`SingularityCE` Container for Discovery

- Userguide -

November 2025

Contents

1 Quick reference guide

Here is a list of useful links, commands and actions which detail the main steps to build a working container for Discovery and test some basic Discovery functions.

install `SingularityCE`/Apptainer

All the installation steps are defined in this installation guide. A working singularity installation is necessary in order to use the Discovery container. The definition, usage and content of the latter are described in Sect.2 while more details about `SingularityCE` are given in Sect.3.2. Note that remote HPC machines may already have a working installation of `SingularityCE` (or similar packages, such as Apptainer).

download Discovery

The Discovery package can be obtained by cloning this public git repository. Documentation regarding the installation and content of Discovery is available at the same repository.

build the `SingularityCE` container

```
$ sudo singularity build discovery_container.sif discovery_container.def
```

or, e.g.

```
$ sudo apptainer build discovery_container.sif discovery_container.def
```

NOTE: it is possible that root privileges (i.e. `sudo` privileges) are not available on the specific HPC machine in use. If that is the case, then the container can be built locally on any other machine where root privileges are available, and then copied to the desired destination on the HPC machine.

open a shell within the container

```
$ sudo singularity shell discovery_container.sif
```

or

```
$ sudo singularity shell --bind "<path>/:$HOME" discovery_container.sif
```

where `<path>` is any valid path which will be used as the `home/` directory by the container

NOTE: if root privileges (i.e. `sudo` privileges) are not available on the specific HPC machine in use, the commands above can be run without the `sudo` string.

run Discovery unit tests

Once a shell in the Discovery container is activated (see commands above) the terminal should prompt a line starting with `Singularity>`. If a working copy of Discovery is available on the computing machine and if the unit-test files (see Sect.2.3) have been already copied in the `examples/` directory within the Discovery folder, these unit tests can be simply run with:

```
Singularity>  python <unit_test>.py
```

where `<unit_test>.py` is any of the unit-test scripts described in Sect.2.3

2 The SingularityCE container for Discovery

This section describes the content and usage of the `discovery_container` container. This is the specific `SingularityCE` container developed for the usage of `Discovery`. All users of the Discovery container must have installed `SingularityCE` on their machine. A detailed guide for the installation of `SingularityCE` can be found here. At the same time, Sect.3.2.4 presents a brief summary and introduction of `SingularityCE` installation, tailored for a Debian-based OS (although instructions for Windows and MacOS systems can be retrieved in the `SingularityCE` installation guide).

Note that different container platforms than `SingularityCE` may be already installed on the specific target machine where the Discovery container will be built. In this case, it is recommended to carefully check the compatibility of such platforms with `SingularityCE` containers. As an example, the `Apptainer` platform is a fork derived from `SingularityCE`, therefore it natively supports `SingularityCE` containers.

2.1 Building the container

Provided that a working installation of `SingularityCE` is available on the host machine, the Discovery container can be built simply by executing the command:

```
$ sudo singularity build <SIF_filename>.sif discovery_container.def
```

where `<SIF_filename>` is an arbitrary name for the container Singularity Image File (or SIF hereafter, see Sect.3.2.1) while `discovery_container.def` is the container definition file (available at this github repository - provisional link?), which includes all the specifications of the appropriate software environment for Discovery, as described in 2.4. The above command builds an immutable SIF (see Sect.3.2.1 for details) of the Discovery container, which can be either executed locally or copied and executed on any remote machine.

Important notes:

1. it is likely that root privileges are not available on the target host machine where the container should be built. In that case, the `sudo` command will not be available although being necessary to build the container (as explained in the final note of Sect.3.2. In this case, the `discovery_container` can be built from the `discovery_container.def` file on any machine where root privileges are available and then copied to the target machine (e.g. a remote host, where Discovery will be run).
2. the `discovery_container` contains the instructions to build complex software environments (e.g. an Anaconda environment tailored on the requirements of Discovery). For this reason, the procedure to build the container may last several minutes or even up to a few hours.

