Abstract: A vehicle trajectory is ergodic with respect to some spatial distribution if the time spent in a region is proportional to the region's integrated density. One method of generating ergodic trajectories is projection-based trajectory optimisation, a gradient descent method that supports non-linear dynamics and balances trajectory ergodicity with control effort. In this study, the authors survey the existing literature on projection-based trajectory generation, focusing on implementation. They introduce an easy-to-use, open-source software package that generates ergodic trajectories orders of magnitude faster than previously reported. The authors’ goal is to simplify the implementation of projection-based ergodic control so it can be applied widely across disciplines.

Nomenclature

Basic

\[ X \subset \mathbb{R}^i \] domain in \( i \) dimensions

\( x \) some point in \( X \)

\( x(t) \) state at time \( t \) according to trajectory \( x \)

\( T \) trajectory duration

\( f \) system dynamics; \( x(t) = f(x(t), u(t)) \)

\( \phi \) distribution over \( X \)

\( c \) spatial distribution representing time-averaged statistics of trajectory

Ergodicity

\( k \) multi-index

\( K_k \) number of Fourier coefficients per dimension

\( F_k \) Fourier basis function

\( \phi_k \) Fourier coefficients of domain

\( \mathbf{\phi}(x), F(\mathbf{x}) \) ergodic metric for trajectory \( x \), ergodic metric for discrete trajectory \( \mathbf{x} \)

\( i \) imaginary unit

\( \mathcal{J} \) Bessel function

Linearisation

\( \delta x(t), \delta u(t) \) perturbations about state \( x(t) \) and control input \( u(t) \), respectively

\( A(t), B(t) \) gradients of \( f \) with respect to state and control evaluated at time \( t \), respectively

Discretisation

\( X_{\text{cells}} \) discrete domain (a set of cells)

\( x_c \) centre of a specific cell in \( X_{\text{cells}} \)

\( V_c \) size of a single cell in \( X_{\text{cells}} \)

\( N \) number of time discrete time steps system is controlled

\( \Delta t \) time step between discrete states

\( \delta x \) discrete trajectory consisting of \( N + 1 \) states \( \{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_N\} \)

\( \hat{u} \) discrete control sequence consisting of \( N \) control inputs \( \{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{N-1}\} \)

\( \delta \mathbf{x}_n, \delta \mathbf{u}_n \) perturbations about the state \( \mathbf{x}_n \) and control input \( \mathbf{u}_n \)

\( A_n, B_n \) gradients of \( f \) with respect to state and control evaluated at timestep \( n \)

\( \mathbf{\hat{A}}_n, \mathbf{\hat{B}}_n \) matrices such that \( \delta \mathbf{x}_{n+1} = \mathbf{\hat{A}}_n \delta \mathbf{x}_n + \mathbf{\hat{B}}_n \delta \mathbf{u}_n \)

Objective

\( J \) objective function

\( J_b \) boundary penalty function

\( q \) weight on ergodic score

\( R \) weight on control effort

\( c_b \) weight on boundary penalty

Optimisation

\( \mathcal{E} \) descent direction for \( \mathbf{x} \); contains a direction \( \mathbf{\xi}_n \) for each state \( \mathbf{x}_n \)

\( \mathcal{V} \) descent direction for \( \mathbf{u} \); contains a direction \( \mathbf{\eta}_n \) for each control input \( \mathbf{u}_n \)

\( \mathbf{\delta} \mathbf{u}_n \) gradient of \( J \) with respect to \( \mathbf{x}_n \) and evaluated at \( \mathbf{x}_n, \mathbf{u}_n \)

\( \mathbf{\bar{u}}_n \) identity matrix with appropriate dimensions

\( Q_D \) weights on state descents in LQ descent direction optimisation; often \( Q_D = I \)

\( R_D \) weight on control input descents in LQ descent direction optimisation; often \( R_D = I \)

\( Q_F \) weight on state deviations in projection operator gains LQR; often \( Q_F = Q_D = I \)

\( R_F \) weight on control deviations in projection operator gains LQR; often \( R_F = R_D = I \)

\( \gamma \) step size

\( \rho \in (0, 1) \) Armijo gradient constant

\( \tau \in (0, 1) \) Armijo step size shrink factor

Other

\( p_x, p_y \) \( x \)-coordinate, \( y \)-coordinate

\( v_x, v_y \) speed in the \( x \)-direction, speed in the \( y \)-direction

1 Introduction

Recently, ergodic control has been applied to robotic agents such as mobile sensors. Ergodic control generates agent trajectories that are ergodic with respect to some spatial distribution. A trajectory is ergodic with respect to a spatial distribution if the time spent in a
region is proportional to the integrated spatial density of that region. Fig. 1 compares an ergodic trajectory to one that maximises time spent in the highest density region. The ergodic trajectory distributes its time between the modes in proportion to their density, whereas the maximising trajectory just moves to the highest density point.

There are a number of robotic applications where an ergodic trajectory is a sensible strategy. One compelling example is active sensing, where a mobile sensor acts to reduce uncertainty in an environment. Examples from the literature include localisation of hidden targets with mobile sensors [1–3]. In these tasks, the spatial sensing, where a mobile sensor acts to reduce uncertainty in an environment. Examples from the literature include localisation of hidden targets with mobile sensors [1–3]. Therefore, the trajectory's coefficients will be zero at all points except those along the trajectory. We require a metric capturing the ergodicity of trajectory $x$ with respect to the distribution $\phi$.

Consider an arbitrary domain $X \subset \mathbb{R}^d$. Let $\phi : X \to \mathbb{R}$ be a distribution over $X$, where $\phi(x)$ is the density at a point $x \in X$. A continuous trajectory over time horizon $T$ is a function $x:[0,T] \to X$. We use $x$ to refer to the entire trajectory and $x(t) \in X$ to refer to the state at the time $t$. We require a metric capturing the ergodicity of trajectory $x$ with respect to the distribution $\phi$.

A popular ergodic metric defined by Mathew and Mezić [12] decomposes $\phi$ and $x$ into Fourier coefficients. The spatial distribution is decomposed into $\prod_{i=1}^{K} (K_i + 1)$ coefficients $\phi_k$, where the vector $k = [k_1, \ldots, k_l]$ is a multi-index used to simplify notation; $\phi_k$ is short for $\phi_{k_1, \ldots, k_l}$ and $\sum_k$ is short for $\sum_{k_1=0}^{K_1} \cdots \sum_{k_l=0}^{K_l}$. The coefficients can be computed with

$$
\phi_k = \int_X \phi(x) F_k(x) dx,
$$

where $F_k$ is the Fourier basis function for the vector $k$. The choice of basis function depends on the distribution's domain; examples are shown in Sections 2.2 and 2.3. To use projection-based trajectory optimisation, the basis function must be differentiable with respect to the state.

We can define a distribution $c$ from the trajectory $x$, where the density at a point $x$ is

$$
c(x) = \frac{1}{T} \int_0^T \delta(x - x(t)) dt,
$$

where $\delta(\cdot)$ is the Dirac delta function. In ergodic control, we are effectively driving the trajectory's time-averaged statistics $c$ to match the spatial distribution $\phi$. For this reason, (2) contains a normalising factor $1/T$ that ensures $c$ integrates to 1. Likewise, $\phi$ must be a proper density that also integrates to 1, so we can compare $c$ and $\phi$. The metric we consider compares $c$ and $\phi$ by breaking both into Fourier coefficients, although another metric uses the KL divergence, a common measure of the similarity of two distributions [7].

We should note that use of the Dirac delta distribution in $c$ means we only consider states in the trajectory. However, there may be situations where the agent footprint is larger than a single state. Consider an autonomous painting task in which the robot has a brush with some thickness, or an information gathering task in which the agent observes all states in some radius of the agent's state. Then it might make sense to define $c$ so it includes all states in the footprint at any given time. Using different agent footprints has recently been explored [7], but the majority of ergodic control work – especially use for information gathering – has used the Dirac delta footprint. Further, the projection-based method is formulated to use the Dirac delta, so we use it here.

