Figaro Release Notes

About Figaro

Reasoning under uncertainty requires taking what you know and inferring what you don't know, when what you know doesn't tell you for sure what you don't know. A well-established approach for reasoning under uncertainty is probabilistic reasoning. Typically, you create a probabilistic model over all the variables you're interested in, observe the values of some variables, and query others. There is a huge variety of probabilistic models, and new ones are being developed constantly. Figaro is designed to help build and reason with the wide range of probabilistic models.

Developing a new probabilistic model normally requires developing a representation for the model and a reasoning algorithm that can draw useful conclusions from evidence, and in many cases also an algorithm to learn aspects of the model from data. These can be challenging tasks, making probabilistic reasoning require significant effort and expertise. Furthermore, most probabilistic reasoning tools are standalone and difficult to integrate into larger programs.

Figaro is a probabilistic programming language that helps address both these issues. Figaro makes it possible to express probabilistic models using the power of programming languages, giving the modeler the expressive tools to create all sorts of models. Figaro comes with a number of built-in reasoning algorithms that can be applied automatically to new models. In addition, Figaro models are data structures in the Scala programming language, which is interoperable with Java, and can be constructed, manipulated, and used directly within any Scala or Java program.

Figaro is extremely expressive. It can represent a wide variety of models, including:

- directed and undirected models
- models in which conditions and constraints are expressed by arbitrary Scala functions
- models involving inter-related objects
- open universe models in which we don't know what or how many objects exist
- models involving discrete and continuous elements
- models in which the elements are rich data structures such as trees
- models with structured decisions
- models with unknown parameters

Figaro provides a rich library of constructs to build these models, and provides ways to extend this library to create your own model elements. Figaro's library of reasoning algorithms is also extensible. Current built-in algorithms include:

- Exact inference using variable elimination
- Belief propagation
- Lazy factored inference for infinite models
- Importance sampling

- Metropolis-Hastings, with an expressive language to define proposal distributions
- Support computation
- Most probable explanation (MPE) using variable elimination or simulated annealing
- Probability of evidence using importance sampling
- Particle Filtering
- Factored frontier
- Parameter learning using expectation maximization
- Gibbs sampling

Figaro provides both regular (the algorithm is run once) and anytime (the algorithm is run until stopped) versions of some of these algorithms. In addition to the built-in algorithms, Figaro provides a number of tools for creating your own reasoning algorithms.

Figaro is free and is released under an open-source license (see license file). The public code repository for Figaro can also be found at https://github.com/p2t2

What's new in Figaro 4.0?

Many new features have been introduced into Figaro 4.0 since Figaro 3.0 was released. These include:

- Release of structured factored inference (SFI). SFI is an alternative means of reasoning on a model. SFI recursively decomposes a model into a set of sub-models, where each sub-model can be solved by a different inference algorithm. There are a number of different strategies that can be applied that determine how to decompose a model, and how to select an inference algorithm to apply to each sub-model. Figaro now contains SFI versions of variable elimination, belief propagation, and Gibbs sampling, as well as some hybrid approaches that automatically choose the best algorithm for a sub-model.
- A number of new algorithms:
    - Gibbs sampling on factor graphs
    - Parallel Importance sampling and particle filtering
- Improvements to existing algorithms:
    - Probability, decision-making, and learning algorithms now all use the same factor generation code.
    - Importance sampling, forward sampling, and particle filtering now all use a unified likelihood weighting method.
    - Chains no longer implement caching, which has not been moved to individual algorithms to implement.
- Features to make programming easier:
    - Added a new test harness for evaluating non-deterministic tests
- New items in the library:
    - Caches for implementing caches in Figaro algorithms
- Added examples from Practical Probabilistic Programming book
- Numerous minor improvements and bug fixes

How can I use Figaro as a project dependency?

If you wish to integrate Figaro's features into your own software project, Figaro is available on Maven Central (http://search.maven.org). Shown below are a few examples of how you can add Figaro as a dependency to your existing project:

Simple Build Tool (SBT) Projects

```
libraryDependencies += "com.cra.figaro" %% "figaro" % "4.0.0.0"
```

Apache Maven Projects

```
<dependency>
    <groupId>com.cra.figaro</groupId>
    <artifactId>figaro_2.11</artifactId>
    <version>4.0.0.0</version>
</dependency>
```

Apache Ivy Projects

```
<dependency org="com.cra.figaro" name="figaro_2.11" rev="4.0.0.0" />
```

How do I compile Figaro from source code?

Figaro is maintained as open source on GitHub. The GitHub project is Probabilistic Programming Tools and Techniques (P2T2), located at https://github.com/p2t2. P2T2 currently contains the Figaro sources, but we plan to update it with more tools. If you want to see the source code and build Figaro yourself, please visit our GitHub site.

