space, but the concept works the same in 4D. The basic idea is to consider each quadrilateral to be enclosed by a minimal 4D box (the bounding box) having its edges parallel to the \(x, y, p_x\), and \(p_y\) axes. Then, to check whether two quadrilaterals \(Q^u_{ik}\) and \(Q^s_{j\ell}\) might intersect, simply check whether their corresponding bounding boxes intersect. If they do not, then we can reject the possibility of the quadrilaterals intersecting; if the boxes do intersect, then additional testing is required. Figure 4 illustrates how this test works in 2D. The test is equivalent to checking whether the maximum \(x\) coordinate of the 4 vertices of \(Q^u_{ik}\) is less than the minimum \(x\) coordinate of the 4 vertices of \(Q^s_{j\ell}\); we also reverse the roles of \(Q^u_{ik}\) and \(Q^s_{j\ell}\) and repeat the check. The same is also done for the \(y\), \(p_x\), and \(p_y\) coordinates; if any of the checks are true, then the quadrilaterals do not intersect.

**Möller Quick Test**  Once the bounding box test has rejected most of the non-intersecting quadrilateral pairs, there are still often a fair number of pairs left. Hence, we run a second fast rejection test on the remaining pairs which is due to Möller.\(^{18}\) The full Möller test is a two-part triangle-triangle intersection test developed for applications in 3D graphics; we use only the first part of this test, which is for quick rejection of non-intersecting pairs of triangles. Given two triangles in 3D space, the quick test checks whether all the vertices of one triangle are on the same side of the plane formed by the other; if so, the pair of triangles cannot intersect. In our case, since our objects are in 4D \((x, y, p_x, p_y)\) space, we project the quadrilaterals and their constituent triangles onto 3D \((x, y, p_x)\) space in order to carry out the test. Of course, if the 3D projected quadrilaterals and triangles do not intersect, then neither will the full quadrilaterals and triangles in 4D space.

Denote the projected quadrilaterals in \((x, y, p_x)\) space as \(\tilde{Q}^u_{ik}\) and \(\tilde{Q}^s_{j\ell}\), and their constituent projected triangles as \(\tilde{T}^{u1}_{ik}\), \(\tilde{T}^{u2}_{ik}\) and \(\tilde{T}^{s1}_{j\ell}\), \(\tilde{T}^{s2}_{j\ell}\). Then, the first step in the Möller quick test is to see whether all four vertices of \(\tilde{Q}^s_{j\ell}\) are (1) on the same side of \(\tilde{T}^{u1}_{ik}\), and (2) on the same side of \(\tilde{T}^{u2}_{ik}\). Both statements (1) and (2) must be true in order to rule out an intersection with \(\tilde{Q}^u_{ik}\). To check (1), we first need the equation of the plane in which \(\tilde{T}^{u1}_{ik}\) lies. This can be found using standard

![Figure 4. Illustration of bounding box test in 2D space](image-url)
techniques; the plane will be comprised of all points \( X \in \mathbb{R}^3 \) such that

\[
f_{ik}^{u1}(X) \overset{\text{def}}{=} N_{ik}^{u1} \cdot (X - \tilde{W}_{i1}^u(\theta_{u,i}, s_{u,k})) = 0
\]  

(29)

\[
N_{ik}^{u1} = \left[ \tilde{W}_{i1}^u(\theta_{u,i+1}, s_{u,k}) - \tilde{W}_{i1}^u(\theta_{u,i}, s_{u,k}) \right] \times \left[ \tilde{W}_{i1}^u(\theta_{u,i+1}, s_{u,k+1}) - \tilde{W}_{i1}^u(\theta_{u,i}, s_{u,k}) \right]
\]  

(30)

where \( \tilde{W}_{i1}^u(\theta, s) \) just denotes the projection of the point \( W_{i1}^u(\theta, s) \) into \((x, y, p_x)\) space. With the equation (29) of the plane of \( T_{ik}^{u1} \) found, we can determine whether all vertices of \( Q_{j\ell}^u \) are on the same side of \( T_{ik}^{u1} \) by simply evaluating \( f_{ik}^{u1}(X) \) at the four vertices and seeing if the resulting four values all have the same sign. If they do, then (1) is true, otherwise it is not.