2.2 Container usage

Once the SIF of the `discovery_container` has been built (and eventually copied to the target machine), it is possible to activate it as any standard `SingularityCE` container. Once active, the container will provide the user with a fresh installation of an operating system including all the software tools and packages needed to run Discovery. To activate the container, hence binding it to a terminal shell, it is simply necessary to execute the following command:

```
$ sudo singularity shell discovery_container.sif
```

NOTE: as for previous cases, it is possible that root privileges are not available on the machine hosting the container. In this case, the container can be executed without the `sudo` command.

The above command will bind the container to a shell on the host machine. In detail, the folder-tree inside the container will coincide with that on the host machine outside the container. As a consequence, for instance, the `$home` folder of the container will be the same as the one on the host machine. While this can be an advantage, as it allows the user to effortlessly access to files and folders as on the "usual" host (that is: outside the container), it is possible that using a different `$home` folder only inside the container may be necessary. To assign a different home folder to the container, it is possible to use:

```
$ sudo singularity shell --bind "<path>/:$HOME" discovery_container.sif
```

where `<path>` is a given folder on the host machine. As for previous cases, this command can be executed without the `sudo` string.

Once the container is installed and active, Discovery can be run inside it. To guide the user, a few Discovery unit tests (namely: a few example scripts and notebooks) can be found at this github repository. Sect. 2.3 provides a brief overview of these unit tests.

NOTE: in order to run the unit tests it is necessary to download the Discovery software and copy the unit tests within the "`examples/`" folder inside the Discovery directory.

2.3 Discovery unit tests

This repository provides five unit tests which illustrate the behavior of Discovery with respect to useful analysis procedures it can perform on Pulsar Timing Array (PTA) data. For example, these unit-tests show how to build pulsars models, add noise to them, build the likelihoods of these models and estimate their parameters with different optimization methods. The next section briefly describes each of these unit-tests.

2.3.1 Unit tests description

`cw_example`
: This script loads a sample of 5 pulsar data from the EPTA DR2 catalog and performs a parameter estimation for a nanograv-like model. The sampling is performed with GPU acceleration via the python `jax` package.

`flow_example`
: This is an example script which showcases how to performa rapid parameter estimation for typical PTA datasets. In particular, this script employs variational inference and normalizing flows, as in Vallisneri et al. (2024). This procedure is typically very slow unless GPU-acceleration is used, through the python package `jax[cuda]`. This test is performed on 15yr Nanograv data.

`likelihood_example`
: This showcases how to build different models of PTA likelihoods. Starting from a single PTA, sampling a realization of the model parameters from their priors and evaluating the likelihood for the given parameter set. It also shows how to make this model more complicate by adding noise according to different prescriptions and a timing model. It also shows how to build models for several Pulsars at the same time, producing both noise and spatially-correlated components.

`numpyro_example`
: This example shows how to perform parameter estimation by using discovery and numpyro. More in detail, it creates MCMC chains with discovery likelihoods and numpyro's NUTS sampler. In detail, this example shows that Discovery enhances the numpyro sampler thanks to the method "`to_df()`" which returns the sampler chain as a pandas Data-Frame.

`os_example`
: This shows how to build optimal statistics with Discovery. This is done by means of a global object built with the `GlobalLikelihood` method of Discovery.

2.3.2 Unit tests usage

The above examples are provided here both as ready-to-use jupyter notebooks or as their corresponding python scripts. This aims at providing a quick guide to the unexperienced user of Discovery. Both versions of the unit tests can be found in the `unit_tests/` directory once this repository has been cloned locally.

NOTE: since the unit tests require PTA data to be loaded and analyzed, before using them it is necessary to copy each single unit-test file (either the notebook or script version) within the `examples/` sub-directory of the Discovery package directory.

To execute the python scripts of the unit-tests it is sufficient to:

1. activate the container (see Sect.2.2 or the quick guide in Sect.1)
2. use the command: `$ python <script_example.py>`

where `<script_example.py>` is any of the scripts described above (e.g. `os_example.py`).