To build Figaro from GitHub source, make a fork of the Figaro repository to your GitHub account, then use git's clone feature to get the source code from your GitHub account to your machine.

```
git clone https://github.com/[your-github-username]/figaro.git
```

There are several branches available; checkout "master" for the latest stable release or the latest "DEV" branch for more cutting edge work and features (this is work in progress and therefore less stable).

Figaro uses Simple Build Tool (SBT) to manage builds, located at http://www.scala-sbt.org/. Download and install SBT, open a command prompt, switch to your newly cloned local Figaro directory, and enter this SBT command set:

```
sbt clean compile package publishLocal assembly
```

This will create Figaro for Scala 2.11; you will find the resulting artifacts in the "target" directory. To run the Figaro unit tests, use this SBT command

```
sbt test
```

Note that some of the unit tests may not always pass because their results are non-deterministic.

How do I run my own Figaro programs without using Simple Build Tool (SBT)?

While SBT is a useful tool, you may want to manage your own workspace differently.

To run Figaro, you will first need Scala. The Scala compiler can either be run from the command line or within an Integrated Development Environment (IDE). Two IDEs that support Scala development are Eclipse and IntelliJ Idea. NetBeans also has a Scala plugin but it does not appear to support recent versions of Scala (but that may have changed). This section focuses on how to obtain Scala and Figaro and run Scala programs that use Figaro from the command line. If you choose to use an IDE, please see the documentation of your IDEs and Scala plugins for details of how to include the Figaro library.

To get started, download Scala from http://scala-lang.org/download/. You will need Scala version 2.11.7 or later to run the latest version of Figaro. Follow the Scala installation instructions at http://scala-lang.org/download/install.html and make sure you can run, compile, and execute the "Hello World" program provided in the documentation.

The next step is to obtain Figaro. The Figaro binary distribution is hosted at the Charles River Analytics, Inc. Web site. Go to https://www.cra.com/figaro. The current version, as of January 2016, is 4.0.0.0, and is available for Scala 2.11. Always make sure the Figaro version you use matches the Scala version you're using. Each available download link is a compressed archive containing the Figaro jar (jar is the Java/Scala format for compiled byte code), examples, documentation, Scaladoc, and source code files. Click the appropriate link and then uncompress the downloaded archive to access the Figaro jar file. In the distribution, the Figaro jar name ends with "fat", indicating that this is a fat jar containing all the necessary libraries to run Figaro. Using a fat jar simplifies the Scala classpath needed to run Figaro programs.

Optionally, you can add the fully qualified path name of the Figaro jar to your classpath. This can be done by adding the Figaro jar to the CLASSPATH environment variable in your operating system. The process for editing the CLASSPATH varies from operating system to operating system. You can see details about using the PATH and CLASSPATH environment variables in http://docs.oracle.com/javase/tutorial/essential/environment/paths.html.

If the CLASSPATH does not exist yet, create it. It is good practice to include the current working directory, so set the CLASSPATH to ".", then proceed to add the Figaro jar, as in the next step.

By this point, the CLASSPATH already exists, so we can add the Figaro path to it. For example, on Windows 8, if figaro_2.11-4.0.0.0-fat.jar is in the "C:\Users\apfeffer" folder and the CLASSPATH is currently equal to ".", change the CLASSPATH to "C:\Users\apfeffer\figaro_2.11-4.0.0.0-fat.jar;." (replace "4.0.0.0" with the appropriate Figaro version number).

Now you can compile and run Figaro programs just like any Scala program. Put the Test program below in a file named Test.scala. First, let's assume you followed step 4 and updated the CLASSPATH.

If you run

```
scala Test.scala
```

from the directory containing Test.scala, the Scala compiler will first compile the program and then execute it. It should produce the output 1.0.

If you run

```
scalac Test.scala
```

(note the c at the end of "scalac"), the Scala compiler runs and produces .class files. You can then execute the program by running scala Test from the same directory.

If you did not follow step 4, you can set the CLASSPATH from the command line using the –cp option. For example, to compile and execute Test.scala, assuming figaro_2.11-4.0.0.0-fat.jar is in the "C:\Users\apfeffer" folder, you can run

```
scala –cp C:\Users\apfeffer\figaro_2.11-4.0.0.0-fat.jar Test.scala
```

Here's the Test program:

```
import com.cra.figaro.language._
import com.cra.figaro.algorithm.sampling._

object Test {
  def main(args: Array[String]) {
    val test = Constant("Test")
    val algorithm = Importance(1000, test)
    algorithm.start()
    println(algorithm.probability(test, "Test"))
  }
}
```

This program should output 1.0 when run.