To check (2), we do the same procedure, but using the vertices of \( T_{ik}^{u2} \) instead of those of \( T_{ik}^{u1} \). If both (1) and (2) are true, then we reject the possibility of \( Q_{j\ell}^u \) intersecting \( Q_{ik}^u \). Otherwise, we carry out the same test as above, but with the roles of \( Q_{j\ell}^u \) and \( Q_{ik}^u \) swapped so that we check whether all four vertices of \( Q_{ik}^u \) are (1) on the same side of \( T_{ik}^{u1} \), and (2) on the same side of \( T_{ik}^{u2} \). If both of these statements are true, then we conclude that \( Q_{j\ell}^u \) cannot intersect \( Q_{ik}^u \); otherwise, we move on to the final, precise test.

**Precise Test**

Thankfully, after running the bounding box and quick Möller tests on the quadrilateral pairs generated from two half-layer meshes, the number of unrejected quadrilateral pairs left to check is usually quite small. For our earlier example with 2,424,307,712 pairs of quadrilaterals to be checked for each pair of half-layers, after the two quick rejection tests, there would only be at very most a few tens of thousands of cases left to test, and usually far less. In fact, for pairs of half-layers with smaller indices (such as \( U^+_2, S^+_1 \)), our experience is that the bounding box test alone rejects all of the quadrilateral pairs! For those few pairs of quadrilaterals which have not been rejected by either the bounding box or the quick Möller tests, we now need to run a more computationally heavy test to check for intersections, as well as computing the intersection if it exists.

Suppose that \( Q_{ik}^u \) and \( Q_{j\ell}^u \) are such a pair of quadrilaterals which still need to be checked for intersection. At this stage, we start dealing exclusively with their constituent triangles; we check whether any of \( T_{ik}^{u1} \) or \( T_{ik}^{u2} \) intersect either of \( T_{j\ell}^{s1} \) or \( T_{j\ell}^{s2} \). For this, we need an algorithm to determine whether (and if so, where) two triangles intersect each other in 4D space. Let \( T_1 \) and \( T_2 \) be two triangles with vertices \( x_1, x_2, x_3 \in \mathbb{R}^4 \) and \( y_1, y_2, y_3 \in \mathbb{R}^4 \), respectively. Then, to determine whether \( T_1 \) and \( T_2 \) intersect, the first step is to find the intersection of the planes containing these two triangles. The equation to solve to help find this is

\[
x_2 + (x_1 - x_2)a + (x_3 - x_2)b = y_2 + (y_1 - y_2)c + (y_3 - y_2)d
\]  

(31)

where \( a, b, c, d \in \mathbb{R} \) are the quantities to be solved for. Equation (31) is a 4D linear equation with 4 unknowns, so it generically admits a unique solution. After solving for \( a, b, c, \) and \( d \), it is easy to see that the LHS and RHS of equation (31) are themselves equal to the intersection point of the planes containing the two triangles; the values of \( a, b, c, \) and \( d \) determine whether this intersection point lies on each triangle itself. The conditions for the intersection to be in \( T_1 \) and \( T_2 \) are simple; we just need \( a,b,c,d \geq 0 \), \( a + b \leq 1 \), and \( c + d \leq 1 \).

