ares Documentation

Release 0.1

Jordan Mirocha

January 28, 2017

Contents

The Accelerated Reionization Era Simulations (*ares*) code was designed to rapidly generate models for the global 21-cm signal. It can also be used as a 1-D radiative transfer code, stand-alone non-equilibrium chemistry solver, or meta-galactic radiation background calculator.

A few papers on how it works:

- 1-D radiative transfer: Mirocha et al. (2012).
- Uniform backgrounds & global 21-cm signal: Mirocha (2014).
- Parameter inference: Mirocha, Harker, & Burns (2015).
- Galaxy luminosity functions: Mirocha, Furlanetto, & Sun (2016).

Be warned: this code is still under active development – use at your own risk! Correctness of results is not guaranteed. This documentation is as much of a work in progress as is the code itself, so if you encounter gaps or errors please do let me know.

Current status:

CHAPTER 1

Quick-Start

To make sure everything is working, a quick test is to generate a realization of the global 21-cm signal using all default parameter values:

```
import ares

sim = ares.simulations.Global21cm()
sim.run()
sim.GlobalSignature()
```

See `example_gs_standard` in Examples for a more thorough introduction to this type of calculation.

CHAPTER 2

Contents

2.1 Installation

ares depends on:

- numpy
- scipy
- matplotlib

and optionally:

- python-progressbar
- hmf
- emcee
- mpi4py
- h5py
- setuptools
- mpmath
- shapely
- descartes

If you have mercurial installed, you can clone *ares* and its entire revision history via:

```
hg clone https://bitbucket.org/mirochaj/ares ares
cd ares
python setup.py install
```

If you do not have mercurial installed, and would rather just grab a tarball of the most recent version, select the Download repository option on bitbucket.

You'll need to set an environment variable which points to the *ares* install directory, e.g. (in bash)

```
export ARES=/users/<yourusername>/ares
```

ares will look in `$ARES/input` for lookup tables of various kinds. To download said lookup tables, run

```
python remote.py
```

This might take a few minutes. If something goes wrong with the download, you can run

```
python remote.py fresh
```

to get fresh copies of everything. If you're concerned that a download may have been interrupted and/or the file appears to be corrupted (strange I/O errors may indicate this), you can also just download fresh copies of the particular file you want to replace. For example, to grab a fresh initial conditions file, simply do

```
python remote.py fresh inits
```

2.1.1 *ares* branches

ares has two main branches. The first, `default`, is meant to be stable, and will only be updated with critical bug fixes or upon arrival at major development milestones. The "bleeding edge" lives in the `ares-dev` branch, and while you are more likely to find bugs in `ares-dev`, you will also find the newest features.

By default after you clone *ares* you'll be using the `default` branch. To switch, simply type:

```
hg update ares-dev
```

To switch back,

```
hg update default
```

For a discussion of the pros and cons of different branching techniques in mercurial, this article is a nice place to start.

2.1.2 *ares* versions

The first stable release of *ares* was used in Mirocha et al. (2015), and is tagged as *v0.1* in the revision history. The tag *v0.2* is associated with Mirocha, Furlanetto, & Sun (submitted to MNRAS). Note that these tags are just shortcuts to specific revisions. You can switch between them just like you would switch between branches, e.g.,

```
hg update v0.2
```

If you're unsure which version is best for you, see the Development History.

2.1.3 Don't have Python already?

If you do *not* already have Python installed, you might consider downloading yt, which has a convenient installation script that will download and install Python and many commonly-used Python packages for you. Anaconda is also good for this.

2.1.4 Help

If you encounter problems with installation or running simple scripts, first check the Troubleshooting page in the documentation to see if you're dealing with a common problem. If you don't find your problem listed there, please let me know!

2.2 Examples

2.2.1 Running Individual Simulations

- **Reionization & Global 21-cm Signal**
  - `example_gs_standard`
- **Uniform Radiation Backgrounds**
  - `example_crb_uv`
  - `example_crb_xr`
- **1-D Radiative Transfer**
  - `example_rt06_1`
  - `example_rt06_2`

2.2.2 Parameter Studies and Inference

- **Running Large Suites of Models**
  - `example_grid`
  - `example_mc_sampling`
- **Fitting and Forecasting**
  - `example_mcmc_gs`
  - `example_mcmc_lf`
- **Analyzing Sets of Models**
  - `example_inline_analysis`
  - `example_grid_analysis`
  - `example_mcmc_analysis`

2.2.3 Extensions

- `example_adv_RT_w_He`
- `example_galaxypop`
- `example_litdata`
- `example_embed_ares`

2.3 Parameters

We use keyword arguments to pass parameters around to various *ares* routines. A complete listing of parameters and their default values can be found in `ares.util.SetDefaultParameterValues.py`.