Decomposing the distribution $c$ into Fourier coefficients is similar to decomposing $\phi$, but we can leverage the fact that $\delta(\cdot)$ will be zero at all points except those along the trajectory. Therefore, the trajectory's coefficients $c_k$ are

$$
c_k(x) = \frac{1}{T} \int_0^T F_k(x(t)) dt.
$$

The ergodic metric proposed by Mathew and Mezić [12] is the sum of the squared differences between the distribution and trajectory coefficients.
$$\mathcal{E}(x) = \sum_k \Lambda_k \| c_k(x) - \phi_k \|^2. \quad (4)$$

Proceeding, we drop the argument to $c_k$ to simplify the notation. The scaling factor $\Lambda_k$ weights low frequency features more heavily [12] and is defined as

$$\Lambda_k = \frac{1}{(1 + \| k \|_2^{1+d})^{\nu/2}}. \quad (5)$$

The ergodic metric is a functional, or function over functions. Here, $\mathcal{E}$ is a function over trajectory $x$, which is a function yielding sensor state at the time $t$. Minimising trajectory functional is a common problem in optimal control, allowing existing optimisation schemes to be leveraged. Further, $\mathcal{E}$ is differentiable, allowing it to be used in gradient descent techniques. The real definition of trajectory ergodicity is $\mathcal{E}(x) \to 0$ as $T \to \infty$ [12]. However, we call trajectory $x$ ergodic when $\mathcal{E}(x)$ is small, even for a finite horizon $T$.

One way to validate the implementation of the ergodic metric is to construct a distribution $\phi'$ from the trajectory coefficients $c_k$.

$$\phi'(x) = \sum_k c_k F_k(x). \quad (6)$$

If the ergodic metric is low, $\phi(x) \approx \phi'(x)$, $\forall x \in X$ and the distributions should be visually similar.

### 2.2 Decomposition in Euclidean space

Equation (1) shows how to decompose a distribution $\phi$, but we still need to provide a basis function $F_k$. Suppose $\phi$ covers a rectangular domain $X \subset \mathbb{R}^d$ with the lower left corner at the origin so that $X = [0, L_1] \times [0, L_2] \times \ldots \times [0, L_d]$. Mathew and Mezić [12] suggest the basis function

$$F_k(x) = \frac{1}{h_k} \prod_{i=1}^d \cos \left( \frac{k_i x_i}{L_i} \right). \quad (7)$$

where $x_i$ is the $i$th element of point $x$ and $h_k$ is a normalising term defined as [12]

$$h_k = \left( \prod_{i=1}^d \cos \left( \frac{k_i x_i}{L_i} \right) \right)^{1/2}. \quad (8)$$

For example, if the domain is two-dimensional (2D) (exists in $\mathbb{R}^2$), then

$$h_k = \left( \int_0^{L_1} \int_0^{L_2} \cos \left( \frac{k_1 x_1}{L_1} \right) \cos \left( \frac{k_2 x_2}{L_2} \right) \, dx \, dx \right)^{1/2}. \quad (9)$$

Trajectory generation requires computing the gradient of this basis function. If the domain exists in $\mathbb{R}^2$, the gradient is

$$\mathcal{N} F_k(x) = \frac{\pi}{h_k} \begin{pmatrix} \frac{k_i}{L_i} \sin \left( \frac{k_i x_i}{L_i} \right) \cos \left( \frac{k_i x_i}{L_i} \right) \\ \frac{k_i}{L_i} \cos \left( \frac{k_i x_i}{L_i} \right) \sin \left( \frac{k_i x_i}{L_i} \right) \end{pmatrix}. \quad (9)$$

### 2.3 Decomposition in SE(2)

The special Euclidean group SE(2) contains the set of planar translations and rotations. A state in SE(2) consists of a 2D position and a heading $\theta$ measured north of west. Spatial distributions over SE(2) are common in mobile sensing tasks, where the measurement model depends on robot position and orientation. The mobile sensor might be a ground vehicle or a flying robot with an orientation-dependent sensor like a camera; what the sensor sees depends on the vehicle's location and orientation. Then the expected information density covers a region in SE(2), denoting the value of each robot position-orientation combination.

Miller and Murphey [9] suggest using the following basis function for SE(2) domains:

$$F_k(x) = \Re^{\nu/2} \exp(ik_3 \psi + (k_1 - k_3)\theta)]J_{k_1, -k_3}(k_r), \quad (10)$$

where $i$ is the imaginary unit, $J$ is the $(k_1 - k_3)$-th order Bessel function and $(r, \psi, \theta)$ is the agent’s state expressed in polar coordinates. That is, if the state $x = [p_r, p_\psi, \theta]^T$, then

$$r = \sqrt{p_r^2 + p_\psi^2}, \quad \psi = \arctan(p_\psi / p_r), \quad \theta = \theta. \quad (11)$$

When using distributions over $\mathbb{R}^d$, $c_k$ and $\phi_k$ are scalars, so the norm in (4) simply functions as an absolute value. However, the basis function for SE(2) defined in (10) returns complex numbers. Therefore, when using distributions over SE(2), the Euclidean norm in complex numbers should be used [9]

$$\| c_k - \phi_k \| \approx \sqrt{\Re(c_k - \phi_k)^2 + \Im(c_k - \phi_k)^2}. \quad (12)$$

where the functions $\Re$ and $\Im$ yield the real and imaginary parts of their arguments, respectively.

The multi-index $k$ takes a slightly different meaning in the context of SE(2) [13]. The indices $k_1$ and $k_2$ take integer values between $-K_1, \ldots, 0, \ldots, K_1$ and $-K_2, \ldots, 0, \ldots, K_2$. In theory, $k_3$ can take any non-negative value. To limit the number of coefficients used, we limit $k_3$ to positive integers up to some $K_3$.

The inverse Fourier transform used to reconstruct a distribution from coefficients $\phi$ is also more complex than in the Euclidean case

$$\phi(x) = \sum_{k_1, k_2, k_3} \prod_{i=1}^3 \varphi F_k(x) \mathcal{N} F_k(x) \, dk_3. \quad (13)$$

where $\sum_{k_1, k_2}$ is the sum over all $k_1$ and $k_2$, and $\mathcal{N} F_k(x)$ is the complex conjugate of $F_k$. If we limit $k_3$ to positive integers as described above, the inverse transform simplifies to

$$\phi(x) = \sum_{k_1, k_2} \varphi F_k(x) \mathcal{N} F_k(x). \quad (14)$$

We found the approximation introduced by limiting $k_3$ to integer values to be reasonable – reconstructed distributions roughly matched the originals.

To find the gradient with respect to state $x = [p_r, p_\psi, \theta]^T$, we start with the partial derivatives of each state variable

$$\frac{\partial F_k(x)}{\partial p_r} = \Re^{\nu/2} \exp(\psi) \frac{\partial \psi}{\partial \psi} (J_{k_1, -k_3}(k_3 \psi + (k_1 - k_3)\theta)) \frac{\partial J_{k_1, -k_3}(k_r)}{\partial k_r} + J_{k_1, -k_3}(k_r) \exp(\psi) \frac{\partial \psi}{\partial \psi} \frac{\partial J_{k_1, -k_3}(k_r)}{\partial k_r} \quad (15)$$

where $\exp(\cdot) = \exp(k_3 \psi + (k_1 - k_3)\theta)$ and the derivatives are

$$\frac{\partial J_{k_1, -k_3}(k_r)}{\partial k_r} = \frac{1}{2} \left( J_{k_1, -k_3}(k_r) - J_{k_1, -k_3}(k_r) \right),$$

$$\frac{\partial \psi}{\partial \psi} = 0.$$  \quad (16)$$

The derivative $\partial F_k(x)/\partial p_r$ is identical to (15) except $\partial \psi / \partial \psi$ is replaced with $\partial \theta / \partial \theta$ and $\partial \psi / \partial \psi$ is replaced with $\partial \theta / \partial \theta$. 

---

This is an open access article published by the IET under the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/)
The partial derivative with respect to \( \theta \) is
\[
\frac{\partial F_k(x)}{\partial \theta} = f_{k_2-k_1}(x; \theta) \exp(\cdot)(k_2 - k_1).
\] (18)

Putting together the partial derivatives, the gradient of the SE(2) basis function is
\[
\nabla F_k(x) = \begin{bmatrix} \frac{\partial F_k(x)}{\partial p_x} & \frac{\partial F_k(x)}{\partial p_y} & \frac{\partial F_k(x)}{\partial \theta} \end{bmatrix}^T.
\] (19)

2.4 Discretisation

So far, we have discussed ergodicity in continuous domains and trajectories. However, most real systems are discretely implemented; consider active sensing tasks, where sensors often output discrete measurements at some sampling frequency. We want these measurement locations to be ergodic. Furthermore, discretising will enable faster trajectory generation. Therefore, we focus on discrete trajectories.