If the three conditions just given are not all satisfied, then it is confirmed that \( T_1 \) and \( T_2 \) do not intersect. If they are all satisfied, then the triangles do intersect and the intersection point is given by either side of Equation (31). Furthermore, if \( T_1 \) and \( T_2 \) are \( T_{ik}^{u1} \) or \( T_{ik}^{u2} \) and \( T_{j\ell}^{s1} \) or \( T_{j\ell}^{s2} \), respectively, we can actually use the values of \( a, b, c, \) and \( d \) to estimate a solution for the equation

\[
W_{1i}^u(\theta, s) = W_{2i}^s(\theta, s)
\]  

(32)
If, for instance, $T_1 = T_{ik}^1$ with $x_1 = W_1^u(\theta_{u,i+1}, s_{u,k}), x_2 = W_1^v(\theta_{u,i}, s_{u,k}), x_3 = W_1^z(\theta_{u,i}, s_{u,k+1})$; and $T_2 = T_{j\ell}^2$ with $y_1 = W_2^x(\theta_{s,j+1}, s_{s,\ell}), y_2 = W_2^y(\theta_{s,j}, s_{s,\ell}), y_3 = W_2^z(\theta_{s,j}, s_{s,\ell+1})$, we have

$$
(\theta_{u,i}, s_{u,k}) \approx ((1-a)\theta_{u,i} + a\theta_{u,i+1}, (1-b)s_{u,k} + bs_{u,k+1}) \quad (33)
$$

$$
(\theta_{s,j}, s_{s,\ell}) \approx ((1-c)\theta_{s,j} + c\theta_{s,j+1}, (1-d)s_{s,\ell} + ds_{s,\ell+1}) \quad (34)
$$

We store these approximate solutions of equation (32) along with the mesh intersection points.

One thing to note about this search for manifold intersections is that we represent the manifolds using planar meshes, since the mesh faces are triangles. The true manifolds lie close to these triangular faces, but the manifolds are curved rather than planar. Hence, any intersection found from this search will be subject to some error, on the order of the squares of $a, b, c,$ and $d$. We will describe how to refine these manifold intersections to higher accuracy later in the paper.

**Computational Implementation**

With the three quadrilateral-to-quadrilateral tests of our mesh intersection algorithm explained, we now describe the implementation of our manifold mesh intersection method in a computer program. Our programs were written using the Julia programming language, a relatively new high-level language which has gained significant interest in recent years due to its excellent performance, multiple-dispatch features, and variety of high-quality packages (many of which work seamlessly with each other, thanks to multiple dispatch). There is a Julia package called OpenCL.jl which allows one to transfer data to and from an OpenCL device and run OpenCL kernels from Julia programs; the C code for the kernel is simply passed as a large string to an OpenCL.jl function, which generates a kernel which can be run from within Julia. Identification of manifold mesh vertices belonging to a certain half-layer is just a matter of careful indexing; most of the computations occur in trying to detect intersections of two half-layers, so it is this part of the method we focus on here.

As was defined earlier, let $W_1^u(\theta_{u,i}, s_{u,k}), i = 0, \ldots, N_1 - 1, k = 1, \ldots, M_1$ and $W_2^x(\theta_{s,j}, s_{s,\ell}), j = 0, \ldots, N_2 - 1, \ell = 1, \ldots, M_2$ be the vertices of the unstable and stable manifold half-layer meshes being considered, respectively. We store the vertex coordinates of each manifold half-layer mesh in four 2D arrays, one for each of $x, y, p_x,$ and $p_y$; using the convention of 1-based indexing as in MATLAB and Julia, the $(i + 1, k)$ entries of the four arrays containing unstable half-layer coordinates are the $x, y, p_x,$ and $p_y$ coordinates of $W_1^u(\theta_{u,i}, s_{u,k})$. The $(j + 1, \ell)$ entries of the four arrays containing stable half-layer coordinates are the $x, y, p_x,$ and $p_y$ coordinates of $W_2^x(\theta_{s,j}, s_{s,\ell})$. These eight 2D arrays are provided as inputs to our mesh intersection function, which starts off by transferring the coordinate arrays to the GPU. We also allocate an output buffer of $N_1N_2(M_1 - 1)(M_2 - 1)$ 8-bit integers on the GPU; we will store a true or false value for each quadrilateral pair in this buffer, to record the status of the quick rejection tests. We also supply the GPU with a pointer to an integer initialized to zero, which will serve as a counter for how many quadrilateral pairs are not rejected.

