

rad

1.4.2

Generated by Doxygen 1.8.15

1 Synopsis	1
2 Namespace Index	3
2.1 Namespace List	3
3 Data Structure Index	5
3.1 Data Structures	5
4 File Index	7
4.1 File List	7
5 Namespace Documentation	9
5.1 rad Namespace Reference	9
5.1.1 Detailed Description	9
6 Data Structure Documentation	11
6.1 rad.Grid Class Reference	11
6.1.1 Detailed Description	11
6.1.2 Constructor & Destructor Documentation	12
6.1.2.1 __init__()	12
6.1.3 Member Function Documentation	12
6.1.3.1 __str__()	12
6.1.3.2 eqpart()	12
6.1.3.3 lower_bound_index()	13
6.1.4 Field Documentation	13
6.1.4.1 data	13
6.1.4.2 datafile	13
6.1.4.3 weight	14
6.2 grid_t Struct Reference	14
6.2.1 Detailed Description	14
6.3 model_t Struct Reference	14
6.3.1 Detailed Description	15
6.4 objpart_t Struct Reference	15
6.4.1 Detailed Description	16
6.4.2 Field Documentation	16
6.4.2.1 fnc	16
6.4.2.2 str	16
6.5 objvar_t Struct Reference	17
6.5.1 Detailed Description	17
6.6 rad.ParameterDependence Class Reference	17
6.6.1 Detailed Description	18
6.6.2 Constructor & Destructor Documentation	18
6.6.2.1 __init__()	18
6.6.3 Field Documentation	18

6.6.3.1 data_files	18
6.6.3.2 parameter_string	18
6.7 pmap_t Struct Reference	19
6.7.1 Detailed Description	19
6.8 rad.RadialAttentionModel Class Reference	19
6.8.1 Detailed Description	20
6.8.2 Constructor & Destructor Documentation	20
6.8.2.1 __init__()	20
6.8.3 Member Function Documentation	20
6.8.3.1 __parse_fnc__()	20
6.8.3.2 get_radius_dynamics()	21
6.8.3.3 get_wealth_dynamics()	21
6.8.3.4 model_sting()	21
6.8.3.5 save_cost_rsections()	21
6.8.3.6 save_latex_table()	21
6.8.3.7 save_learning_rsections()	22
6.8.4 Field Documentation	22
6.8.4.1 grids	22
6.8.4.2 parameters	22
6.8.4.3 specification	22
6.8.4.4 variables	22
6.9 setup_t Struct Reference	23
6.9.1 Detailed Description	23
6.10 sol_t Struct Reference	23
6.10.1 Detailed Description	24
6.11 rad.Variable Class Reference	25
6.11.1 Detailed Description	25
6.11.2 Constructor & Destructor Documentation	25
6.11.2.1 __init__()	25
6.11.3 Member Function Documentation	26
6.11.3.1 __str__()	26
6.11.3.2 save_figs()	26
6.11.3.3 surf_visual()	26
6.11.4 Field Documentation	27
6.11.4.1 data	27
6.11.4.2 datafile	27
6.11.4.3 r_grid	27
6.11.4.4 x_grid	27
7 File Documentation	29
7.1 cross_comp.h File Reference	29
7.1.1 Detailed Description	29

7.2 grid_t.h File Reference	29
7.2.1 Detailed Description	30
7.2.2 Function Documentation	30
7.2.2.1 grid_calc()	30
7.2.2.2 grid_copy()	30
7.2.2.3 grid_free()	31
7.2.2.4 grid_init()	31
7.2.2.5 grid_init_str()	31
7.2.2.6 grid_liei()	32
7.2.2.7 grid_load()	32
7.2.2.8 grid_save()	33
7.3 logger.h File Reference	33
7.3.1 Detailed Description	33
7.4 pmap_t.h File Reference	33
7.4.1 Detailed Description	34
7.4.2 Function Documentation	34
7.4.2.1 pmap_add()	34
7.4.2.2 pmap_add_double()	35
7.4.2.3 pmap_add_int()	35
7.4.2.4 pmap_cvalue()	35
7.4.2.5 pmap_find()	37
7.4.2.6 pmap_free()	37
7.4.2.7 pmap_gkey()	37
7.4.2.8 pmap_gvalue()	38
7.4.2.9 pmap_init()	38
7.4.2.10 pmap_save()	39
7.5 rad_setup.h File Reference	39
7.5.1 Detailed Description	40
7.5.2 Function Documentation	40
7.5.2.1 setup_find_last_saved()	40
7.5.2.2 setup_free()	40
7.5.2.3 setup_init()	41
7.5.2.4 setup_load()	41
7.5.2.5 setup_resume()	42
7.5.2.6 setup_save()	42
7.5.2.7 setup_solve()	43
7.6 rad_specs.h File Reference	43
7.6.1 Detailed Description	43
7.6.2 Function Documentation	44
7.6.2.1 cost()	44
7.6.2.2 radt()	44
7.6.2.3 util()	45