Here, we'll provide a brief description of each parameter.

- `params_grid`

- params_physics
- params_populations
- params_hmf
- params_control
- params_cosmology

For relatively complex calculations it can be difficult to know / remember which parameters are needed. Because of this, a convenience object called the *ParameterBundle* was introduced in June of 2016 to package together sets of commonly-used parameters and values. See the following page for more information on creating and using these objects:

- param_bundles

also relevant to problem initialization:

- problem_types

2.3.1 Custom Defaults

To adapt the defaults to your liking *without* modifying the source code (all defaults set in ares.util.SetDefaultParameterValues.py), open the file:

```
$HOME/.ares/defaults.py
```

which by default contains nothing:

```
pf = {}
```

To craft your own set of defaults, simply add elements to the pf dictionary. For example, if you want to use a default star-formation efficiency of 5% rather than 10%, open $HOME/.ares/defaults.py and do:

```
pf = {'fstar': 0.05}
```

That's it! Elements of pf will override the defaults listed in ares.util.SetDefaultParameterValues.py at run-time.

Alternatively, within a python script you can modify defaults by doing

```
import ares
ares.rcParams['fstar'] = 0.05
```

This is similar to how things work in matplotlib (with the matplotlibrc file and matplotlib.rcParams variable).

2.3.2 Custom Axis-Labels

You can do the analogous thing for axis labels (all defaults set in ares.util.Aesthetics.py). Open the file:

```
$HOME/.ares/labels.py
```

which by default contains nothing:

```
pf = {}
```

If you wanted to change the default axis label for the 21-cm brightness temperature, from δT_b (mK) to T_b, you would do:

```
pf = {'dTb': r'$T_b$'}
```

This change will automatically propagate to all built-in analysis routines.

2.4 Field Listing

The most fundamental quantities associated with any calculation done in ares are the gas density, species fractions and the gas temperature.

2.4.1 Species Fractions

Our naming convention is to denote ions using their chemical symbol (in lower-case), followed by the ionization state, separated by an underscore. Rather than denoting the ionization state with roman numerals, we simply use integers. For example, neutral hydrogen is *h_1* and ionized hydrogen is *h_2*.

Here is a complete listing:

- Neutral hydrogen fraction: `'h_1'`
- Ionized hydrogen fraction: `'h_2'`
- Neutral helium fraction: `'he_1'`
- Singly-ionized helium fraction: `'he_2'`
- Doubly-ionized helium fraction: `'he_3'`
- Electron fraction: `'e'`
- Gas density (in g cm^{-3}): `'rho'`

These are the default elements in the `history` dictionary, which is an attribute of all `ares.simulations` classes.

We also generally keep track of the ionization and heating rate coefficients:

- Rate coefficient for photo-ionization, `k_ion`.
- Rate coefficient for secondary ionization by photo-electrons, `k_ion2`.
- Rate coefficient for photo-heating, `k_heat`.

Each of these quantities are multi-dimensional because we store the rate coefficients for each absorbing species separately.

2.4.2 Two-Zone IGM Models

For calculations of the reionization history or global 21-cm signal, in which we use a two-zone IGM formalism, all quantities described in the previous sections keep their usual names with one important change: they now also have an *igm* or *cgm* prefix to signify which phase of the IGM they belong to. The *igm* phase is of course short for inter-galactic medium, while the *cgm* phase stands for the circum-galactic medium (really just meant to indicate gas near galaxies).

- Kinetic temperature, `igm_Tk`.
- HII region volume filling factor, `cgm_h_2`.
- Neutral fraction in the bulk IGM, `igm_h_1`.
- Heating rate in the IGM, `igm_k_heat`.
- Volume-averaged ionization rate, `cgm_k_ion`.