Computational improvements can also come from discretising the density \( \phi \). The domain can be discretised into a set of cells \( X_{\text{cells}} \). We assume each cell has the same size \( V_c \), with the notion of size depending on the number of dimensions in \( X \) (size is width in 1D, area in 2D, volume in 3D etc.). Denoting the centre point of a cell as \( x_c \), the Fourier coefficient \( c_k \) can then be approximated as
\[
\phi_k \approx \sum_{x_c \in X_{\text{cells}}} \phi(x_c) F_k(x_c) V_c.
\] (20)

Just as with the continuous case, the distribution must be normalised so that \( \sum_{x_c \in X_{\text{cells}}} \phi(x_c) V_c = 1 \). Of course, the approximation improves as the number of cells increases, but increasing the number of cells increases computational expense.

Trajectories are often discretised in path planning and control problems. We use \( \tilde{x} \) to denote a discrete trajectory consisting of \( N + 1 \) discrete points with \( \Delta t \) time between them. We use \( \tilde{x}_n \in X \) to denote the state at timestep \( n \) (or time \( n\Delta t \)), for \( n = 0, \ldots, N \). The set of \( N \) discrete control inputs is denoted \( \tilde{u} \); for all \( N = 0, \ldots, N - 1 \), \( \tilde{u}_n \) is the control input applied at \( t = n\Delta t \) for a duration of \( \Delta t \).

Using this trajectory discretisation, the trajectory Fourier coefficients \( c_k \) can be defined. Prabhakar et al. [10] suggested using a left Riemann sum
\[
c_k(\tilde{x}) = \frac{1}{N} \sum_{n=0}^{N-1} F_k(\tilde{x}_n) \Delta t.
\] (21)

We prefer a slightly different discretised approximation for \( c_k \):
\[
c_k(\tilde{x}) = \frac{1}{N+1} \sum_{n=0}^{N} F_k(\tilde{x}_n).
\] (21)

This version makes explicit that we are looking for an ergodic distribution of discrete points. This makes sense in certain contexts, such as information gathering, where mobile sensors often receive measurements discretely at some frequency. We also add the final state \( \tilde{x}_N \) to the ergodic score; otherwise, there is no reason to apply any control at a time step \( N - 1 \), in which case \( \tilde{x}_{N-1} \) and \( \tilde{u}_{N-2} \) are the last state and control inputs that matter in the optimisation, effectively reducing the horizon to \( N - 1 \). However, some users might want to remove the first or final states from the ergodic score. For example, if no measurement is made in the first state (perhaps measurements are only made after moving to a state), we might not count it towards the ergodicity. It is possible to remove the first or final states from the ergodic metric, as long as the sum in (21) is divided by the total number of points used in the sum.

3 Trajectory generation
3.1 System dynamics

When designing a trajectory, the system's dynamics need to be satisfied. We describe a system's dynamics by showing how the system state evolves in time as a function of state and control input
\[
x(t) = f(x(t), u(t)).
\] (22)

The dynamics can also have an explicit time dependence, but we ignore it here without loss of generality.

In discrete trajectories with points temporally spaced \( \Delta t \) apart, the state update is
\[
\tilde{x}_{n+1} = \tilde{x}_n + f(\tilde{x}_n, \tilde{u}_n) \Delta t.
\] (23)

This update is the forward Euler method, an explicit integration scheme. It is efficient to compute but can introduce errors, as we discuss in Section 5.3.

In order to optimise trajectories efficiently, it sometimes helps to linearise about a nominal trajectory. Suppose we have a continuous trajectory \( x \) and with the corresponding control sequence \( u \). We can linearise about this trajectory to see how perturbations – small changes in the trajectory – propagate. The perturbations \( \delta x(t) \) and \( \delta u(t) \) are small changes in the state and control at the time \( t \). If the perturbations are small, we can approximate their propagation as
\[
\delta x(t) = A(t)(\delta x(t)) + B(t)(\delta u(t)),
\] (24)

where \( A(t) \) and \( B(t) \) represent the linearisation at the time \( t \):
\[
A(t) = \nabla_x f(x(t), u(t)),
\]
\[
B(t) = \nabla_u f(x(t), u(t)).
\] (25)

Note that \( f \) is a vector-valued function, so \( \nabla_x f \) is a matrix whose entry in row \( i \) and column \( j \) is
\[
(V_x f)_{ij} = \frac{\partial f_i}{\partial x_j}.
\] (26)

We use the subscript on \( V \) to denote which argument to differentiate with respect to; we drop the subscript if there is only one argument.

We can write a perturbation update for the discrete case using the forward Euler method
\[
\delta x_{n+1} = \delta x_n + (A_n(\delta x_n) + B_n(\delta u_n)) \Delta t,
\] (27)

where \( A_n \) and \( B_n \) are simply evaluated at the time \( n\Delta t \):
\[
A_n = \nabla_x f(\tilde{x}_n, \tilde{u}_n),
\]
\[
B_n = \nabla_u f(\tilde{x}_n, \tilde{u}_n).
\] (28)

We can simplify the discrete perturbation update
\[
\delta \tilde{x}_{n+1} = \tilde{A}_n(\delta \tilde{x}_n) + \tilde{B}_n(\delta \tilde{u}_n),
\] (29)

where the matrices \( \tilde{A}_n \) and \( \tilde{B}_n \) are
\[
\tilde{A}_n = I + \Delta t A_n,
\]
\[
\tilde{B}_n = \Delta t B_n,
\] (30)

and \( I \) is the appropriately-sized identity matrix.

3.2 Objective and constraints

The goal in ergodic trajectory generation is to produce trajectories that yield a low ergodic metric while using little control effort. This goal is captured with the objective...
where \( q \) weights the ergodic metric, \( R \) weights control effort and \( J_0 \) is an optional boundary penalty applied if the sensor leaves the domain. In practice, the boundary penalty is often needed. The issue arises from the periodic nature of the basis functions in the ergodic metric [3, 10]. A result of this periodicity is that a trajectory can achieve a low ergodic cost while spending no time in the search domain over which \( \phi \) is defined. This phenomenon is well described in Section 3.7.3 of [3].

A quadratic barrier function is briefly suggested in previous work [3]. Instead of constraining the trajectory to remain in the domain, we penalise deviations from the domain [14]. To keep the trajectory in a rectangular domain \( X = [0, L] \times [0, L] \times \cdots \times [0, L_n] \), we might use the quadratic penalty

\[
J_0(\tilde{x}) = c_0 \sum_{n=1}^{N-1} \left( \max (\tilde{x}_n - L_n, 0)^2 + \min (\tilde{x}_n, 0)^2 \right).
\]

where \( c_0 > 0 \) weights the boundary penalty and \( \tilde{x}_n \) is the \( n \)th element of state \( x_\phi \).

The weights \( q, R, \) and \( c_0 \) are chosen for the desired performance. When \( q \) is large compared to the elements of \( R \), a preference is given to minimising the ergodic metric. When the elements of \( R \) are large compared to \( q \), a preference is given minimising control effort. If the resulting trajectory leaves the domain, \( c_0 \) can be increased.

There are two constraints on the optimisation. The first is that the trajectory adheres to the vehicle's dynamics. The second constraint is that the first state of the resulting trajectory must be \( x_\phi \). The optimisation problem can then be defined as

\[
\begin{align*}
\text{minimise } & J(\tilde{x}, \tilde{u}) \\
\text{subject to } & \tilde{x}_{n+1} = \tilde{x}_n + f(\tilde{x}_n, \tilde{u}_n) \Delta t, \\
& n = 0, \ldots, N-1; \quad \tilde{x}_0 = x_\phi.
\end{align*}
\]

### 3.3 Solution method

Unfortunately, the optimisation problem defined in (31) is not convex and there exists no efficient way to find the global optimum. However, we can use an iterative gradient descent method – specifically, a projection-based descent. The projection method is attractive because nonlinear constraints are removed during calculation of the descent direction, making the computation much simpler. However, the resulting trajectory might not be dynamically feasible, so it is projected back into feasible space.