Similarly, the usage of notebooks depends on the activation of the `discovery_container` (see command 1. above, or Sect.1). Once that is ensured, different methods are available to be able to run a notebook inside the container. The most straightforward among these is to use the *firefox* browser installed within the container, which allows to execute a notebook with the following command:

```
$ exec jupyter notebook --allow-root ${PORT} ${IPNAME}
```

An alternative way to run the jupyter unit-test notebooks is to set up an `ssh` connection to the discovery container hosted on a remote machine (e.g. an HPC machine). This task is highly dependent on the method used to connect to remote hosts, such as command line interface (CLI), text editors such as vim, emacs, VScode or sublime text, just to give some examples. Since the setup of an ssh connection is significantly different for each of these methods, this user guide does not focus on their explanation and only restricts to the CLI usage of the `discovery_container`, as presented above.

2.4 Discovery container description

The `discovery_container` includes all the software and packages required to run Discovery, independently of the specific machine where this task may be performed. This section briefly details the overall content of the `discovery_container` in order to highlight its main features. For details, the exact content of the container can be understood by consulting its definition file, namely: `discovery_container.def`.

The `discovery_container` is based on an image bootstrapped from the `Docker` bootstrap agent (see Sect.3.2.2 for details). This is specifically selected to be compatible with the usage of the jax software and nvidia drivers for GPUs.

Furthermore, the `discovery_container` presented in this guide includes all the tools required for running Discovery, as well as its dependencies. Since Discovery is a python-based software, most of its dependencies are included in the container through the default installation of Miniconda and the definition of a custom conda environment (in case more details about conda/Anaconda are needed, please check the conda user guide. The conda environment built by default within the `discovery_container` is defined by means of a `.yml` file (namely: `discovery-gpu.yml`), which specifies all the packages to be installed in the conda environment. The latter is then executed at runtime, every time the `discovery_container` is activated.

As detailed in the `discovery-gpu.yml` file, the `discovery_container` is based on `python 3.10` and includes an array of common packages, such as e.g.: ipython, jupyter, numpy, pandas, scipy and others. More specific packages, needed for Discovery are also included and installed, for instance: jax, tensorflow, jaxlib[cuda], enterprise or numpyro.

3 General notes on `SingularityCE` and containers

This is just a very brief introduction about Containers and `SingularityCE` which provides a few basic concepts on which the installation and usage of the `SingularityCE` container for the Discovery software are based.

3.1 Unix containers

Generally speaking, a UNIX operating system (OS) is composed of two parts:

- the **kernel**: this is the part of the OS which interacts with the hardware and provides core system features (e.g. managing system resources, handling processes, controlling input/output, and providing a foundation for other software to ru). Kernel functions include memory management, process scheduling, and file system management
- the **user space**: this is a front-end environment used by the applications, libraries and other programs.

Traditionally, OSs have a fixed combination of kernel and user-space. Containers transform the user-space into a swap-able component, making it independent from the specific kernel which is being used. This allows a great flexibility in the portability of codes and applications across different machines, which also ensures a high degree of reproducibility of the tasks performed by this software. In other words, containers provide the tools to transform a given OS as well as its main tools and features in a self-contained environment which can be exported to different machines and replicated in a highly standardized way.

3.2 `SingularityCE`

`SingularityCE` is a container platform based on the Go language. A working installation of `SingularityCE` (or any derived platform such as Apptainer) is required in order to build and use the Discovery container presented in this user guide.
NOTE: it is possible that the specific HPC machine to be used to run the Discovery container already features an installation of `SingularityCE` (or any equivalent platform). In this case, the installation steps described in this section may be skipped.

The `SingularityCE` user-guide provides a complete description and a wide range of technical information, including a general introduction about how to install `SingularityCE` as well as how to build and use a `SingularityCE` container. Furthermore, `SingularityCE` offers a library of pre-built containers, which provide a useful starting point to build custom containers. This service is also offered by the Docker Hub container registry. Finally, a complete guide for the `SingularityCE` command line interface can be found here.

`SingularityCE` natively runs on Linux systems but can be easily run on Mac and Windows machines, as explained in the installation section of `SingularityCE`'s user guide. By default, within any new container, `SingularityCE` mounts: *i*) the `$HOME` directory, *ii*) other "standard" system directories, and *iii*) the user working directory. This implies that any application within the container will have full and direct access to

all files and software owned by the user on a specific machine. This allows the user to easily incorporate any application into their workflow on the machine hosting the `SingularityCE` container.