7.6.2.4 wltt()	45
7.7 rad_types.h File Reference	45
7.7.1 Detailed Description	46
7.7.2 Function Documentation	46
7.7.2.1 model_init()	47
7.7.2.2 model_load()	48
7.7.2.3 model_save()	48
7.7.2.4 solution_free()	49
7.7.2.5 solution_init()	49
7.7.2.6 solution_load()	50
7.7.2.7 solution_save()	50
Index	51

Chapter 1

Synopsis

This is the top level element documentation of the code that accompanies the numerical section of the article [Costly Consideration and Dynamic Choice Set Formation](#) (henceforth the article).

The code implements a simple version of the radial attention model and solves it. For more details on the model's technicalities see the [article](#). The solver is based on a concurrent re-formulation of the value function iteration algorithm. Further, it uses an adaptive search grid implementation to provide more accurate optimal control approximations.

If you use this software, please consider citing it. The permanent link of the software's release version is [. Suggested citations can be found at this link.](#)

Installation

The source code of the project can be found at this [repository](#). The easiest way to build the C applications of the project is to use [CMAKE](#). The following macros can be passed to CMAKE :

- If you want to build with debug symbols and warnings use `CMAKE_BUILD_TYPE=Debug`.
- If you want to build fully optimized for execution speed use `CMAKE_BUILD_TYPE=Release`.
- If you want to have bound checking for the grids, set `GRID_T_SAFE_MODE=1`.
- If you want the parameter maps to check if the passing keys exist, set `PMAP_T_SAFE_MODE=1`.
- Lastly, if you want the compilation to include debugging development functionality use `RAD_DEBUG=1`.

CMAKE produces four targets; three executables and one documentation target. The last one gives this documentation. The executable targets are

- `rad_msol` : Solves the radial attention model based on the saved parameterization file.
- `rad_mcont` : Resumes the solution of the model that is halted in a previous execution. This is useful when you are using the code in environments with execution-time limits such as in clusters.
- `rad_pardep` : Produces the data for the parameter dependence section of the [article](#).

The C code was compiled and tested using

- gcc version 4.8.5 20150623 (Red Hat 4.8.5-36) (GCC)
- gcc version 8.3.0 (GCC)
- gcc version 9.1.1 20190503 (Red Hat 9.1.1-1) (GCC)
- Microsoft (R) C/C++ Optimizing Compiler Version 19.16.27025.1 for x86
- Microsoft (R) C/C++ Optimizing Compiler Version 19.16.27025.1 for x64

The Python code was tested using

- Python 3.7.3
- Jupyter notebook 5.7.8

Functionality

The C code is used to approximate the solutions of the radial attention model. It also stores the solution and parameter analysis' binary data in the file system. The Python code is used to create model logic level objects from the stored binary data. The notebook is using the resulting objects to produce the tables and the figures of the [article](#). The resulting notebook can be accessed [here](#) in an html format.

Concurrency

The concurrency is written on an operating system level using low level abstractions (i.e. mutexes and locks). In UNIX based systems it is written using the POSIX Threads API [pthreads](#). In windows systems it uses the native windows [threading](#) and [synchronization](#) APIs.

Documentation

The documentation of the project can be found online [here](#) and it is also available for [downloading](#) in a PDF format. It is built using [DOXYGEN](#) and follows the 'repeat your-self documentation approach'.

Only top element functionality of the C code is documented. Drilling into lower level functions requires that you have an understanding of the radial attention model. Knowledge of the standard value function iteration algorithm is another prerequisite. For the parallelization, basic knowledge of system multithreading implementations and of the pipeline parallelization pattern is required. There are also some accompanying implementation comments in the source code that are not reported in this documentation.

The Python code is simple and its elements are briefly documented. The Jupyter notebook is also available with the documentation.

Contributors

Pantelis Karapanagiotis

Feel free to join, share, contribute, distribute.

