Benchmarking infrastructure for Bayesian workflows?

Together, we are exploring and pushing the Pareto frontier of simulation-based inference across Bayesian models varying in many axes of difficulty.

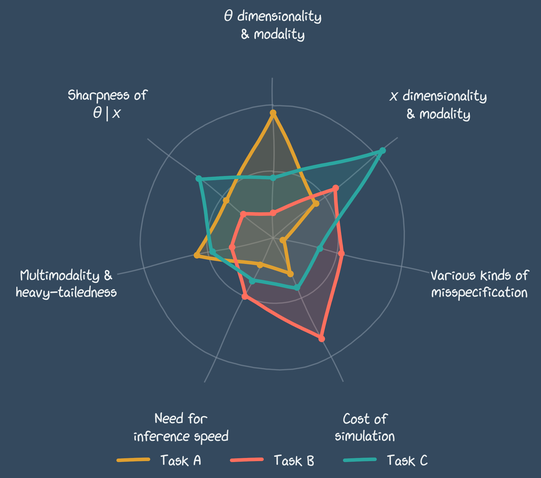

Fig 1: Difficulty profiles of different tasks.

Advancing the state of the art in inference depends on measuring methods against high-quality community benchmarks.

Driving adoption of simulation-based and amortized Bayesian inference depends on task-specific method recommendations grounded in those benchmarks.

Ideals to aim for

Cohesion — the same benchmark tasks are both accessible in principle and actively used by everybody to allow straightforward comparison between method contributions for identical tasks.

Dynamism — there is no significant hurdle to propose new tasks, so the diversity of tasks keeps up dynamically with method development and community members actively contribute new tasks regularly.

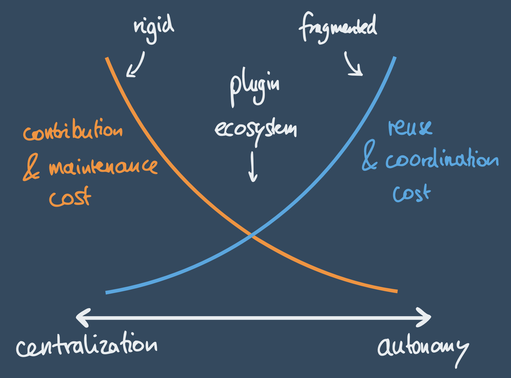

Fig 2: The centralization-autonomy trade-off. Too centralized is rigid, too autonomous becomes fragmented; a plugin ecosystem strikes the balance.

A plugin ecosystem of benchmarks

Scientists build, test and maintain tasks in their **own repositories**. Each repository is a pip installable package registering an `abibench.tasks` entry point; method papers cite **unique, fingerprinted tasks**. Authoring task plugins is flexible and convenient due to separation of **contract** and **implementation** in runtime checkable **protocols** and hackable **mixins**.

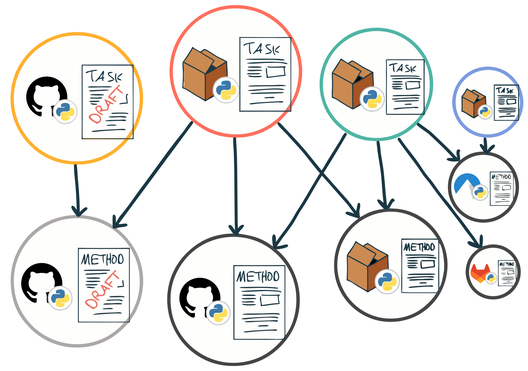

Fig 3: Tasks and methods are interchangeable and independently owned; drafts start being used locally and can grow into public benchmark tasks.