There are also new (passive) quantities, like the neutral hydrogen excitation (or ‘‘spin” temperature), the 21-cm brightness temperature, and the Lyman-α background intensity:

- 21-cm brightness temperature: `'igm_dTb'`.
- Spin temperature: `'igm_Ts'`.
- J_α: `'igm_Ja'`.

Each of these are only associated with the IGM grid patch, since the other phase of the IGM is assumed to be fully ionized and thus dark at 21-cm wavelengths.

2.5 Under the Hood

Super incomplete, sorry!

2.5.1 Source Populations

- `uth_pop_sfrd`
- `uth_pop_radiation`

2.5.2 Solvers

- `uth_solver_chem`

2.6 Troubleshooting

This page is an attempt to keep track of common errors and instructions for how to fix them. If you encounter a bug not listed below, fork ares on bitbucket and an issue a pull request to contribute your patch, if you have one. Otherwise, shoot me an email and I can try to help. It would be useful if you can send me the dictionary of parameters for a particular calculation. For example, if you ran a global 21-cm calculation via

```
import ares

pars = {'parameter_1': 1e6, 'parameter_2': 2} # or whatever

sim = ares.simulations.Global21cm(**pars)
sim.run()
```

and you get weird or erroneous results, pickle the parameters:

```
import pickle
f = open('problematic_model.pkl', 'wb')
pickle.dump(pars, f)
f.close()
```

and send them to me. Thanks!

> **Note:** If you’ve got a set of problematic models that you encountered while running a model grid or some such thing, check out the section on “problem realizations” in `example_grid_analysis`.

2.6.1 Plots not showing up

If when running some *ares* script the program runs to completion without errors but does not produce a figure, it may be due to your matplotlib settings. Most test scripts use `draw` to ultimately produce the figure because it is non-blocking and thus allows you to continue tinkering with the output if you'd like. One of two things is going on:

- You invoked the script with the standard Python interpreter (i.e., **not** iPython). Try running it with iPython, which will spit you back into an interactive session once the script is done, and thus keep the plot window open.
- Alternatively, your default `matplotlib` settings may have caused this. Check out your `matplotlibrc` file (in `$HOME/.matplotlibrc`) and make sure `interactive : True`.

Future versions of *ares* may use blocking commands to ensure that plot windows don't disappear immediately. Email me if you have strong opinions about this.

2.6.2 `IOError: No such file or directory`

There are a few different places in the code that will attempt to read-in lookup tables of various sorts. If you get any error that suggests a required input file has not been found, you should:

- Make sure you have set the `$ARES` environment variable. See the Installation page for instructions.
- Make sure the required file is where it should be, i.e., nested under `$ARES/input`.

In the event that a required file is missing, something has gone wrong. Run `python remote.py fresh` to download new copies of all files.

2.6.3 `LinAlgError: singular matrix`

This is known to occur in `ares.physics.Hydrogen` when using `scipy.interpolate.interp1d` to compute the collisional coupling coefficients for spin-exchange. It is due to a bug in LAPACK version 3.4.2 (see this thread). One solution is to install a newer version of LAPACK. Alternatively, you could use linear interpolation, instead of a spline, by passing `interp_cc='linear'` as a keyword argument to whatever class you're instantiating, or more permanently by adding `interp_cc='linear'` to your custom defaults file (see Parameters section for instructions).

2.6.4 21-cm Extrema-Finding Not Working

If the derivative of the signal is noisy (due to numerical artifacts, for example) then the extrema-finding can fail. If you can visually see three extrema in the global 21-cm signal but they are either absent or crazy in `ares.simulations.Global21cm.turning_points`, then this might be going on. Try setting the `smooth_derivative` parameter to a value of 0.1 or 0.2. This parameter will smooth the derivative with a boxcar of width $\Delta z =$ `smooth_derivative` before performing the extrema finding. Let me know if this happens (and under what circumstances), as it would be better to eliminate numerical artifacts than to smooth them out after the fact.

2.6.5 `AttributeError: No attribute blobs.`

This is a bit of a red herring. If you're running an MCMC fit and saving 2-D blobs, which always require you to pass the name of the function, this error occurs if you supply a function that does not exist. Check for typos and/or that the function exists where it should.