`SingularityCE` containers include *runscripts*, namely: user-defined scripts that define the actions a container should perform at runtime. Runscripts can be triggered with the run command or simply by calling the container as though it were an executable. `SingularityCE` containers also accept "usual" bash commands such as redirects (i.e. `>`), pipe (i.e. `|`) and similar ones.

NOTE: by default, when running `SingularityCE`, the container user is the same user as on the host machine. This means that, in order to install software inside the container (e.g. with "apt-get" on Linux-Ubuntu), `SingularityCE` *must be run with root privileges*. If it's not possible to use root privileges (e.g. on a remote HPC machine, for instance), `SingularityCE` offers the option to work "remotely" by creating a fake-root user. Details about this functionality can be found here.

3.2.1 `SingularityCE` container formats: SIF and Sandboxes

`SingularityCE` produces immutable images (i.e. non-writable files) stored in a *Singularity Image File* (SIF, hereafter). Being non-writable, these image files cannot be changed once produced. This implies that the structure of a `SingularityCE` container specified by a SIF is fixed. To allow users to test and modify the container properties without being restricted by the usage of SIFs, `SingularityCE` supports the *Sandbox* format, which is simply a container built as a directory rather than a SIF. To summarize:

`SIF`: **S**ingularity **I**mage **F**ile - a non-writable image file which specifies the structure and properties of a `SingularityCE` container

`Sandbox`: a folder which contains all the structure, properties and files of a `SingularityCE` container.

Therefore, the sandbox format is a viable option for testing a given container configuration and make changes to the container according to the user needs. This guide provides a step-to-step procedure to build a `SingularityCE` Sandbox from scratch. Note that it is always possible to switch between SIFs and Sandboxes.

Converting Sandboxes to SIFs
The options to convert a Sandbox to a SIF are as follows:

1. building a new container using a given Sandbox as template (as explained in this guide)
2. adding the `--writable` option to an existing Sandbox. On unix-based OSs this can be done with the command:
   `sudo singularity shell --writable sandbox_example/`
   where `sandbox_example` is the name of a sandbox, in this example.

NOTE: it is generally *very* recommended to build SIF containers directly from a `SingularityCE` definition file (see Sect.3.2.2 for details). This is because changes made to a writable Sandbox *before* its conversion to a SIF are not registered in the definition file of the SIF. In other words: once a given SIF is built from a template Sandbox, it will not be possible to know exactly how the template Sandbox was defined and/or if any changes were made to it before its conversion to the SIF.

Converting SIFs to Sandboxes
SIFs can be converted to Sandboxes by simply creating a new Sandbox from an existing SIF container. This can be simply done by using the SIF as a template, as explained in this guide.

3.2.2 Custom `SingularityCE` containers: definition files

A `SingularityCE` definition file is a set of blueprints that define how to build a custom container. This ensures that any specific container defined by a given combination of OS, software and instructions, can be replicated on different machines. Definition files include the OS to build, the base container to start from (if any), the specific software to install, environment variables to set at runtime, files to add from the host system (if any) and other metadata. These instructions define the OS on which the `SingularityCE` container is based, the actions taken by the container, its content, the installed software and more features. As a consequence, definition files make explicit which elements were bundled together to produce a specific container, allowing users to clearly understand the container structure and contents.

NOTE: it is *highly recommended* to build SIFs or Sandboxes starting from definition files, as this not only allows users to have a fixed, reproducible definition of their containers, but also to build in advance the specific custom settings for any new container in an organized way.

As detailed below, definition files have a specific structure divided into sections separated by keywords. The only *required* section of a definition file is the so-called *bootstrap* one. This allows users to leverage prebuilt

containers provided by public libraries (see Sect.3.2) as the basis for custom containers. In other words, within the context of definition files, the version of a prebuilt container which is used to built a new container is the bootstrap. In the following, a few of the most important and common sections of `SingularityCE` definition files are detailed. For the interested reader, this user guide provides a complete description of the definition files structure.