License

The code is distributed under the MIT License. See the [LICENSE](#) file.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

rad	Python radial attention model classes	9
---------------------	---	-------------------

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

rad.Grid		
	Grid wrapper class	11
grid_t		
	Grid structure	14
model_t		
	Model structure	14
objpart_t		
	Objective function part structure	15
objvar_t		
	Objective function input structure	17
rad.ParameterDependence		
	Parameter dependence class	17
pmap_t		
	Parameter file structure	19
rad.RadialAttentionModel		
	Radial attention Model class	19
setup_t		
	Execution consolidating structure	23
sol_t		
	Solution structure	23
rad.Variable		
	Variable class	25

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

cross_comp.h	Cross compilation functionality	29
grid_t.h	Grid type and grid functionality	29
logger.h	Logging macros	33
pmap_t.h	Parameter map type and functionality	33
rad_setup.h	Radial attention model solver	39
rad_specs.h	Radial attention model functional specification	43
rad_types.h	Data type definitions and model functionality	45

Chapter 5

Namespace Documentation

5.1 rad Namespace Reference

Python radial attention model classes.

Data Structures

- class [Grid](#)
Grid wrapper class.
- class [ParameterDependence](#)
Parameter dependence class.
- class [RadialAttentionModel](#)
Radial attention Model class.
- class [Variable](#)
Variable class.

5.1.1 Detailed Description

Python radial attention model classes.

The file contains three basic classes; namely a grid, a variable and a model class. These classes provide are used as a bridge between the c applications and the Python Jupyter notebook. The c applications solve the radial attention model, generate solution approximation data and store them in the file system. These classes load the data and provide an interface for them to be used in a Python Jupyter notebook. The notebook functionality creates the tables and the figures that are used in the article.

Chapter 6

Data Structure Documentation

6.1 rad.Grid Class Reference

[Grid](#) wrapper class.

Public Member Functions

- `def __init__ (self, datafile)`
Constructor.
- `def __str__ (self)`
String representation of the grid object.
- `def eqpart (self, size)`
Get indices approximating equi-partition.
- `def lower_bound_index (self, values)`
Get the index of the lower bracket value.

Static Public Attributes

- `datafile = None`
The filename of the binary grid data file.
- `data = None`
The grid data.
- `weight = None`
The grid point weighting.

6.1.1 Detailed Description

[Grid](#) wrapper class.

Loads and holds the binary data created by the c `grid_t` structure.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 `__init__()`

```
def rad.Grid.__init__ (
    self,
    datafile )
```

Constructor.

Parameters

<i>datafile</i>	The filename of the binary grid data file.
-----------------	--

6.1.3 Member Function Documentation

6.1.3.1 `__str__()`

```
def rad.Grid.__str__ (
    self )
```

String representation of the grid object.

Returns

A string representation of the object.

6.1.3.2 `eqpart()`

```
def rad.Grid.eqpart (
    self,
    size )
```

Get indices approximating equi-partition.

The function returns a set of indices, the values of which approximate a grid's equipartition. The function calculates the values of an equipartition of the passed size for the continuous domain of the grid. For each calculated value, it locates the index that corresponds to the lowest upper bound of this value in the discretized grid's domain. The set of these indices are returned.

Parameters

<i>size</i>	The number of points in the partition.
-------------	--

Returns

An array with the indices approximating the partition points.

6.1.3.3 lower_bound_index()

```
def rad.Grid.lower_bound_index (
    self,
    values )
```

Get the index of the lower bracket value.

For each element of the passed values, it finds and returns the minimum among the indices with values that greater than the value.

Parameters

<i>values</i>	Array with values the lower bound indices of which are to be located.
---------------	---

Returns

An array with the indices of the discretized grid's lower bound.

6.1.4 Field Documentation**6.1.4.1 data**

```
rad.Grid.data = None [static]
```

The grid data.

6.1.4.2 datafile

```
rad.Grid.datafile = None [static]
```

The filename of the binary grid data file.

6.1.4.3 weight

```
rad.Grid.weight = None [static]
```

The grid point weighting.

The documentation for this class was generated from the following file:

- [rad.py](#)

6.2 grid_t Struct Reference

Grid structure.

```
#include <grid_t.h>
```

Data Fields

- short [n](#)
Number of grid points.
- double [m](#)
Minimum grid point.
- double [M](#)
Maximum grid point.
- double [w](#)
Weighting exponent.
- double * [d](#)
Data.

6.2.1 Detailed Description

Grid structure.

Grid type.

Contains data for grid allocation, creation and storage. The weighting of the grid controls the distribution of grid points over the grid's domain. The weighting is performed using a power function. The weighting parameter is expected to be positive.

