A Extended Related Work

We address related work within two primary areas: dexterous high-dimensional control, and robotic pianists.

Dexterous Manipulation and High-Dimensional Control The vast majority of the control literature uses much lower-dimensional systems (i.e., single-arm, simple end-effectors) than high-dimensional dexterous hands. Specifically, only a handful of general-purpose policy optimization methods have been shown to work on high-dimensional hands, even for a single hand [10, 9, 12, 11, 16, 14, 17, 7], and of these, only a subset has demonstrated results in the real world [10, 9, 12, 11, 16]. Results with bi-manual hands are even rarer, even in simulation only [15, 18].

As a benchmark, perhaps the most distinguishing aspect of ROBOPIANIST is in the definition of "task success". As an example, general manipulation tasks are commonly framed as the continual application of force/torque on an object for the purpose of a desired change in state (e.g., SE(3) pose and velocity). Gradations of dexterity are predominantly centered around the kinematic redundancy of the arm or the complexity of the end-effector, ranging from parallel jaw-grippers to anthropomorphic hands [44, 15]. A gamut of methods have been developed to accomplish such tasks, ranging from various combinations of model-based and model-free RL, imitation learning, hierarchical control, etc. [45, 10, 13, 12, 46, 47]. However, the class of problems generally tackled corresponds to a definition of dexterity pertaining to traditional manipulation skills [19], such as re-orientation, relocation, manipulating simply-articulated objects (e.g., door opening, ball throwing and catching), and using simple tools (e.g., hammer) [20, 21, 15, 11, 22]. The only other task suite that we know of that presents bi-manual tasks, the recent Bi-Dex [15] suite, presents a broad collection of tasks that fall under this category.

While these works represent an important class of problems, we explore an alternative notion of dexterity and success. In particular, for most all the aforementioned suite of manipulation tasks, the "goal" state is some explicit, specific geometric function of the final states; for instance, an open/closed door, object re-oriented, nail hammered, etc. This effectively reduces the search space for controls to predominantly a single "basin-of-attraction" in behavior space per task. In contrast, the ROBOPIANIST suite of tasks encompasses a more complex notion of a goal, which is encoded through a musical performance. In effect, this becomes a highly combinatorially variable sequence of goal states, extendable to arbitrary difficulty by only varying the musical score. "Success" is graded on accuracy over an entire episode; concretely, via a time-varying non-analytic output of the environment, i.e., the music. Thus, it is not a matter of the "final-state" that needs to satisfy certain termination/goal conditions, a criterion which is generally permissive of less robust execution through the rest of the episode, but rather the behavior of the policy *throughout the episode* needs to be precise and musical.

Similarly, the literature on humanoid locomotion and more broadly, "character control", another important area of high-dimensional control, primarily features tasks involving the discovery of stable walking/running gaits [48, 49, 50], or the distillation of a finite set of whole-body movement priors [51, 52, 53], to use downstream for training a task-level policy. Task success is typically encoded via rewards for motion progress and/or reaching a terminal goal condition. It is well-documented that the endless pursuit of optimizing for these rewards can yield unrealistic yet "high-reward" behaviors. While works such as [51, 54] attempt to capture *stylistic* objectives via leveraging demonstration data, these reward functions are simply appended to the primary task objective. This scalarization of multiple objectives yields an arbitrarily subjective Pareto curve of optimal policies. In contrast, performing a piece of music entails both objectively measurable precision with regards to melodic and rhythmic accuracy, as well as a subjective measure of musicality. Mathematically, this translates as *stylistic* constraint satisfaction, paving the way for innovative algorithmic advances.