Reproducible artifacts like training and test datasets or reference samples

Standardized task and simulator contracts make every artifact's path and content predictable, so abibench can **keep cached data in sync with its simulator**. Inheriting `SampleJointMixin`, `LocalArtifactStoreMixin` and `UsesRNGMixin`, gets this for free:

from a script …

```python
t = LinearRegressionTask()
joint_train, _ = t.sample_joint(seed=0, n=5000, disk=True)
```

…or the command line

```terminal
~ $ abibench linear_regression sample_joint 5000 --seed 0
```

→

request artifact for

```
id      = linear_regression
version = 0.1.0
n       = 5000
seed    = 0
```

abibench resolves it

1. **ensure simulator consistent**
   - `load` stored fingerprint `(n=2, s=0)`
   - `generate` fresh fingerprint `(n=2, s=0)`
   - `if not identical` → **raise RuntimeError**
2. **get data**
   - `if` in store → load
   - `else` generate fresh → **store & return**

Refactor the simulator freely: cached artifacts stay verified. A fingerprint mismatch? Revert the change, or **bump the task's version**.

Semantic versioning rules

```
version: str = f"{major}.{minor}.{patch}"
```

major increase: generative model changes
minor increase: seed breaks; generative model untouched
patch increase: implementation, docs, …; seed determinism intact

Any Bayesian Inference method has its perks. Always Benchmarking Improves clarity. Accessible Benchmarks Invite reuse. Awesome Benchmarking Infrastructure is decentralized and open. Anyone Builds, Integrates, and maintains their own tasks. All Bayesians Invited!

Getting started

```terminal
~ $ pip install abibench
~ $ cookiecutter abibench-task-template
  [1] task_name: Holy Grail of Inference
  [2] task_id: holy_grail
  [3] package_name: abibench-holy-grail
  ...
~ $ pip install -e abibench-holy-grail
~ $ abibench holy_grail sample_joint 5000 --seed 42
# no central registration required to be fully
  interoperable with the ecosystem
```

Protocols & base classes & mixins

	identity	simulator		capabilities	
Protocol	**TaskSpec**	**SimulatorLike** UsesRNG HasPriorBatched HasObsBatched	*SampleJointProvider*	*LocalArtifactProvider* RemoteArtifactProvider	ReferenceProvider
Base/Mixin	–	SimulatorBase UsesRNGMixin	SampleJointMixin	LocalArtifactMixin RemoteArtifactMixin	ReferenceMixin

bold = required *italic* = recommended plain = optional

Test Driven Task Development

```terminal
~ $ abibench compliance holy_grail
  holy_grail - conforms
  required TaskSpec ok
  required SimulatorLike ok
  recommended LocalArtifactProvider ok
  ...
  optional HasObsBatched n/a
  40 passed, 2 skipped in 1.83s
```

The CLI offers a comprehensive compliance check building on `pytest` for TDD. Required protocols ensure structure and typing compliance; optional protocols gate further behavioral checks of task objects.

Example code snippets

the task protocol

```python
@runtime_checkable
class TaskSpec(Protocol):
    id: str       # stable identifier
    version: str  # semantic versioning string
    name: str     # brief, free-form descriptive title
    simulator: SimulatorLike
    @property params: dict[str, float | int | bool | str] # optional.
```

a concrete plugin task

```python
class LinearRegressionTask(SampleJointMixin, LocalArtifactMixin, ReferencesMixin):
    id = "linear_regression"
    version = "0.1.0"
    name = "Bayesian Linear Regression"
    builtin_references = ("analytic",)
    # fixed, no self.params
    settings = dict(beta_mean=np.zeros(2), beta_cov=np.eye(2))

    def __init__(self) -> None:
        self.simulator = LinearRegressionSimulator(**LinearRegressionTask.settings)
        analytic = LinRegAnalyticPosterior(**LinearRegressionTask.settings)
        self.register_runner_reference(
            name="analytic",
            sample_fn=analytic.sample,
            log_prob_fn=analytic.log_prob,
        )
```

abibench

Hans Olischläger | hi@hans.olischlaeger.com | TU Dortmund University | supervised by Prof. Paul Bürkner and Prof. Stefan Radev