Once the data is transferred to the GPU and buffers allocated, we execute our rejection test kernel. As OpenCL kernels are written in a version of C, the 2D indexing used in Julia does not apply here when reading from the arrays of half-layer coordinates. Furthermore, we need a way to enumerate which two quadrilaterals $Q_{ik}^u$ and $Q_{j\ell}^x$ out of the $N_1N_2(M_1 - 1)(M_2 - 1)$ possible pairs, will be tested by each work item. Hence, inside this kernel, we first do some index conversion. Each work item tests a different pair of quadrilaterals for rejection; the entire kernel is too long to include in this paper, but the essential steps are summarized below.
1. Store the work item global ID as int gid; gid will range from 0 to \( N_1 N_2 (M_1 - 1)(M_2 - 1) - 1 \).

2. Convert gid to a 4D Cartesian index \((i, j, k, \ell)\) using the C mod (%) and integer division (/) operations: 
   \[ i = \text{gid} \mod N_1; \ j = (\text{gid} \mod (N_1 \cdot N_2)) / (N_1); \ k_1 = (\text{gid} \mod (N_1 \cdot N_2 \cdot (M_1 - 1))) / (N_1 \cdot N_2); \ \ell_1 = \text{gid} / (N_1 \cdot N_2 \cdot (M_1 - 1)) \].
   These formulas assign a unique \((i, j, k_1, \ell_1)\) to each gid, with \( i = 0, \ldots, N_1 - 1 \); \( j = 0, \ldots, N_2 - 1 \); \( k_1 = 0, \ldots, M_1 - 2 \); \( \ell_1 = 0, \ldots, M_2 - 2 \).

3. Consider \((i, k_1)\) and \((j, \ell_1)\) as 2D zero-indexed Cartesian indices; convert them to linear indices \(\text{idx}_1\) and \(\text{idx}_2\), respectively, using column major indexing.

4. Define \(k = k_1 + 1\) and \(\ell = \ell_1 + 1\). Read the coordinates of the \(Q_{ik}^u\) and \(Q_{j\ell}^s\) vertices from the input coordinate arrays using the linear indices \(\text{idx}_1\) and \(\text{idx}_2\). One vertex of \(Q_{ik}^u\) will be stored at \(\text{idx}_1\); the others will be at \(\text{idx}_1+1, \text{idx}_1+N_1, \text{and} \text{idx}_1+N_1+1\). Similar for \(Q_{j\ell}^s\).

5. Use fmin/fmax to find smallest/largest \(x, y, p_x, p_y\) coordinate of vertices of \(Q_{ik}^u\) and \(Q_{j\ell}^s\). Carry out bounding box test; store 0 in temp_var if intersection is rejected, otherwise 1.

6. Skip this step if temp_var == 0, else compute normals \(N_{ik}^{u1}\) and \(N_{ik}^{u2}\). Carry out quick Möller test to check if \(Q_{j\ell}^s\) vertices are all on the same side of \(T_{ik}^{u1}\) and \(T_{ik}^{u2}\). OpenCL has functions for cross and dot products. Store 0 in temp_var if intersection is rejected, otherwise 1.

7. Skip this step if temp_var == 0, else repeat step 6 with roles of \(Q_{ik}^u\) and \(Q_{j\ell}^s\) reversed. Store 0 in temp_var if intersection is rejected, otherwise 1.