2.6.6 TypeError: __init__() got an unexpected keyword argument 'assume_sorted'

Turns out this parameter didn't exist prior to scipy version 0.14. If you update to scipy version >= 0.14, you should be set. If you're worried that upgrading scipy might break other codes of yours, you can also simply navigate to `ares/physics/Hydrogen.py` and delete each occurrence of `assume_sorted=True`, which should have no real effect (except for perhaps a very slight slowdown).

2.7 *ares* Development: Staying Up To Date

Things are changing fast! To keep up with advancements, a working knowledge of mercurial will be very useful. If you're reading this, you may already be familiar with mercurial to some degree, as its `clone` command can be used to checkout a copy of the most-up-to-date version (the ''tip" of development) from bitbucket. For example (as in Installation),

```
hg clone https://bitbucket.org/mirochaj/ares ares
cd ares
python setup.py install
```

If you don't plan on making changes to the source code, but would like to make sure you have the most up-to-date version of *ares*, you'll want to use the `hg pull` command regularly, i.e., simply type

```
hg pull
```

from anywhere within the *ares* folder. After entering your bitbucket credentials, fresh copies of any files that have been changed will be downloaded. In order to accept those updates, you should then type:

```
hg update
```

or simply `hg up` for short. Then, to re-install *ares*:

```
python setup.py install
```

If you plan on making changes to *ares*, you should fork it so that your line of development can run in parallel with the ''main line" of development. Once you've forked, you should clone a copy just as we did above. For example (note the hyperlink change),

```
hg clone https://bitbucket.org/mirochaj/ares-jordan ares-jordan
cd ares-jordan
python setup.py install
```

There are many good tutorials online, but in the following sections we'll go through the commands you'll likely be using all the time.

2.7.1 Checking the Status of your Fork

You'll typically want to know if, for example, you have changed any files recently and if so, what changes you have made. To do this, type:

```
hg status
```

This will print out a list of files in your fork that have either been modified (indicated with `M`), added (`A`), removed (`R`), or files that are not currently being tracked (`?`). If nothing is returned, it means that you have not made any changes to the code locally, i.e., you have no ''outstanding changes.''

If, however, some files have been changed and you'd like to see just exactly what changes were made, use the `diff` command. For example, if when you type `hg status` you see something like:

```
M tests/test_solver_chem_h.py
```

follow-up with:

```
hg diff tests/test_solver_chem_h.py
```

and you'll see a modified version of the file with + symbols indicating additions and − signs indicating removals. If there have been lots of changes, you may want to pipe the output of `hg diff` to, e.g., the UNIX program `less`:

```
hg diff tests/test_solver_chem_h.py | less
```

and use `u` and `d` to navigate up and down in the output.

2.7.2 Making Changes and Pushing them Upstream

If you convince yourself that the changes you've made are *good* changes, you should absolutely save them and beam them back up to the cloud. Your changes will either be:

- Modifications to a pre-existing file.
- Creation of an entirely new file.

If you've added new files to *ares*, they should get an `?` indicator when you type `hg status`, meaning they are untracked. To start tracking them, you need to add them to the repository. For example, if we made a new file `tests/test_new_feature.py`, we would do:

```
hg add tests/test_new_feature.py
```

Upon typing `hg status` again, that file should now have an `A` indicator to its left.

If you've modified pre-existing files, they will be marked `M` by `hg status`. Once you're happy with your changes, you must *commit* them, i.e.:

```
hg commit -m "Made some changes."
```

The `-m` indicates that what follows in quotes is the ''commit message" describing what you've done. Commit messages should be descriptive but brief, i.e., try to limit yourself to a sentence (or maybe two), tops. You can see examples of this in the ares commit history.

Note that your changes are still *local*, meaning the *ares* repository on bitbucket is unaware of them. To remedy that, go ahead and `push`:

```
hg push
```

You'll once again be prompted for your credentials, and then (hopefully) told how many files were updated etc.

If you get some sort of authorization error, have a look at the following file:

```
$ARES/.hg/hgrc
```

You should see something that looks like

```
[paths]
default = https://username@bitbucket.org/username/fork-name

[ui]
username = John Doe <johndoe@gmail.com>
```

If you got an authorization error, it is likely information in this file was either missing or incorrect. Remember that you won't have push access to the main *ares* repository: just your fork (hence the use of ''fork-name" above).