Before starting the optimisation, the spatial distribution of \( \phi \) must be decomposed into Fourier coefficients \( \phi_n \). This decomposition can be computationally expensive and must be included when evaluating the time to generate a trajectory. After spatial decomposition, the optimisation is started with an initial trajectory and control sequence, which are iteratively modified until a local minimum in the objective function is reached. Often, this initial trajectory is generated by applying a constant control input. However, the initial trajectory can change the local minimum reached, so it can make sense to run the optimisation from different initial trajectories. Each iteration consists of five steps. First, the matrices \( \tilde{A}_n \) and \( \tilde{B}_n \) are computed for each state and control in the current trajectory. Second, the descent direction is computed. Third, gains for the projection operator are computed. Fourth, a descent step size \( \gamma \) is selected. Finally, the step size and descent direction are combined with the current trajectory to yield a candidate trajectory. This candidate might not be dynamically feasible, so it is projected into a feasible trajectory. This trajectory is then modified in the next iteration unless a convergence criterion is met. This criterion is typically a directional derivative threshold.

### 3.4 Descent direction

The descent direction describes how the current trajectory and control sequence should change to decrease the objective function. The trajectory descent direction is denoted \( \tilde{z}_n \) and consists of \( N + 1 \) directions \( \tilde{z}_n \), one for each state \( \tilde{x}_n \) in \( \tilde{x} \). Likewise, \( \tilde{v} \) is the descent direction for the control sequence \( \tilde{u} \), with one direction \( \tilde{v}_n \) for each input \( u_n \). The total descent direction \( (\tilde{z}, \tilde{v}) \) is derived from the gradient of the objective and can solve with the following linear quadratic (LQ) problem:

\[
\begin{align*}
\text{minimise } & \sum_{n=0}^{N} \left( \tilde{B}_n^\top \tilde{z}_n + \tilde{v}_n^\top \tilde{B}_{n+1} \tilde{z}_n + \tilde{z}_n^\top \tilde{Q}_n \tilde{z}_n + \tilde{v}_n^\top \tilde{R}_{n+1} \tilde{v}_n \right) \\
& + \tilde{z}_N^\top \tilde{Q}_N \tilde{z}_N, \\
\text{subject to } & \tilde{z}_{n+1} = \tilde{A}_n \tilde{z}_n + \tilde{B}_n \tilde{v}_n, \\
& n = 0, \ldots, N-1; \quad \tilde{z}_0 = \tilde{0}.
\end{align*}
\]

where \( \tilde{A}_n \) and \( \tilde{B}_n \) are the gradients of the objective \( J \) with respect to state \( \tilde{x}_n \) and control \( \tilde{u}_n \), respectively. Each element in the initial state descent direction \( \tilde{z}_0 \) is constrained to zero because the sensor's starting state is fixed and cannot be changed. The descent direction is constrained to the perturbational dynamics from (29) because the descent direction is effectively a perturbation from the current trajectory.

Equation (34) has been theoretically justified [3, 8, 15], but we briefly provide some intuition here. Our goal is to perform gradient descent on the objective in (31). Because the gradient points in the direction of greatest increase of the function, we should step in the opposite direction, which is \( -\tilde{A} \) and \( -\tilde{B} \). However, the resulting trajectory might not satisfy the agent's dynamics. Therefore, we start with the optimisation problem of minimising \( \sum \tilde{B}_n^\top \tilde{z}_n + \tilde{v}_n^\top \tilde{R}_{n+1} \tilde{v}_n \), which is minimised when \( [\tilde{z}_n] \) and \( [\tilde{v}_n] \) point along the negative gradients. Generally, imposing nonlinear constraints on an optimisation problem makes it much harder to solve, so we apply the linear constraint from (29), assuming the descent direction is a perturbation about the current trajectory. Later, a projection operator will correct any errors by imposing the dynamic's constraints. The objective \( \sum \tilde{B}_n^\top \tilde{z}_n + \tilde{v}_n^\top \tilde{R}_{n+1} \tilde{v}_n \) is unbounded below because the linear constraint does not prevent the descents from being arbitrarily scaled. Therefore, we penalise the descent magnitude with the quadratic cost \( \sum \tilde{z}_n^\top \tilde{Q}_n \tilde{z}_n + \tilde{v}_n^\top \tilde{R}_{n+1} \tilde{v}_n \). We assume the positive semi-definite matrix \( \tilde{Q}_n \) and positive definite matrix \( \tilde{R}_n \) are constant instead of using \( Q_0 \) and \( R \) that depend on time step \( n \) in most cases; there is no reason to vary these weights with time. It is tempting to choose \( Q_0 \) and \( R_0 \) to reflect the weights on ergodic and control costs, but the weights \( q \) and \( R \) already show up in \( \tilde{a}_n \) and \( \tilde{b}_n \), respectively. In practice, \( Q_0 \) and \( R_0 \) are usually set to identity matrices.

Let us now explicitly define the gradient \( \tilde{a}_n \):
The boundary penalty gradient with respect to $\tilde{x}_n$ is

$$\nabla_{\tilde{x}_n} J_b(\tilde{x}) = 2\beta \sum_{i=1}^{N} \max(\tilde{x}_{n,i} - L_i, 0) + \min(\tilde{x}_{n,i}, 0). \quad (37)$$

This gradient is zero if $\tilde{x}_n$ is within the domain.

The control gradient $\tilde{b}_n$ is

$$\tilde{b}_n = \Delta t R\tilde{u}_n. \quad (38)$$

Once the gradients are computed, a Riccati-like backwards iteration can be used to compute the solution to the LQ problem

$$P_n = Q_n, \quad r_n = \tilde{a}_n, \quad \Gamma_n = R_n + \tilde{B}_n^T P_{n+1} \tilde{B}_n, \quad \mathcal{H}_n = \Gamma_n \tilde{B}_n^T P_{n+1} \Gamma_n, \quad P_{n+1} = Q_n + \tilde{A}_n^T P_{n+1} \tilde{A}_n - \mathcal{H}_n \Gamma_n \mathcal{H}_n, \quad r_{n+1} = \tilde{a}_{n+1} + \tilde{A}_n \tilde{x}_{n+1} + \tilde{B}_n \tilde{v}_n, \quad n = 0, 1, \ldots, N - 1. \quad (39)$$

Once the Riccati gains are computed, the descent directions for each individual state and control input are

$$\tilde{z}_n = 0, \quad \tilde{v}_n = -\mathcal{H}_n^{-1}(\tilde{b}_n + \tilde{B}_n^T r_{n+1}), \quad (40)$$

$$\tilde{z}_{n+1} = \tilde{A}_n \tilde{x}_n + \tilde{B}_n \tilde{v}_n, \quad n = 0, 1, \ldots, N - 1. \quad (41)$$

### 3.5 Projection operator

For the moment, let us assume we have a step size $\gamma > 0$ and ignore the process by which we attain this step size. This step size can be used with the descent direction $(\tilde{z}, \tilde{v})$ to create a candidate trajectory $\tilde{a}$ and candidate control sequence $\tilde{u}$:

$$\tilde{a} = \tilde{x} + \gamma \tilde{z}, \quad \tilde{u} = \tilde{u} + \gamma \tilde{v}. \quad (42)$$

The candidate trajectory and control sequence might not be feasible – they might not satisfy the system dynamics if the system is non-linear. Therefore, the candidate $(\tilde{a}, \tilde{u})$ is projected into a feasible space with the projection operator $\mathcal{P}$:

$$\mathcal{P}(\tilde{a}, \tilde{u}, \tilde{x}_0): \begin{cases} \tilde{u}_0 = \tilde{u}_0 + \mathcal{P}(\tilde{a}_0 - \tilde{x}_0) \\ \tilde{x}_{n+1} = \tilde{x}_n + f(\tilde{x}_n, \tilde{u}_n) \Delta t, \quad n = 0, \ldots, N - 1. \end{cases} \quad (43)$$

where the gains $\mathcal{H}_n$ come from an LQ regulator (LQR) problem. If the agent’s dynamics are non-linear, the projected trajectory will deviate from the candidate trajectory. However, the candidate trajectory was computed to reduce the objective function so we want to minimise these deviations. If we use $\delta \tilde{x}$ and $\delta \tilde{u}$ to denote deviations from the candidate trajectory and control sequence at a time step $n$, we can write the following LQR problem:

$$\begin{align} \min_{(\delta \tilde{x}_n, \delta \tilde{u}_n)} & \sum_{n=0}^{N-1} \left( \delta \tilde{x}_n^T Q_n \delta \tilde{x}_n + \delta \tilde{u}_n^T R_n \delta \tilde{u}_n \right) \\ + & \delta \tilde{x}_N^T Q_n \delta \tilde{x}_N \\ \text{subject to} & \delta \tilde{x}_{n+1} = \tilde{A}_n \delta \tilde{x}_n + \tilde{B}_n \delta \tilde{u}_n, \\ & n = 0, \ldots, N - 1; \\ & \delta \tilde{x}_0 = 0. \end{align} \quad (44)$$