Structure of a definition file

`HEADER:`

The header section should be located at the beginning of the definition file. It defines the version and the OS on which the container is based. This will be the core operating system to build within the container. For example, the header can specify the Linux distribution, the specific version and the packages that must be part of the core install (e.g. borrowed from the OS of the host machine).

`Bootstrap:`

This is the only **required** section of a definition file. It determines the *bootstrap agent* that will be used to create the base-OS of the container. For instance, by using `Bootstrap: library`, a container from the public Singularity Container Library database will be downloaded and used as the base container on which the remaining features will be built. Analogously, `Bootstrap: docker` will pull a Docker Hub layer as a base container.
Notes:

- `Bootstrap` *needs* to be the first keyword in the `HEADER`.
- Depending on the *bootstrap agent* (i.e. the value assigned to `Bootstrap`), other keywords may become available. For instance, the keyword `From` becomes available when using `Bootstrap: library` and allows the user to specify the OS version to be used for the container. For example:
  ```
  Bootstrap: library
  From: ubuntu22.04
  ```
- Several possible *bootstrap agents* can be used. Among the most common are:
  - library which provides images hosted on the `SingularityCE` Container Library,
  - docker which provides images hosted on Docker Hub.
  - shub which provides images hosted on Singularity Hub,
  - oras which provides images from supported OCI registries,
  - scratch which is a flexible option for building a container from scratch.

  When unsure about which OS to use for the container, it is possible to check one of these links, look for an OS version which fits the need of the container to build and/or check the description of the pre-build images in any of the above bootstrap agents.

`BODY:`

The body of a `SingularityCE` container definition file is divided into different sections. The order with which they are written in the definition file is unimportant, although some of them are executed in a specific order. In particular, the first one to be executed is always the `%setup` section.

`%setup:`

During the build process, commands in the `%setup` section are always executed as a first step. **NOTE:** these commands are executed i) on the host system, ii) *outside* the environment defined by the container, iii) with *root privileges* and iv) after the base OS has been installed within the container. For these reasons, **commands inside the `%setup` section can potentially have disruptive effects on the host system**. In general, it is recommended to avoid using the `%setup` section whenever possible and work *inside* the container using the `%post` block instead (see below).

`%post:`

commands in the `%post` section are executed *within* the container at build time, after the base OS has been installed. Generally, the `%post` section is the place to perform installations of new libraries and applications.

`%files:`

The `%files` section allows the user to copy or create files into the container with greater safety than using the `%setup` section. The general usage of `%files` is as follows:

```
%files [from stage]
<source> [<destination>]
```

where:

- `[from stage]` is an optional statement which allows the user to copy files from previous stages of the build process (e.g. when performing a multi-stage build).
- `<source>` is either: i) a valid path to a file on the host machine, ii) a valid file-match such as a regular expression or `*filename*` or iii) a valid path defined by a previous build stage.
- `[<destination>]` is an *optional* path *inside* the current container. If `[<destination>]` is omitted, by default it will be set to `<source>`, but always *inside the container*.

`%environment:`
This section defines environment variables that will be available to the container at runtime.

`%runscript:`
This section defines standard actions to be taken every time the container is activated. These commands will *not* be run at build time. The contents of `%runscript` are written to a dedicated file within the container that is executed every time the container image is invoked. Any argument following the container name, when the latter is invoked, are passed to the `%runscript` section.

`%test:`
This section is executed at the very end of the build process. It is useful to validate the container build as soon as it's finished or to test the container functionality at any time after it is built. In the second case, the test can be performed through the `test` command:

```
$ singularity test <container_name>.def
```

`%labels:`
This section allows to add custom metadata to the container.

This guide provides a complete list of sections which can be used within the `BODY` of `SingularityCE` definition files, together with further details and specifications about each of them.

3.2.3 Building a `SingularityCE` container from a definition file

A significant part of building a `SingularityCE` container, in practice, is undertaken by the `build` command, as thoroughly explained in this overview. Similarly to any other bash command, `build` has an associated list of options which can be found here. In general terms, `build` takes a *target* as input and produces a container as output. `build` can work in two main ways:

1. produce a SIF as output (see Sect. 3.2.1). This is the *default* operational mode.
2. produce a Sandbox as output (see Sect. 3.2.1). This mode needs the option `--sandbox`.