8. If temp_var == 1, atomic increment the counter whose pointer was provided as a kernel input.

9. Set the gid entry of the output array to temp_var. Exit kernel.

Once the rejection test kernel has finished, we read the counter value to find out how many true values there are in the output array. This value becomes the size of a new array of 32-bit integers which is given as the true idxs array to the findall kernel from Figure 1. We use the findall kernel to find and store the linear indices of the quadrilateral pairs which might intersect; we then transfer this list of indices from the GPU to the host computer memory, where we convert them to Cartesian indices using the formulas given in kernel step 2. Finally, using these Cartesian indices to find the relevant quadrilaterals, we apply the precise test to any pairs of quadrilaterals which were not rejected by the rejection test kernel. This test is carried out on the CPU, and the resulting intersections are saved as well as the corresponding approximate solutions \((\theta_u, \theta_s, s_u, s_s)\) to Equation (32).

**Computational Results**

The Julia program implementing the previous algorithms was tested on both a laptop as well as on the JPL DGX High Performance Computing platform. The laptop has a 2.9GHz quad core Intel i7 CPU, and an AMD Radeon Pro 560 GPU with 4GB VRAM. On the DGX platform, we assigned 16 CPU threads to Julia, and used a single Nvidia Tesla V100 GPU with 16GB VRAM. The application used for benchmarking the algorithm was the computation of connections between 3:4 \(W^u\) and 5:6 \(W^s\) manifolds in the Jupiter-Europa PERTBP, globalized until layers \(U_{14}\) and \(S_{14}\). We had \(N_1 = 1024\), \(N_2 = 2048\), and \(M_1 = M_2 = 35\). This application was used in our previous study as well, and we achieved the same numerical results and set of mesh intersections using our new Julia GPU/CPU based tools as we did in the previous study using MATLAB CPU-only tools. However, the performance of our new programs massively improves upon our previous study.
Our old MATLAB based tools did not include the Möller test; the only cheap rejection test they carried out was the bounding box test, which took approximately 8 seconds for each pair of half-layers even when using all 4 laptop CPU cores. This was by far the most expensive part of the computation. An earlier version of the Julia code using OpenCL also did not include the Möller test; setting the OpenCL device to the laptop CPU instead of the laptop GPU, the runtime in this case for the bounding box test kernel between two half-layers was also approximately 7.5 seconds. In contrast, running the same exact code after setting the OpenCL device to the laptop GPU resulted in the bounding box test kernel taking only 0.2 seconds! In fact, running all 14 layers through the manifold mesh intersection code using the laptop GPU took about 62 seconds on average. And running the same code on the JPL DGX platform GPU gave a bounding box test kernel runtime of 0.03 seconds, and a total manifold mesh intersection code runtime of approximately 32 seconds.

At this point, the performance-limiting factor became the number of precise tests being carried out, hence the introduction of the Möller test. Implementing the Möller test in the quick rejection test kernel slows down the kernel execution time somewhat, but this is made up for by the decrease in the number of precise tests required, especially for larger layer indices. After implementing the Möller test, the laptop continues to take approximately 62 seconds to complete the manifold mesh intersection search we are using as a benchmark. However, the JPL DGX platform speeds up by a factor of 2, to approximately 16 seconds for the entire intersection search across all 14 layers.

For the sake of completeness, we show the resulting mesh intersections in Figure 5; the plot on the left is the zoomed-in projection onto $(x, y, p_x)$ space of the intersections, shown as yellow circles. The plot on the right is the projection onto $(x, y, p_y)$ space of the same intersections. These figures are repeated from our previous study; as mentioned, there was no change in the numerical results, so newly generated plots of the mesh intersections found using Julia look exactly the same.

**REFINEMENT OF APPROXIMATE MANIFOLD INTERSECTIONS**

We have shown how to represent the unstable and stable manifolds $W^u_1$ and $W^s_2$ as meshes, and have also given fast methods for finding intersections of these meshes in 4D space. However, these meshes are made of triangles, which linearly interpolate points between their vertices. Of course, this interpolation has error, so an intersection of the meshes is not an exact heteroclinic connection.
We now seek to correct the approximate heteroclinic connections found in the mesh-based search from the previous section. We wish to find solutions \( x = (\theta_u, s_u, \theta_s, s_s) \) of the equation
\[
f(x) = f(\theta_u, s_u, \theta_s, s_s) \triangleq W_1^u(\theta_u, s_u) - W_2^s(\theta_s, s_s) = 0 \tag{35}
\]
As discussed during the description of the precise triangle intersection test, we already will have decent initial guesses for \( (\theta_u, s_u, \theta_s, s_s) \) from the mesh search. Hence, we can use differential correction to solve equation (35), but to do that we must be able to differentiate its LHS.