The gains $\mathcal{H}_n$ can be solved with the same Riccati equations used for the LQ problem, but substituting the projection weights $Q_R$ and $R_F$ for the descent weights $Q_D$ and $R_D$. If $Q_D = Q_F$ and $R_D = R_F$, the resulting gains $\mathcal{H}_n$ from the LQ and LQR problems will be equal, eliminating the need to actually solve the LQR problem. Setting $Q_D$ and $R_F$ to identity matrices usually provides acceptable results [16], and we suggested setting the LQ gains to identity matrices in Section 3.4; therefore, we arrive at $Q_D = Q_F$ and $R_D = R_F$ in practice. Regardless of how the LQR gains $\mathcal{H}_n$ are obtained, they are then plugged into the projection operator from (42).

Recall that matrices $A_n$ and $B_n$ are linearisations about the trajectory at the beginning of the current iteration. Let us call this the nominal trajectory. In the projection operator, we are not minimising perturbations about this trajectory but rather the candidate trajectory. However, we assume this candidate is just a perturbation about the nominal trajectory and that the linearisation is still valid [16, 17].

If the system dynamics are linear, the candidate trajectory will be feasible, because the descent (LQ) problem uses linearised dynamics. The candidate trajectory will then be a fixed point of $\mathcal{P}$; the projected trajectory equals the input trajectory [15]. Therefore, the projection is unnecessary if the agent has linear dynamics.

### 3.6 Step size and termination

Once a descent direction is found, a step size is needed. There exist simple ways to pick a step size; if $\gamma$ is an arbitrary constant and $i$ is the optimisation iteration, then $\gamma = \gamma / i$ or $\gamma = \gamma / \sqrt{i}$ will converge to a local optimum. These simple options are efficient to compute but might lead to excessive iterations.

Backtracking line search has been used effectively in existing projection-based ergodic trajectory research [1, 3, 8–10]. The candidate step size is initialised to $\gamma_0$ and multiplied by a constant $\tau \in (0, 1)$ each iteration. For example, $\tau = 0.5$ halves the candidate step size each iteration. Iteration terminates when the Armijo condition [18] is reached. This condition is

$$J(\mathcal{P}(\tilde{x} + \gamma \tilde{z}, \tilde{u} + \gamma \tilde{v}, \tilde{x}_0)) \leq J(\tilde{x}, \tilde{u}) + \rho \gamma \nabla J(\tilde{x}, \tilde{u}) \cdot (\tilde{z}, \tilde{v}), \quad (44)$$

where $\rho \in (0, 1)$ is an arbitrary constant and the directional derivative, or slope, along with the descent direction is

$$\nabla J(\tilde{x}, \tilde{u}) \cdot (\tilde{z}, \tilde{v}) = \sum_{n=0}^{N} \tilde{a}_n^T \tilde{z}_n + \sum_{n=0}^{N-1} \tilde{b}_n^T \tilde{v}_n. \quad (45)$$

This backtracking search is an inexact line search, meant to sufficiently find a step size that is ‘good enough’ and sufficiently decreases the objective function. The idea is to start with a large step size and reduce it until the Armijo condition is satisfied; this process is guaranteed to lead to convergence in the trajectory optimisation [17]. With enough iterations, the Armijo condition will eventually be reached. If the value of $\rho$ were 1, the right side of (44) would simply be a first-order approximation of the left side. Because $\rho < 1$, the right side of (44) will be larger than the left for sufficiently small $\gamma$ [14].

Once the step size $\gamma$ has been selected, the descent is carried out and projected into feasible space, and we assign the resulting trajectory as our new $\tilde{x}, \tilde{u}$:

$$\tilde{x}, \tilde{u} := \mathcal{P}(\tilde{x} + \gamma \tilde{z}, \tilde{u} + \gamma \tilde{v}, \tilde{x}_0). \quad (46)$$

A first-order necessary condition for local optimality is that the directional derivative in the descent direction (45) equal zero. Therefore, iterations can be carried out until the magnitude of this term falls below some threshold [3]; thresholds on the order of $1 \times 10^{-3}$ have been used [8]. Alternate termination conditions include maximum iterations or ergodic thresholds. However, there is no guarantee that a specified ergodic threshold will be reached – due vehicle dynamics or the spatial distribution, it is possible that any feasible trajectory will have a high ergodic cost.
can be run interactively and plotted easily. Below is some example code that could be run as a script or entered line by line into the Julia shell. Fig. 2 shows the resulting plot.

```
using ErgodicControl
# domain and distribution
d = Domain([1], 100)
phi = gaussian(d, [0.5, 0.5], 0.03*eye(2))
K = 5
em = ErgodicManagerR2(d, phi, K)
# trajectory params
x0 = [0.49, 0.01, 0.0, 0.0]
dt = 0.5
N = 50
tm = TrajectoryManager(x0, dt, N)
dynamics!(tm, DoubleIntegrator(2, dt))
# trajectory generation and plotting
xd, ud = pto_trajectory(em, tm)
plot(em, xd)
```

This code is only an example, and the package might change in the future. Current code and documentation can be found at https://github.com/dressel/ErgodicControl.jl.

### 4 Software package and examples

We present ErgodicControl.jl, a software package that generates ergodic trajectories using the methods described in this paper. This package is written in the Julia programming language, a high-level programming language with good performance [11]. As a result, our software package offers two major benefits.

The first benefit is ease of use. In contrast to C code, Julia code can be run interactively and plotted easily. Below is some example code that could be run as a script or entered line by line into the Julia shell. Fig. 2 shows the resulting plot.

```
using ErgodicControl
# domain and distribution
d = Domain([1,1], 100)
phi = gaussian(d, [0.5,0.5], 0.03*eye(2))
K = 5
em = ErgodicManagerR2(d, phi, K)
# trajectory params
x0 = [0.49, 0.01, 0.0, 0.0]
dt = 0.5
N = 40
tm = TrajectoryManager(x0, dt, N)
dynamics!(tm, DoubleIntegrator(2, dt))
# trajectory generation and plotting
xd, ud = pto_trajectory(em, tm)
plot(em, xd)
```

Fig. 2 Trajectory generated with provided example code

Fig. 3 Effect of domain discretisation level on trajectory generated. Left domain discretised into cells and right domain discretised into cells. The resulting trajectories are essentially identical.

4.1 Speed comparison

To compare the speed with existing software, we use a double integrator example presented by Prabhakar et al. [10]. This example was a 30 s trajectory with time steps of $\Delta t = 0.6$ s designed to be ergodic with a double Gaussian in $\mathbb{R}^2$. The ergodic and control weights were $q = 1$ and $R = 0.011I$. The Fourier decomposition used $K_1 = K_2 = 5$ coefficients. This number seems to capture the spatial distribution well while keeping computation reasonable so we use the same number in all examples shown in this paper.

Prabhakar et al. [10] report that optimising discrete trajectories is much faster than optimising continuous trajectories because simple Riemann sums are easier to compute than more complicated integration schemes. However, we also discretise the domain into 100$^2$ cells and integrate with (20). Prabhakar et al. [10] report around 50 s to run 100 iterations of the gradient descent, while our code performs 100 iterations in roughly 0.2 s – over two orders of magnitude faster. We also only run 100 iterations to compare with previous work; the trajectory reaches a directional derivative criterion of $1 \times 10^{-3}$ after 43 iterations and 0.09 s. The resulting trajectory looks similar to those in Fig. 3.

Much of this speed improvement is likely due to our use of Julia, whereas previous work used Mathematica [10]. We might see further improvements with a C++ implementation. Further, there are other methods for generating ergodic trajectories that do not use projection-based optimisation and can be performed more quickly. However, our speed improvement proves projection-based ergodic trajectories can be generated in real time. To our knowledge, there are no other software packages that achieve this. As a result, users can now generate projection-based ergodic trajectories for applications that require computing trajectories in real time.