Robotic Piano Playing Robotic pianists have a rich history within the literature, with several works dedicated to the design of specialized hardware [23, 24, 25, 26, 27, 28], and/or customized controllers for playing back a song using pre-programmed commands (open-loop) [29, 30]. The work

in [31] leverages a combination of inverse kinematics and trajectory stitching to play single keys and playback simple patterns and a song with a Shadow hand [37]. More recently, in [32], the author simulated robotic piano playing using offline motion planning with inverse kinematics for a 7-DoF robotic arm, along with an Iterative Closest Point-based heuristic for selecting fingering for a four-fingered Allegro hand. Each hand is simulated separately, and the audio results are combined post-hoc. Finally, in [33], the authors formulate piano playing as an RL problem for a single Allegro hand (four fingers) on a miniature piano, and additionally leverage tactile sensor feedback. However, the tasks considered are rather simplistic (e.g., play up to six successive notes, or three successive chords with only two simultaneous keys pressed for each chord). The ROBOPIANIST benchmark suite is designed to allow a general bi-manual controllable agent to emulate a pianist's growing proficiency on the instrument by providing a curriculum of musical pieces, graded in difficulty. Leveraging two underactuated anthropomorphic hands as actuators provides a level of realism and exposes the challenge of mastering this suite of high-dimensional control problems.

B Detailed Reward Function

Reward	Formula	Weight	Explanation
Key Press	$0.5 \cdot g(\|\|k_s - k_g\|\|_2) + 0.5 \cdot (1 - \mathbf{1}_{\{\text{false positive}\}})$	1	Press the right keys and only the right keys
Energy Penalty	$\|\tau_{\text{joints}}\|^\top \|\text{v}_{\text{joints}}\|$	-5e-3	Minimize energy expenditure
Finger Close to Key	$g(\|\|p_f - p_k\|\|_2)$	1	Shaped reward to bring fingers to key

Table 2: The reward function used to train ROBOPIANIST agents. τ represents the joint torque, v is the joint velocity, p_f and p_k represent the position of the finger and key in the world frame respectively, k_s and k_g represent the current and the goal states of the key respectively, and g is a function that transforms the distances to rewards in the $[0, 1]$ range.

C Full Repertoire Results

Figure 9: Results on the full repertoire of 150 songs.

D ROBOPIANIST Training details

Computing infrastructure and experiment running time

Our model-free RL codebase is implemented in JAX [55]. Experiments were performed on a Google Cloud `n1-highmem-64` machine with an Intel Xeon E5-2696V3 Processor hardware with 32 cores (2.3 GHz base clock), 416 GB RAM and 4 Tesla K80 GPUs. Each "run", i.e., the training and evaluation of a policy on one task with one seed, took an average of 5 hrs wall clock time. These run times are recorded while performing up to 8 runs in parallel.

Network architecture

We use a regularized variant of clipped double Q-learning [56, 57], specifically DroQ [41], for the critic. Each Q-function is parameterized by a 3-layer multi-layer perceptron (MLP) with `ReLU` activations. Each linear layer is followed by dropout [58] with a rate of 0.01 and layer normalization [59]. The actor is implemented as a `tanh`-diagonal-Gaussian, and is also parameterized by a 3-layer MLP that outputs a mean and covariance. Both actor and critic MLPs have hidden layers with 256 neurons and their weights are initialized with Xavier initialization [60], while their biases are initialized to zero.

Training and evaluation

We first collect 5000 seed observations with a uniform random policy, after which we sample actions using the RL policy. We then perform one gradient update every time we receive a new environment observation. We use the Adam [61] optimizer for neural network optimization. Evaluation happens in parallel in a background thread every 10000 steps. The latest policy checkpoint is rolled out by taking the mean of the output (i.e., no sampling). Since our environment is "fixed", we perform only one rollout per evaluation.

Reward formulation

The reward function for training the RL agent consists of three terms: 1) a key press term r_{key}, 2) a move finger to key term r_{finger}, and 3) an energy penalty term r_{energy}.

r_{key} encourages the policy to press the keys that need to be pressed and discourages it from pressing keys that shouldn't be pressed. It is implemented as:

$$r_{\text{key}} = 0.5 \cdot \left(\frac{1}{K} \sum_{i}^{K} g(||k_s^i - 1||_2) \right) + 0.5 \cdot (1 - \mathbf{1}_{\{\text{false positive}\}}),$$

where K is the number of keys that need to be pressed at the current timestep, k_s is the normalized joint position of the key between 0 and 1, and $\mathbf{1}_{\{\text{false positive}\}}$ is an indicator function that is 1 if any key that should not be pressed creates a sound. g is the `tolerance` function from the `dm_control` [50] library: it takes the L2 distance of k_s and 1 and converts it into a bounded positive number between 0 and 1. We use the parameters `bounds=0.05` and `margin=0.5`.

r_{finger} encourages the fingers that are active at the current timestep to move as close as possible to the keys they need to press. It is implemented as:

$$r_{\text{finger}} = \frac{1}{K} \sum_{i}^{K} g(||p_f^i - p_k^i||_2),$$

where p_f is the Cartesian position of the finger and p_i is the Cartesian position of a point centered at the surface of the key. g for this reward is parameterized by `bounds=0.01` and `margin=0.1`.