Denote \( \partial_\theta = \frac{\partial}{\partial \theta} \) and \( \partial_x = \frac{\partial}{\partial x} \). To differentiate the LHS of equation (35), we need the partial derivatives \( \partial_\theta W_1^u, \partial_x W_1^u, \partial_\theta W_2^s, \) and \( \partial_x W_2^s \) evaluated at \( (\theta_u, s_u, \theta_s, s_s) \). If \( s_u \notin [-D_u, D_u] \) or \( s_s \notin [-D_s, D_s] \), we cannot just differentiate the Fourier-Taylor parameterizations of the manifolds and evaluate the result at \((\theta_u, s_u, \theta_s, s_s)\). However, the parameterizations are still of use. Applying equation (14) to \( W_1^u \) and equation (15) to \( W_2^s \), we have
\[
W_1^u(\theta, s) = F^m(W_1^u(\theta - m\omega_1, \lambda_u^{-m} s)) \tag{36}
\]
\[
W_2^s(\theta, s) = F^{-n}(W_2^s(\theta + n\omega_2, \lambda_s^n s)) \tag{37}
\]
where \( \omega_1, \omega_2 \) are the rotation numbers of \( W_1^u \) and \( W_2^s \). Differentiating equations (36) and (37) gives
\[
\partial_\theta W_1^u(\theta, s) = DF^m(W_1^u(\theta - m\omega_1, \lambda_u^{-m} s)) \partial_\theta W_1^u(\theta - m\omega_1, \lambda_u^{-m} s) \tag{38}
\]
\[
\partial_x W_1^u(\theta, s) = \lambda_u^{-m} DF^m(W_1^u(\theta - m\omega_1, \lambda_u^{-m} s)) \partial_x W_1^u(\theta - m\omega_1, \lambda_u^{-m} s) \tag{39}
\]
\[
\partial_\theta W_2^s(\theta, s) = DF^{-n}(W_2^s(\theta + n\omega_2, \lambda_s^n s)) \partial_\theta W_2^s(\theta + n\omega_2, \lambda_s^n s) \tag{40}
\]
\[
\partial_x W_2^s(\theta, s) = \lambda_s^n DF^{-n}(W_2^s(\theta + n\omega_2, \lambda_s^n s)) \partial_x W_2^s(\theta + n\omega_2, \lambda_s^n s) \tag{41}
\]
Now, if we choose \( m \) and \( n \) large enough such that \(|\lambda_u^{-m} s_u| < D_u\) and \(|\lambda_u^{n} s_s| < D_s\), then one can use equations (38)-(41) to compute the partials at any \((\theta_u, s_u, \theta_s, s_s)\). Since \( W_1^u \) has a Fourier-Taylor series parameterization valid for \(|s| < D_u\), we can directly evaluate \( W_1^u(\theta_u - m\omega_1, \lambda_u^{-m} s_u) \). We can also differentiate the parameterization with respect to \( \theta \) and \( s \) to get Fourier-Taylor series for \( \partial_\theta W_1^u \) and \( \partial_x W_1^u \), which can then both be evaluated at \((\theta, s) = (\theta_u - m\omega_1, \lambda_u^{-m} s_u)\). Finally, the \( DF^m \) from equations (38) and (39) is just a state transition matrix, found by time-\(2\pi m/\Omega_\theta\) numerical integration of the variational equations starting around \( W_1^u(\theta_u - m\omega_1, \lambda_u^{-m} s_u) \). All this allows us to compute the partials of \( W_1^u \); the partials of \( W_2^s \) are done very similarly.