However, there is still room for improvement. For example, trajectory generation in SE(2) is much slower, because of the computationally expensive Bessel function. Trajectory generation for an SE(2) example with the domain split into 50$^2$ cells takes roughly 3 s; half is spent on the domain’s Fourier decomposition and half is spent performing 56 iterations to convergence. This time is too slow to compute in real time, and future work might try to minimise the effect of the Bessel function calls.

4.2 Effect of domain discretisation

Domain discretisation is one factor in our code’s improved performance. A reasonable concern is the effect of this discretisation – does the resulting coarse numerical integration over the domain affect the trajectory generated? To investigate, we performed the same example from above with 10$^2$ cells and 100$^2$ cells. The resulting trajectories can be seen in Fig. 3. It does not seem that the discretisation level has much effect on the trajectory. However, the time required to compute the 100$^2$ trajectory is longer; 100 iterations took 0.35 s to compute, about half of which was spent decomposing the spatial distribution. In contrast, both the 10$^2$ and 100$^2$ trajectories took 0.2 s, and almost none of this time was spent decomposing the distribution because the numerical integration was much coarser. Although this test suggests domain discretisation level has little effect on the resulting trajectory, it is not conclusive; discretisation might introduce noticeable errors with distributions that are less smooth or when more Fourier coefficients are used.

4.3 Boundary penalty

A final example shows the importance of the boundary penalty in certain cases. We generate a trajectory for a Dubins car [20] with a velocity of 0.3 m/s and a minimum turn radius of 0.1 m. The car compute times are especially detrimental in tasks that require recomputing trajectories in real time. Recomputing trajectories in a model-predictive fashion has been suggested for active sensing tasks like localisation; the agent plans a trajectory based on the current expected information density and recomputes as new measurements are made and the information density changes [2].
operates in a 1 m by 1 m square discretised into 100² cells. Trajectories are generated to be ergodic with the two Gaussian distributions from previous examples. Fig. 4 shows the trajectories generated identically except that on the left, no boundary penalty is applied. In this case, the car leaves the domain because applying control (rotating the car) has some cost; the car can instead apply no control and leverage the periodic basis functions. The basis functions make it seem that Gaussian distributions are mirrored across the boundary; by applying no control, it can traverse these mirrored distributions and achieve an ergodic trajectory. But when the penalty from (32) is applied and weighted with $c_k = 1$, the car stays in the domain.

5 Additional implementation details

We have covered the basic process of generating ergodic trajectories, but there are a number of details and extensions that are relevant to certain problems. We cover some of these here.

5.1 Workspace and agent state space

So far we have ignored an important detail concerning the problem domain. Sometimes, the agent’s state space differs from the domain over which our distribution is defined. That is, our agent has some state space $X_a$ such that $X_a \neq X$. Then the agent trajectory is a function $x : [0, T] \rightarrow X_a$; in the discrete case, we can say $\tilde{x}_a \in X_a \neq X$. This mismatch is not uncommon. For example, consider a distribution over a domain $X \subset \mathbb{R}^2$ and an agent with double integrator dynamics in $\mathbb{R}^2$. The agent state space $X_a \subset \mathbb{R}^2$ consists of two position and two velocity state variables. This scenario might occur in a mobile sensing task where measurements only depend on the agent’s 2D position, but we need the velocities to propagate the agent’s dynamics. Another example is a Dubins car with trajectories ergodic to a distribution over $\mathbb{R}^2$ — the car’s state has 3D (2D position plus heading) whereas the distribution’s domain consists only of a 2D position. We call $X$ the workspace or distribution domain and $X_a$ the agent’s state space.

In the above examples, $X_a$ is just an augmentation of $X$; velocity or heading variables are simply added to the position variables. However, in other applications, the mismatch is more significant. Consider a robotic arm with several links. We might want the end effector of this arm to move ergodically with respect to some effector of this arm to move ergodically with respect to some.

To rectify the mismatch between $X$ and $X_a$, we need a differentiable function $g : X_a \rightarrow X$ mapping the agent’s state space to the workspace. In the double integrator example, with agent state $x(t) = [p_x, p_y, \dot{p}_x, \dot{p}_y]^T$, the mapping is

$$g(x(t)) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} x(t),$$

leaving only the position variables. Likewise, a Dubins vehicle with the state $x(t) = [p_x, p_y, \theta]^T$ would have the following mapping:

$$g(x(t)) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} x(t).$$

The joint example is more complicated and involves solving the Euler–Lagrange equations, but a relation $g(x(t))$ can be derived [3].

The agent states must be mapped into the distribution domain before feeding them to the ergodic metric. Instead of $\delta(\tilde{x})$ and $c_k(\tilde{x})$, we have $\delta(g(\tilde{x}))$ and $c_k(g(\tilde{x}))$, where $g(\tilde{x})$ applies $g$ to each state $\tilde{x}_a$ in $\tilde{x}$. An extra term appears in the ergodic gradient

$$V_{\tilde{x}}\delta(g(\tilde{x})) = \frac{2}{N+1} \sum_{k} \Lambda_k (\tilde{g}_k - \phi_k) \times V_g(\tilde{x}_a) V_F(\tilde{x}_a),$$

where $\tilde{x}_a = g(\tilde{x}_a)$.

Consider the 2D double integrator example. The extra term is

$$V g(\tilde{x}_a) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}.$$

The gradient of the basis function with respect to the mapped state $g(\tilde{x}_a)$ is a vector of length two; that is, $V_F(\tilde{x}_a) \in \mathbb{R}^2$. Left-multiplying by $V g(\tilde{x}_a)$ simply pads this vector with two zeros, leading to a vector of length four. The agent state has 4D, so the gradient with respect to this variable must be of length four. This example shows that augmentation mismatches are trivial to correct. We can simply take the gradient with respect to agent state variables that make up the distribution domain and set the partial derivatives with respect to the augmented variables to zero.

5.2 Shifting rectangular domains

Recall that in Section 2.2 we restricted our rectangular domain to be of the form $X = [0, L_i] \times [0, L_i] \times \cdots \times [0, L_i]$. That is, one corner is at the origin and the rectangle occupies the positive orthant. However, sometimes we prefer the domain to be positioned differently; we might prefer the domain $[-1, 1] \times [-1, 1]$ instead of $[0, 2] \times [0, 2]$. To shift the domain, we simply shift the arguments of sin and cos in (7)–(9), replacing $x_i$ with $(x_i - \text{imin})$, where $\text{imin}$ indicates the minimum value of the dimension $i$. This assumes $L_i$ denotes the length of a dimension $i$, instead of the maximum value. For example, (7) becomes

$$F_k(x) = \frac{1}{h_k} \prod_{i=1}^n \cos \left( \frac{k \pi (x_i - \text{imin})}{L_i} \right).$$

If the rectangular domain is shifted, the boundary penalty and gradients are also changed in a similar way.

5.3 Integration

So far we have only integrated our dynamics with the simple forward Euler method. This integration scheme is explicit — only current state information is used to generate the state at the next time step. Using a large step size $\Delta t$ can lead to numerical issues when integrating with the explicit forward Euler method because it does not conserve energy; each step in the integration can artificially add energy to the system, leading to unstable trajectories [10].

To understand how energy is added, consider a mass attached to a spring. Initially, the mass starts with zero velocity and some displacement from the spring equilibrium. If generated with the explicit Euler, the next state will have the same displacement — the initial state had no velocity — but non-zero velocity because the
initial displacement induced a spring force. The potential energy is the same, but kinetic energy has been added as the velocity is no longer zero. Our integration has increased the total energy of the system, even though no external force acts on it. Likewise, suppose the mass started at the spring equilibrium but with non-zero velocity. Integration leads to the same velocity as there is no displacement and spring force in the initial state, but the position changes. Now potential energy has been added and kinetic energy remains the same, increasing the system’s total energy. Integrating over many steps would incur more energy errors, leading to oscillations that increasingly grow in amplitude.