Finally, r_{energy} penalizes high energy expenditure and is implemented as:

$$r_{\text{energy}} = |\tau_{\text{joints}}|^\top |\mathsf{v}_{\text{joints}}|,$$

where τ_{joints} is a vector of joint torques and $\mathsf{v}_{\text{joints}}$ is a vector of joint velocities.

The final reward function sums up the aforementioned terms as follows:

$$r_{\text{total}} = r_{\text{key}} + r_{\text{finger}} - 0.005 \cdot r_{\text{energy}}$$

Other hyperparameters

For a comprehensive list of hyperparameters used for training the model-free RL policy, see Table 3.

Hyperparameter	Value
Total train steps	5M
Optimizer	
Type	ADAM
Learning rate	3×10^{-4}
β_1	0.9
β_2	0.999
Critic	
Hidden units	256
Hidden layers	3
Non-linearity	`ReLU`
Dropout rate	0.01
Actor	
Hidden units	256
Hidden layers	3
Non-linearity	`ReLU`
Misc.	
Discount factor	0.99
Minibatch size	256
Replay period every	1 step
Eval period every	10000 step
Number of eval episodes	1
Replay buffer capacity	1M
Seed steps	5000
Critic target update frequency	1
Actor update frequency	1
Critic target EMA momentum (τ_Q)	0.005
Actor log std dev. bounds	$[-20, 2]$
Entropy temperature	1.0
Learnable temperature	True

Table 3: Hyperparameters for all model-free RL experiments.