We now describe how to compute the various quantities on the RHS of equations (38)-(41). Without loss of generality, we describe the process for \( W_1^u \); \( W_2^s \) is done in the same way. Recall from equation (12) that our Fourier-Taylor parameterization \( W_1^u \) is of the form
\[
W_1^u(\theta, s) = \sum_{k \geq 0} W_{1, k}^u(\theta) s^k \tag{42}
\]
The coefficients \( W_{1, k}^u(\theta) \) are stored as arrays of their values at \( N \) evenly spaced \( \theta \) values \( \theta_{u,i} = 2\pi i/N, \ i = 0, 1, \ldots, N - 1 \). Hence, given \((\theta_u, s_u)\), we first evaluate \( W_1^u(\theta_u, \lambda_u^{-m} s_u) \) at all the \( \theta_{u,i} \) using equation (42) and the known coefficients \( W_{1, k}^u(\theta_{u,i}) \). This is followed by a trigonometric interpolation\(^{21}\) to find the value of \( W_1^u(\theta_u - m\omega_1, \lambda_u^{-m} s_u) \) needed in equations (38) and (39), which is used to start the numerical integration of the state transition matrix \( DF^m \).

For the RHS of equation (38), \( \partial_\theta W_1^u(\theta_u - m\omega_1, \lambda_u^{-m} s_u) \) can be found by first using all the \( W_1^u(\theta_u, \lambda_u^{-m} s_u) \) values found earlier to compute \( \partial_\theta W_1^u(\theta_u, \lambda_u^{-m} s_u) \), followed by a trigonometric
interpolation to get the value at \( \theta = \theta_u - m\omega_1 \). To get \( \partial_\theta W_1^u(\theta_{u,i}, \lambda_u^{-m} s_u) \) at all the \( \theta_{u,i} \) from knowledge of \( W_1^u(\theta_{u,i}, \lambda_u^{-m} s_u) \) uses a general FFT based technique. In particular, suppose one has a periodic function \( a(\theta) \) whose values are stored at \( \theta = 2\pi i/N, i = 0, \ldots, N-1 \). Then, using the usual formula relating \( a(\theta_i) \) to its FFT coefficients \( \hat{a}(i) \), we find that

\[
a(\theta_i) = \frac{1}{N} \sum_{k=0}^{N-1} \hat{a}(k) e^{jk\theta_i} \quad \text{and} \quad \partial_\theta a(\theta_i) = \frac{1}{N} \sum_{k=0}^{N-1} [jk\hat{a}(k)] e^{jk\theta_i}
\]

(43)

where \( j \) denotes \( \sqrt{-1} \) in Equation (43). Hence, given the \( a(\theta_i) \) values, to find the values \( \partial_\theta a(\theta_i) \), one takes the FFT, multiplies the \( k \)th Fourier coefficient by \( jk \), and takes the inverse FFT. This completes the tools required to find \( \partial_\theta W_1^u(\theta_u - m\omega_1, \lambda_u^{-m} s_u) \).

For the RHS of equation (39), one can find \( \partial_s W_1^u(\theta_u - m\omega_1, \lambda_u^{-m} s_u) \) by first differentiating equation (42) at each fixed \( \theta_{u,i} \) grid value with respect to \( s \). This gives us \( N \) polynomials in \( s \)

\[
\partial_s W_1^u(\theta_{u,i}, s) = \sum_{k=0}^{N} (k+1) W_1^{u,k+1}(\theta_{u,i}) s^k
\]

(44)

with known coefficients. Next, the \( \partial_s W_1^u(\theta_{u,i}, s) \) series can be evaluated at \( s = \lambda_u^{-m} s_u \) for all the \( \theta_{u,i} \), finally followed by trigonometric interpolation to find \( \partial_\theta W_1^u(\theta_u - m\omega_1, \lambda_u^{-m} s_u) \).