If the system is linear, we can avoid these integration errors by performing an exact integration. This method assumes control is constant over the step (which we assume) and allows us to change performing an exact integration. This method assumes control is

\[
\delta \nu_{n+1} = \delta \nu_n + \Delta \left( A_\nu \delta p_n + A_\nu \delta \nu_n + B_\delta \delta u_n \right).
\]

We can solve for \( \delta \nu_{n+1} \):

\[
\delta \nu_{n+1} = \left[ \Delta t (I - A_{22})^{-1} A_{21} \right] \delta p_n + \left[ \Delta t (I - A_{22})^{-1} \right] \delta \nu_n + \left[ \Delta t (I - \Delta t A_{22})^{-1} B_2 \right] \delta \delta u_n.
\]

We can define the bracketed parts as blocks in new matrices \( \tilde{A}_\nu \) and \( \tilde{B}_\nu \) that describe the discrete update

\[
\begin{bmatrix}
\delta p_{n+1} \\
\delta \nu_{n+1}
\end{bmatrix} =
\begin{bmatrix}
\tilde{A}_\nu & \tilde{B}_\nu \\
\tilde{A}_\nu & \tilde{B}_\nu
\end{bmatrix}
\begin{bmatrix}
\delta p_n \\
\delta \nu_n
\end{bmatrix} +
\begin{bmatrix}
\tilde{B}_\nu \\
\tilde{B}_\nu
\end{bmatrix} \delta \delta u_n.
\]

where the block matrices are as follows:

\[
\tilde{A}_{11} = I + \Delta t A_{11} + (\Delta t)^2 A_{11} (I - \Delta t A_{22})^{-1} A_{11}
\]

\[
\tilde{A}_{12} = \Delta t A_{12} (I - \Delta t A_{22})^{-1}
\]

\[
\tilde{A}_{22} = \left( \Delta t (I - \Delta t A_{22})^{-1} \right) B_2 + \Delta t B_2
\]

\[
\tilde{B}_1 = \Delta t (I - h A_{22})^{-1} B_1
\]

At time step \( n \), we can take the continuous matrices \( A_\nu \) and \( B_\nu \) and use these equations to create \( \tilde{A}_\nu \) and \( \tilde{B}_\nu \) which are then used in the LQ and LQR constraints.

Prabhakar et al. [10] presented a good example demonstrating the importance of symplectic integration. The trajectory is 50 steps long with a step size \( \Delta t = 0.6 s \) (for a time horizon of 30 s). The system is an undamped oscillator with the following dynamics:

\[
\begin{bmatrix}
p_{n+1} \\
\nu_{n+1}
\end{bmatrix} =
\begin{bmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
p_n \\
\nu_n \\
\epsilon_n \\
\epsilon_n
\end{bmatrix}
\]

\[
\begin{bmatrix}
p_{n+1} \\
\nu_{n+1} \\
\epsilon_{n+1} \\
\epsilon_{n+1}
\end{bmatrix} =
\begin{bmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
p_n \\
\nu_n \\
\epsilon_n \\
\epsilon_n
\end{bmatrix} +
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
\epsilon_n \\
\epsilon_n
\end{bmatrix}.
\]

The agent starts from rest at (0.0, −0.49) and the trajectory is initialised using a constant \( \epsilon_n = [1e-4, 1e-4] \). Using explicit, exact, and symplectic integration, we generate trajectories, terminating the optimisation with a directional derivative criterion of \( 1 \times 10^{-4} \).

Fig. 5 shows resulting trajectories when using explicit, exact, and symplectic integration. Explicit integration artificially adds energy to the system. Fighting this extra energy is difficult, so the resulting trajectory deviates wildly from the domain, leveraging the periodicity of the ergodic metric. It is tempting to simply impose a boundary cost and stick with explicit Euler integration. This solution can keep the trajectory in the domain, as Fig. 5 shows, but the system’s energy is still artificially increased. Table 1 shows the control and computational costs for each trajectory. When using explicit integration and a boundary penalty, the control cost is much higher, because the control inputs are fighting against kinetic energy that has been artificially introduced. Further, solving takes much longer, because the solver must balance the competing interests of artificial energy and bounding the domain. It is important to note that symplectic integration is not always needed, but it becomes important as step size increases or in systems with dynamics sensitive to artificial energy increases.
5.4 Time-evolving spatial distributions

Suppose a spatial distribution evolves with time in a known manner. The spatiotemporal distribution, denoted \( \phi(x, t) \), yields a spatial density \( \phi(x) \) at a given time \( t \). We can generate trajectories that are ergodic with respect to \( \phi(x, t) \) by extending ergodicity to time-evolving distributions. To do so, we simply add time as a variable in the workspace [3]. For a rectangular domain in \( \mathbb{R}^3 \), the resulting basis function is

\[
F_k(x, t) = \frac{1}{T} \prod_{i=1}^{s} \cos \left( \frac{k_i x_i}{L_i} \right) \cos \left( \frac{k_{s+1} t}{T} \right),
\]

where \( k \) is of length \( s + 1 \), with the last element corresponding to the newly added time dimension. The distribution decomposition is similar to the time-invariant version shown in (20). Because the trajectory is discrete, one spatial density is needed per step. The decomposition is normalised by the number of spatial densities to normalise the spatiotemporal distribution, leading to

\[
\phi_k \approx \frac{1}{N+1} \sum_{n=0}^{N} \sum_{x_n \in X_n} \phi(x_n, n\Delta t) F_k(x_n, n\Delta t) V_c.
\]  

As noted by Miller [3], the optimisation is unchanged beyond the computation of ergodic metric and gradient. Fig. 6 shows an ergodic trajectory for a single integrator and a time-evolving distribution.

5.5 Multi-agent trajectories

Suppose we have a multi-agent system consisting of \( N_a \) agents and we wish their combined motion to be ergodic with respect to some spatial distribution. We denote the set of individual agent trajectories \( \{\tilde{x}_j\} \) and use \( x_j^\ast \) to refer to the state of the agent \( j \) at time step \( n \). The multi-agent system’s trajectory coefficients are an average of the individual trajectory coefficients

\[
c_k((\tilde{x})) = \frac{1}{N_a(N+1)} \sum_{n=1}^{N_a} \sum_{c=0}^{N} F_k(x_n^\ast).
\]

The ergodic gradient becomes

\[
V_{\Theta}((\tilde{x})) = \frac{2}{N_a(N+1)} \sum_{c=0}^{N} \Lambda_c (c_k - \phi_k) \nabla F_k(x_n^\ast).
\]

Otherwise, the optimisation is unchanged. The LQ and LQR problems can be solved with respect to the overall system state, which is just a concatenation of individual agent states. The gradients at each time step can also be concatenated. The constraint and reward matrices are block diagonal matrices constructed with the individual agent matrices. The multi-agent system can be heterogeneous, with each agent having different dynamic models.

An example multi-agent trajectory is shown in Fig. 7. The multi-agent system consists of two single integrator agents, one starting from \((0.5, 0.02)\) and the other from \((0.2, 0.98)\). The resulting trajectory is efficient – each covers the distribution mode closest to its starting point, resulting in an ergodic multi-agent trajectory with low control cost.

5.6 Spectral multiscale coverage

In addition to presenting a commonly used ergodic metric, Mathew and Mezić [12] derived a simple algorithm for designing ergodic trajectories, called spectral multiscale coverage (SMC). SMC consists of simple feedback laws that are only valid for multi-agent systems of single or double integrators. Most importantly, SMC has low computational cost and generates trajectories quickly, so we briefly cover it here and compare it to projection-based optimisation.

The SMC feedback laws were designed to maximise the rate of decay of the ergodic metric at the end of an infinitesimal time horizon. For single integrator systems, the feedback law leads to a control input \( u(t) \) for an agent \( j \) at the time \( t \) and is defined as

\[
u(t) = -u_{\text{max}} \frac{B_j(t)}{\| B_j(t) \|_2}.
\]

\begin{table}[h]
\centering
\caption{Effect of integration type and boundary penalty on generated trajectories}
\begin{tabular}{|c|c|c|c|c|}
\hline
\text{Integration type} & Explicit & Exact & Symplectic & Explicit \\
\text{boundary penalty} & 0.0 & 0.0 & 0.0 & 1.0 \\
\text{ergodic cost} & 0.0098 & 0.0045 & 0.0038 & 0.0020 \\
\text{control cost} & 0.014 & 0.00074 & 0.00097 & 0.031 \\
\text{iterations} & 12 & 12 & 14 & 393 \\
\text{solve time, s} & 0.019 & 0.019 & 0.046 & 0.812 \\
\hline
\end{tabular}
\end{table}
where $u_{\text{max}}$ is the maximum agent velocity and
\begin{equation}
B(t) = tN_n \sum \lambda_k (c_k(t) - \phi_k) V F_k(x(t)).
\end{equation}
where $N_n$ is the number of agents in the system and $c_k(t)$ is the decomposition of the trajectory executed so far. Denote the current time step $n_i$, so that $t = n_i \Delta t$, the decomposition is
\begin{equation}
c_k(t) = \frac{1}{N_n(n_i + 1)} \sum_{j = n_i}^{n_i + N_n} F_j(\tilde{x}_j).
\end{equation}
In the double integrator case, Mathew and Mezić [12] decompose agent $j$'s state into the agent position $p_j(t)$ and velocity $\nu_j(t)$, leading to the feedback law
\begin{equation}
u_j(t) = -F_{\text{max}} \frac{c_k(j)(t) + B_j(t)}{\|c_k(j)(t) + B_j(t)\|_2},
\end{equation}
where $F_{\text{max}}$ is the maximum force that can be applied and $c_k$ is a parameter describing a penalty on kinetic energy. When $c_k$ is high, the agent will have slower velocities.