E Multitask BC Results

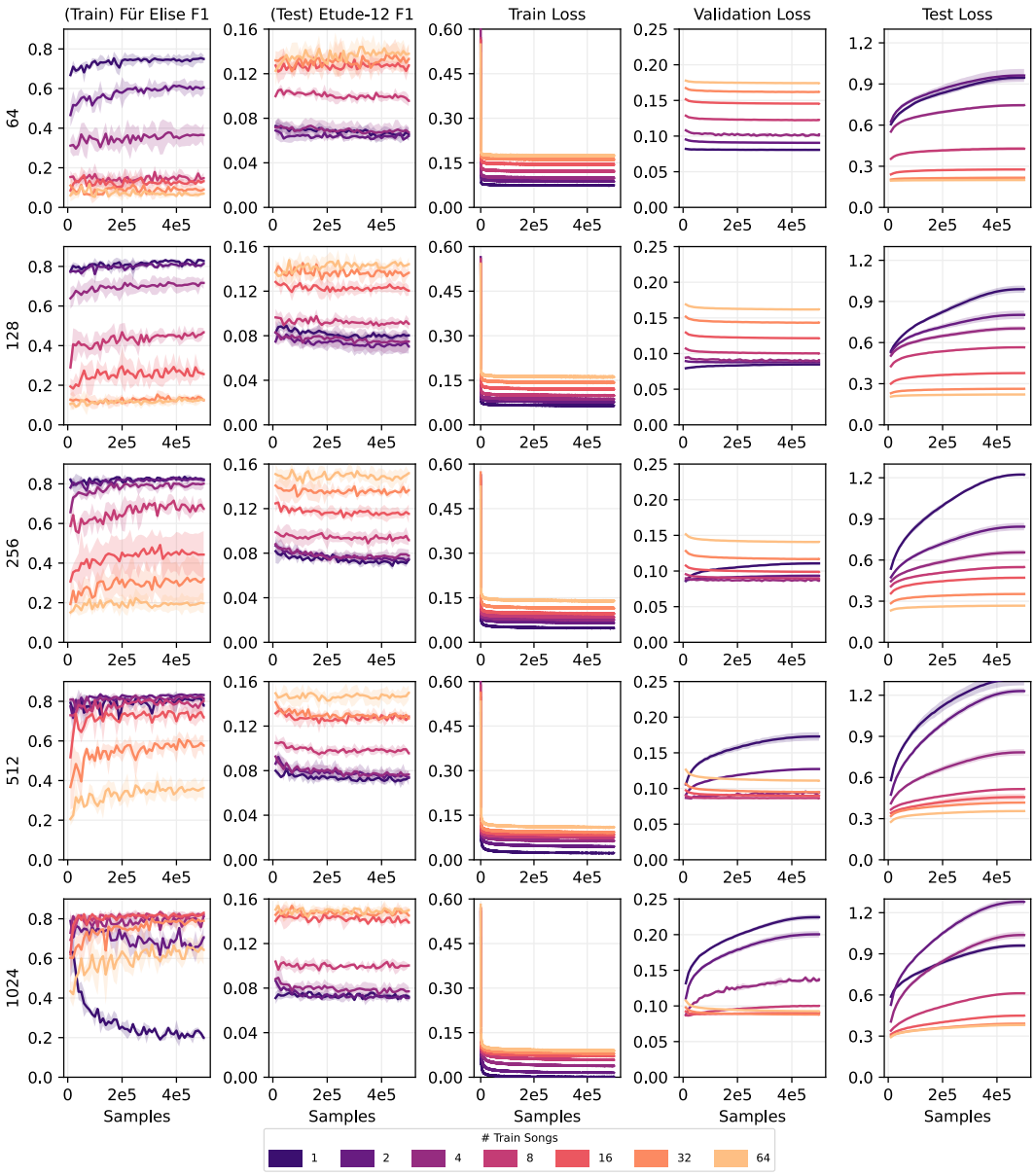

Figure 10: Caption for the figure.

F Baselines

Computing infrastructure and experiment running time

Our MPC codebase is implemented in C++ with MJPC [42]. Experiments were performed on a 2021 M1 Max Macbook Pro with 64 GB of RAM.

Algorithm

We use MPC with Predictive Sampling (PS) as the planner. PS is a derivative-free sampling-based algorithm that iteratively improves a nominal sequence of actions using random search. Concretely, N candidates are created at every iteration by sampling from a Gaussian with the nominal as the mean and a fixed standard deviation σ. The returns from the candidates are evaluated, after which the highest scoring candidate is set as the new nominal. The action sequences are represented with cubic splines to reduce the search space and smooth the trajectory. In our experiments, we used

$N = 10$, $\sigma = 0.05$, and a spline dimension of 2. We plan over a horizon of 0.2 seconds, use a planning time step of 0.01 seconds and a physics time step of 0.005 seconds.

Cost formulation

The cost function for the MPC baseline consists of 2 terms: 1) a key press term c_{key}, 2) and a move finger to key term c_{finger}.

The costs are implemented similarly to the model-free baseline, but don't make use of the g function, i.e., they solely consist in unbounded l2 distances.

The total cost is thus:

$$c_{\text{total}} = c_{\text{key}} + c_{\text{finger}}$$

Note that we experimented with a control cost and an energy cost but they decreased performance so we disabled them.

Alternative baselines

We also tried the optimized *derivative-based* implementation of iLQG [62] also provided by [42], but this was not able to make substantial progress even at significantly slower than real-time speeds. iLQG is difficult to make real time because the action dimension is large and the algorithm theoretical complexity is $O(|A|^3 \cdot H)$. The piano task presents additional challenges due to the large number of contacts that are generated at every time step. This make computing derivatives for iLQG very expensive (particularly for our implementation which used finite-differencing to compute them). A possible solution would be to use analytical derivatives and differentiable collision detection.

Besides online MPC, we could have used offline trajectory optimization to compute short reference trajectories for each finger press offline and then track these references online (in real time) using LQR. We note, however, that the (i) high dimensionality, (ii) complex sequence of goals adding many constraints, and (iii) overall temporal length (tens of seconds) of the trajectories pose challenges for this sort of approach.