Possible *targets* for `build` are either Uniform Resource Identifiers (URIs) or paths to an existing container, a Sandbox directory or the path to a `SingularityCE` definition file. For a complete description of how to build a `SingularityCE` container as well as all the potential uses of the `build` command check this guide.

In practical terms, a standard way to proceed in order to construct a custom `SingularityCE` container is as follows:

1. write a definition file, following the structure outlined in Sect.3.2.2 and in this guide. Broadly speaking, this step should include at least:
   - selecting a *bootstrap agent* and a OS version (see Sect.3.2.2)
   - including all the software and scripts needed inside the adequate sections of the definition file
2. use the definition file as a target for the `build` command.

The generic syntax to complete this latter task with `build` is extremely simple, as it basically reads as:

```
$ sudo singularity build <container_name>.sif <definition_file>.def
```

More detailed information can be found in the dedicated page for definition files of the `SingularityCE` user guide. The method detailed here can be used to build the `SingularityCE` container for Discovery starting from the **discovery_container.def** file, as explained in Sect.2. Nevertheless, before this step it is necessary to have a working installation of `SingularityCE` on your machine (see 3.2.4).

3.2.4 `SingularityCE` installation

All the steps to install `SingularityCE` on your machine are detailed in this installation guide. `SingularityCE` runs natively on linux systems, with minimal requirements:

- $\sim$ 180 Mb of disk space
- no specific CPU or RAM requirements, although 2Gb of RAM are recommended
- kernel: minimum $\geq$ 5.11 ($\geq$ 5.13 recommended) for full functionality. Different `SingularityCE` features may have less stringent kernel requirements (check the installation guide)

A few further software requirements must be met before installing `SingularityCE`. Since some of them are dependent on the specific linux distribution used, it is *highly recommended* to check these dependencies carefully here and to follow the correct installation procedure for the linux distribution running on the target machine here.
NOTE: since root privileges are recommended when installing `SingularityCE`, it is possible that the installation procedure on remote HPC machines cannot be performed by any of their users but instead it must be performed by the IT-management team of that specific HPC systems. Therefore, it is highly recommended to get in contact with the IT-management staff of the target HPC machine *before* installing `SingularityCE` on a remote HPC machine.

Installing `SingularityCE` on personal computers/laptops

What follows is an *indicative* step-by-step guide to get a working installation on a personal computer or laptop. *Please note that this guide is only intended to provide an indicative guidance for the installation. It is highly recommended that the users follow the steps highlighted in the* `SingularityCE` *installation guide.*
For help or support, it is possible to contact the Sylabs team at: https://www.sylabs.io/contact/
The generic steps to follow for installing `SingularityCE` on a debian-based distribution are as follows:

1. system update:

   ```
   sudo snap refresh
   sudo apt update
   sudo apt upgrade
   sudo apt autoclean
   sudo apt autoremove
   ```

2. check your kernel version to be aware of which requirements your installation will need to satisfy:

   ```
   uname -a
   ```

3. in case the usage of `SingularityCE` OCI commands is needed, then `conmon` must be installed:

   ```
   sudo apt install conmon
   ```

4. make sure to install the Go language, in case the target machine does not have it. Since `SingularityCE` may require the newest available version of Go, it is recommended to install Go from the official binaries and to follow the Go installation guide.

5. the following step presents to alternatives. Nevertheless, before performing either of the two, it is recommended to install all the dependencies of `SingularityCE` following this procedure. Once the dependencies are installed:

   - `SingularityCE` can be installed from provided RPM / Deb packages,
   - it is also possible to install `SingularityCE` from source. This installation is recommended, as it is straightforward and explained step by step in the installation guide.

   NOTE: when installing `SingularityCE` from source, any of its previous older version which may be installed must be removed by following these steps.

6. install the desired `SingularityCE` release following these instructions.

7. test your installation with these simple commands