The documentation for this struct was generated from the following file:

- [grid_t.h](#)

6.3 model_t Struct Reference

Model structure.

```
#include <rad_types.h>
```

Data Fields

- double [alpha](#)
Attentional costs factor.
- double [beta](#)
Discount factor.
- double [delta](#)
Memory persistence.
- double [gamma](#)
Complementarity factor.
- double [R](#)
Return.
- objpart_t [util](#)
Utility function.
- objpart_t [cost](#)
Cost function.
- objpart_t [radt](#)
Radius transition.
- objpart_t [wlrt](#)
Wealth transition.

6.3.1 Detailed Description

Model structure.

Model Type.

Contains model's parameters and objective function parts. Hooking the objective function parts is the responsibility of the user. The structure is used in binary file IO operations. The convention for the order of model parameters is that they appear in lexicographic order as fields. Function callbacks always appear in the end and the order of appearance does not matter.

The documentation for this struct was generated from the following file:

- [rad_types.h](#)

6.4 objpart_t Struct Reference

Objective function part structure.

```
#include <rad_types.h>
```

Data Fields

- double(* [fnc](#))(const objvar_t *v)
Objective function part callback.
- const char * [str](#)
Objective function part definition.

6.4.1 Detailed Description

Objective function part structure.

Objective function part type.

Contains a callback function that is used in the evaluation of the model's objective. The expected parts are the temporal utility, the attentional costs, the radius dynamics and the wealth dynamics. In addition it offers a string placeholder for storing the content of the definition of the function. The intended use is to write the content of the callback as a macro and the use `CCM_STRINGIFY` to get a string it and store it in the placeholder. See also [rad_specs.h](#)

6.4.2 Field Documentation

6.4.2.1 fnc

```
double(* objpart_t::fnc) (const objvar_t *v)
```

Objective function part callback.

A function used in the evaluation of the model's objective.

Parameters

v	Input parameters, state variables and controls
---	--

Returns

Evaluated part

6.4.2.2 str

```
const char* objpart_t::str
```

Objective function part definition.

A string of the content of the part' definition

The documentation for this struct was generated from the following file:

- [rad_types.h](#)

6.5 objvar_t Struct Reference

Objective function input structure.

```
#include <rad_types.h>
```

Data Fields

- const struct [model_st](#) * [m](#)
Model parameter data.
- double [x](#)
Current wealth state.
- double [r](#)
Current radius of attention.
- double [q](#)
Average product quantity.
- double [s](#)
Effort.

6.5.1 Detailed Description

Objective function input structure.

Objective function input type.

Contains parameters, state variables and controls used to evaluate the model's objective function. The evaluation involves evaluating the temporal utility, the attentional costs, the new radius of attention and the new wealth state.

The documentation for this struct was generated from the following file:

- [rad_types.h](#)

6.6 rad.ParameterDependence Class Reference

Parameter dependence class.

Public Member Functions

- def [__init__](#) (self, [parameter_string](#))
Constructor.
- def [save_fig](#) (self, domain_data, ylab, yval)
Create, show and save in pdf format the figures with the responses of the value function and the optimal controls on the particular parameter described by the passed the domain data.
- def [save_figs](#) (self, r_indices=None, x_indices=None)
Create, show and save in pdf format the figures with the responses of the value function and the optimal controls on value changes of each parameter.

Static Public Attributes

- `parameter_string` = None
Parameter name.
- `data_files` = None
List of data filenames.

6.6.1 Detailed Description

Parameter dependence class.

Comprises of functionality relevant for parameter dependence analysis of the radial attention model.

6.6.2 Constructor & Destructor Documentation

6.6.2.1 `__init__()`

```
def rad.ParameterDependence.__init__ (
    self,
    parameter_string )
```

Constructor.

Parameters

<code>parameter_string</code>	Parameter name.
-------------------------------	-----------------

6.6.3 Field Documentation

6.6.3.1 `data_files`

```
rad.ParameterDependence.data_files = None [static]
```

List of data filenames.

6.6.3.2 `parameter_string`

```
rad.ParameterDependence.parameter_string = None [static]
```

Parameter name.

The documentation for this class was generated from the following file:

- `rad.py`

6.7 pmap_t Struct Reference

Parameter file structure.

```
#include <pmap_t.h>
```

Data Fields

- `int n`
Number of stored parameters.
- `struct pmap_pair_st * p`
Key-value pair array.

6.7.1 Detailed Description

Parameter file structure.

Parameter file type.