With all quantities from the RHS of equations (38) and (39) found, we can now compute the desired partials at \((\theta_u, s_u)\). As mentioned earlier, the partials of \( W_2^d \) can be found in the exact same manner, after which we can solve equation (35). With \( x = (\theta_u, s_u, \theta_s, s_s) \) and letting \( x_0 \) be the initial guess found earlier for \( x \) solving equation (35), we use the damped Newton method

\[
x_{k+1} = x_k - \alpha Df^{-1}(x_k)f(x_k)
\]

(45)

to differentially correct \( x \) until we have a solution to equation (35) within tolerance. Here, \( Df = [\partial_{\theta_u} f \partial_{s_u} f \partial_{\theta_s} f \partial_{s_s} f] \), and \( 0 < \alpha < 1 \). Trial and error is used to find a value of \( \alpha \) such that the iteration converges. We have used \( \alpha \) values anywhere from 0.5 to 0.01 in our computations.

Using the damped Newton method, we have been able to differentially correct several of the approximate intersections \((\theta_u, s_u, \theta_s, s_s)\) found in the mesh search benchmark described in the previous section, from an error of 0.01 in equation (35) to errors of less than \( 10^{-7} \). An example differential correction is displayed in Figure 6, for one of the connections shown in Figure 5. The initial guess in Figure 6 is shown in green, the iterates in yellow, and the final converged solution in cyan. We changed the value of \( \alpha \) at one point during the iteration, hence the uneven spacing of the iterates. Also, note that the damped Newton iterates move a large distance away from the initial guess found in the mesh search. This is not unexpected; the derivative of the LHS of equation (35) is almost singular, since PCRTBP manifold intersections (and hence solutions to equation (35) in the PCRTBP case) occur along 1D curves rather than at isolated points. In the PCRTBP case, this implies that the derivative at a solution of the equation would actually be singular. The perturbation in the Jupiter-Europa PERTBP is quite weak, so near-singularity is reasonable to expect. It is because of this that the damped Newton method is necessary.

In addition, we found that all of the manifold intersections shown in Figure 5, upon differential correction, actually converged to the same refined solution (the green circle in Figure 6)! This is despite all of the solutions from Figure 5 satisfying equation (35) with an error of 0.01 or less. Hence, we see that intersecting the discrete mesh representations of the manifolds may actually find mostly
Figure 6. 3D projections of iterates of damped Newton method for refinement of an approximate connection found from mesh search

intersections which correspond to near-misses rather than true intersections of the manifolds, especially when the periodic perturbation is weak. This, as well as the distance between the initial guess and the refined solution in Figure 6, demonstrates the importance of carrying out the differential correction in order to find the true intersections. The mesh-based search is necessary in order to quickly narrow down potential points of interest, but the few final accurate intersections must be found by solving equation (35) directly using methods such as those described here.

CONCLUSIONS

In this paper, we presented a suite of concepts, methods, and tools for finding heteroclinic connections between unstable invariant tori in periodically-perturbed PCRTBP models. Using the idea of layers, we can restrict the connection search to only certain subsets of the manifolds. By generating a discrete mesh of points from the manifolds during globalization, one can bring to bear the massively parallel computing power of modern GPUs for the purpose of rapidly detecting and computing intersections of the meshes. Finally, we showed how to refine the solutions found from the mesh search for greater accuracy, using the manifold parameterizations to enable the application of a differential correction algorithm.

Testing our GPU assisted mesh intersection tools, we saw speedups by a factor of over 30 as compared to CPU-only tools. Using an HPC system with a more powerful GPU allows for the checking of 14 layers of two manifolds for intersection in just a matter of seconds. These tools are suitable for exploring many different periodically-perturbed PCRTBP models, as well as the spatial CRTBP, which are both expected areas of future work.

ACKNOWLEDGMENTS

This work was supported by a NASA Space Technology Research Fellowship. This research was carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004). The High Performance Computing resources used in this investigation were provided by funding from the JPL.
REFERENCES