SMC is fast and can easily be computed in real time. However, SMC suffers from two major drawbacks. The first is that it only works for two dynamics models – single and double integrators. Second, resulting trajectories typically require far more control effort than those from projection-based optimisation. This inefficiency arises from the greedy nature of the algorithm and its lack of a control penalty. To see the effects of SMC’s greed, consider an agent starting at one mode of a bimodal distribution, which each mode has equal density. Because it has already spent some time at the mode, the agent will move to the second mode to balance the time spent at the modes, thereby reducing the ergodic metric. However, once the agent has spent more time at the second mode than it had at the first, it will return to the first mode to rebalance. A more efficient approach would spend half of the time in the first mode and then move to the second for the remaining time. However, this approach takes the entire trajectory into account, not just the next step. Mathew and Mezić [12] call this effect multiscale – the agent first covers the distribution coarsely and then starts to cover it more finely.

There are applications where this multiscale effect is desirable, such as uniform coverage problems, where the goal is to evenly sample points in some domain [12]. Given enough time, an agent following an SMC trajectory will visit all states with non-zero density. In contrast, a projection-based trajectory is generated with a specific horizon in mind and balances ergodicity with control effort, leading to trajectories that cover a space more sparsely. This feature helps explain why robotic painting is better solved by SMC [19]. Replicating an image requires covering some areas very densely, even at a high control cost, and it is hard to coarsen the projection-based method to this behaviour [19]. Further, many discrete points are needed to faithfully recreate an image, adding to the already high computational cost of projection-based optimisation.

ErgodicControl.jl also generates SMC trajectories, and Fig. 8 shows an example of the multiscale effect. Trajectories are generated for an agent with single integrator dynamics and 400 time steps. When using SMC, the agent oscillates between the distribution's two modes, which is an inefficient way to cover them. However, the entire trajectory is computed in only 0.001 s, whereas the projection-based trajectory reached directional derivative criteria of $1 \times 10^{-4}$ in 0.48 s.

6 Conclusion

Projection-based trajectory optimisation is not the only technique for generating ergodic trajectories. However, it has been the focus of much more research and its various improvements are spread across several publications. This paper is our attempt to condense this work and cast light on implementation details that could foster new users. We also present a software package, ErgodicControl.jl, which generates ergodic trajectories with projection-based optimisation faster than previously reported. Our goal is to automate ergodic trajectory generation so it can be applied to new applications without having to ‘reinvent the wheel’. The package can be found at https://github.com/dressel/ErgodicControl.jl.

Future work will leverage our software package to evaluate ergodic control in a number of different applications. Future work will also focus on adding functionality to the software package. Added functionality will include new ergodic trajectory generation methods, including a sampling-based motion planner [7] and a potential field-based method [6].

7 Acknowledgments

This work was funded by National Science Foundation grant DGE-114747.

8 References

[1] Miller, L.M., Murphy, T.D.: ‘Optimal planning for target localization and coverage using range sensing’. Proc. IEEE Conf. on Automation Science and Engineering (CASE), Gothenburg, Sweden, 2015
[2] Miller, L.M., Silverman, Y., MacIver, M.A., Murphy, T.D.: ‘Ergodic exploration of distributed information’, IEEE Trans. Robot., 2016, 32(1), pp. 36–52
[3] Miller, L.M.: ‘Optimal Ergodic Control for Active Search and Information Acquisition’. PhD thesis, Northwestern University, 2015
[4] De La Torre, G., Flathkamp, K., Prabhakar, A., Murphy, T.D.: ‘Ergodic exploration with stochastic sensor dynamics’. Proc. American Control Conf. (ACC), Boston, MA, USA, 2016
[5] Ansari, A.R., Murphy, T.D.: ‘Sequential action control: closed-form optimal control for nonlinear and nonsmooth systems’, IEEE Trans. Robot., 2016, 32(5), pp. 1196–1214
[6] Ivić, S., Crnković, B., Mezić, I.: ‘Ergodicity-based cooperative multiagent area coverage via a potential field’, IEEE Trans. Cybern., 2017, 47(8), pp. 1983–1993
[7] Ayvali, E., Salman, H., Choset, H.: ‘Ergodic coverage in constrained environments using stochastic trajectory optimization’. Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), Vancouver, BC, Canada, 2017

IET Cyber-Phys. Syst., Theory Appl., 2019, Vol. 4 Iss. 2, pp. 89-100
This is an open access article published by the IET under the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/)

99
[8] Miller, L.M., Murphey, T.D.: ‘Trajectory optimization for continuous ergodic exploration’. Proc. American Control Conf. (ACC), Washington, DC, USA, 2013.

[9] Miller, L.M., Murphey, T.D.: ‘Trajectory optimization for continuous ergodic exploration on the motion group SE(2)’. Proc. IEEE Conf. on Decision and Control (CDC), Florence, Italy, 2013.

[10] Prabhakar, A., Fläskamp, K., Murphey, T.D.: ‘Symplectic integration for optimal ergodic control’. Proc. IEEE Conf. on Decision and Control (CDC), Osaka, Japan, 2015.

[11] Bezanson, J., Karpinski, S., Shah, V.B., Edelman, A.: ‘Julia: a fast dynamic language for technical computing’, 2012, arXiv preprint arXiv:1209.5145.

[12] Mathew, G., Mezić, I.: ‘Metrics for ergodicity and design of ergodic dynamics for multi-agent systems’, Physica D, Nonlinear Phenom., 2011, 240, (4), pp. 432–442.

[13] Chirikjian, G.S., Kyatkin, A.B.: ‘Engineering applications of noncommutative harmonic analysis: with emphasis on rotation and motion groups’ (CRC press, Boca Raton, FL, USA, 2000).

[14] Nocedal, J., Wright, S.J.: ‘Numerical optimization. Springer series in operations research and financial engineering’ (Springer, New York, 2000).

[15] Hauser, J.: ‘A projection operator approach to the optimization of trajectory functionals’, IFAC Proc. Vol., 2002, 350, (1), pp. 377–382.

[16] Johnson, E.: ‘Trajectory optimization and regulation for constrained discrete mechanical systems’. PhD thesis, Northwestern University, 2012.

[17] Schultz, J., Johnson, E., Murphey, T.D.: ‘Trajectory optimization in discrete mechanics’ (Springer International Publishing, New York, NY, USA, 2015), In Press.

[18] Armijo, L.: ‘Minimization of functions having Lipschitz continuous first partial derivatives. Pac. J. Math., 1966, 160, (1), pp. 1–3.

[19] Prabhakar, A., Mavrommati, A., Schultz, J., Murphey, T.D.: ‘Autonomous visual rendering using physical motion’. Proc. Workshop on the Algorithmic Foundations in Robotics (WAFR), San Francisco, CA, USA, 2016.

[20] Dubins, L.E.: ‘On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents’, Am. J. Math., 1957, 790, (3), pp. 497–516.

[21] Crassidis, J.L., Junkins, J.L.: ‘Optimal estimation of dynamic systems’ (CRC Press, New York, NY, USA, 2012).

[22] Hairer, E., Lubich, C., Wanner, G.: ‘Geometric numerical integration: structure-preserving algorithms for ordinary differential equations’, volume 31 (Springer Science & Business Media, Berlin, Germany, 2006).

This is an open access article published by the IET under the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/).