2.7.3 Contributing your Changes to the main repository

If you've made changes, you should let us know! The most formal way of doing so is to issue a pull request (PR), which alerts the administrators of *ares* to review your changes and pull them into the main line of *ares* development.

2.7.4 Dealing with Conflicts

Will cross this bridge when we come to it!

2.8 *ares* Development: Contributing!

If *ares* lacks functionality you're interested in, but seems to exhibit some features you'd like to make use of, adapting it to suit your purpose should (in principle) be fairly straightforward. The following sections describe how you might go about doing this.

If you end up developing something that might be useful for others and are willing to share, you should absolutely fork ares on bitbucket. Feel free to shoot me an email if you need help getting started!

2.8.1 Adding new modules: general rules

There are a few basic rules to follow in adding new modules to *ares* that should prevent major crashes. They are covered below.

Imports

First and foremost, when you write a new module you should follow the hierarchy that's already in place. Below, the pre-existing sub-modules within *ares* are listed in an order representative of that hierarchy:

- inference
- simulations
- solvers
- static
- populations, sources
- physics, util, analysis

It will hopefully be clear which sub-module your new code ought to be added to. For example, if you're writing code to fit a particular kind of dataset, you'll want to add your new module to `ares.inference`. If you're creating new kinds of source populations, `ares.populations`, and so on. If you're adding new physical constants, rate coefficients, etc., look at `ares.physics.Constants` and `ares.physics.RateCoefficients`.

Now, you'll (hopefully) be making use of at least some pre-existing capabilities of *ares*, which means your module will need to import classes from other sub-modules. There is only one rule here:

When writing a new class, let's say within sub-module X, you cannot import classes from sub-modules Y that lie above X in the hierarchy.

This is to prevent circular imports (which result in recursion errors).

Inheritance

You might also want to inherit pre-existing classes rather than simply making new instances of them in your own. For example, if creating a class to represent a new type of source population, it would be wise to inherit the `ares.populations.Population` class, which has a slew of convenience routines. More on that later.

Again, there's only one rule, which is related to the hierarchy listed in the above section:

Parent Classes (i.e., those to be inherited) must be defined either at the same level in the hierarchy as the Child Classes or below.

This follows from the rule about imports, since a class must be either defined locally or imported before it can be inherited.

2.9 Development History

ares used to exist as two separate codes: *rt1d* and *glorb*, which were introduced in Mirocha et al. (2012) and Mirocha (2014), respectively. Since then, the codes have been combined and restructured to provide a more unified framework for doing radiative transfer calculations, modeling of the global 21-cm signal, and exploring all types of parameter spaces using MCMC.

Here's an attempt to keep track of major changes to the code over time, which will be tagged in the bitbucket repository with version numbers. I haven't followed conventions for version numbering so far. Instead, I've simply tagged commits with a version number when a paper is submitted using that version of the code (e.g., v0.1 and v0.2), or when a series of noteworthy improvements or bug fixes have been made (v0.3).

2.9.1 v0.3

Not tagged yet, but a running list of updates:

- Updated to work with hmf version 2.0.1.
- Bug fix in S_α calculation for Furlanetto & Pritchard (2006): sign error in higher order terms.
- Generalized *HaloProperty* objects from version 0.2 to allow dependence on any number of arbitrary quantities. Now called *ParameterizedQuantity* object.

2.9.2 v0.2

This is the version of the code used in Mirocha, Furlanetto, & Sun (submitted).

Main (new) features:

- Can model the star-formation efficiency as a mass- and redshift-dependent quantity using *HaloProperty* objects.
- This, coupled with the *GalaxyPopulation* class, allows one to generate models of the galaxy luminosity function. Also possible to fit real datasets (using *ares.inference.FitLuminosityFunction* module).
- Creation of a *litdata* module to facilitate use of data from the literature. At the moment, this includes recent measurements of the galaxy luminosity function and stellar population synthesis models (*starburst99* and *BPASS*).
- Creation of `ParameterBundle` objects to ease the initialization of calculations.

2.9.3 v0.1

This is the version of the code used in Mirocha et al. (2015).

Main features:

- Simple physical models for the global 21-cm signal available.
- Can use * emcee to fit these models to data.