Contains data for grid allocation, creation and storage. The weighting of the grid controls the distribution of grid points over the grid's domain. The weighting is performed using a power function. The weighting parameter is expected to be positive.

The documentation for this struct was generated from the following file:

- [pmap_t.h](#)

6.8 rad.RadialAttentionModel Class Reference

Radial attention Model class.

Public Member Functions

- `def __parse_fnc__(self, spec)`
Prepare functional specification string.
- `def __init__(self, save_dir)`
Constructor.
- `def model_sting(self)`
Get a brace-nested, string description of the model object.
- `def get_radius_dynamics(self)`
Calculate the radius transition and return it as a [Variable](#) object.
- `def get_wealth_dynamics(self)`
Calculate the wealth transition and return it as a [Variable](#) object.
- `def save_latex_table(self, filename)`
Create and save a latex table with the model and solution's parameterizations.
- `def save_learning_rsections(self, rsections=None)`
Create.
- `def save_cost_rsections(self, rsections=None)`
Create.

Static Public Attributes

- `data_path` = None
Model setup save directory.
- dictionary `parameters` = {}
Model's parameters.
- dictionary `specification` = {}
Model's functional specification.
- dictionary `grids` = {}
Model's grids.
- dictionary `variables` = {}
Model's variables.

6.8.1 Detailed Description

Radial attention Model class.

Loads binary data of the c setup structure. The format of the binary data is specified in the documentation of the c applications.

6.8.2 Constructor & Destructor Documentation

6.8.2.1 `__init__()`

```
def rad.RadialAttentionModel.__init__ (
    self,
    save_dir )
```

Constructor.

Parameters

<code>save_dir</code>	Model setup save directory.
-----------------------	-----------------------------

6.8.3 Member Function Documentation

6.8.3.1 `__parse_fnc__()`

```
def rad.RadialAttentionModel.__parse_fnc__ (
    self,
    spec )
```

Prepare functional specification string.

6.8.3.2 get_radius_dynamics()

```
def rad.RadialAttentionModel.get_radius_dynamics (
    self )
```

Calculate the radius transition and return it as a [Variable](#) object.

6.8.3.3 get_wealth_dynamics()

```
def rad.RadialAttentionModel.get_wealth_dynamics (
    self )
```

Calculate the wealth transition and return it as a [Variable](#) object.

6.8.3.4 model_sting()

```
def rad.RadialAttentionModel.model_sting (
    self )
```

Get a brace-nested, string description of the model object.

6.8.3.5 save_cost_rsections()

```
def rad.RadialAttentionModel.save_cost_rsections (
    self,
    rsections = None )
```

Create.

plot and save as pdf the cost functions' r sections.

6.8.3.6 save_latex_table()

```
def rad.RadialAttentionModel.save_latex_table (
    self,
    filename )
```

Create and save a latex table with the model and solution's parameterizations.

6.8.3.7 save_learning_rsections()

```
def rad.RadialAttentionModel.save_learning_rsections (
    self,
    rsections = None )
```

Create.

plot and save as pdf the learning dynamics' r sections.

6.8.4 Field Documentation

6.8.4.1 grids

```
rad.RadialAttentionModel.grids = {} [static]
```

Model's grids.

6.8.4.2 parameters

```
rad.RadialAttentionModel.parameters = {} [static]
```

Model's parameters.

6.8.4.3 specification

```
rad.RadialAttentionModel.specification = {} [static]
```

Model's functional specification.

6.8.4.4 variables

```
rad.RadialAttentionModel.variables = {} [static]
```

Model's variables.

The documentation for this class was generated from the following file:

- rad.py

6.9 setup_t Struct Reference

Execution consolidating structure.

```
#include <rad_setup.h>
```

Data Fields

- struct [model_st](#) * [m](#)
Model data.
- struct [sol_st](#) * [s](#)
Solution approximation data.
- struct concurrency_st * [c](#)
Concurrency data.

6.9.1 Detailed Description

Execution consolidating structure.

Setup type.

Setup structure

Contains pointers both model and solution data.

The documentation for this struct was generated from the following file:

- [rad_setup.h](#)

6.10 sol_t Struct Reference

Solution structure.

```
#include <rad_types.h>
```

Data Fields

- struct [grid_st](#) * [xg](#)
Wealth grid.
- struct [grid_st](#) * [rg](#)
Radius grid.
- struct [grid_st](#) * [qg](#)
Quantity grid.
- struct [grid_st](#) * [sg](#)
Effort grid.
- double [qadp](#)
Quantity grid adaptation scale.
- double [sadb](#)
Effort grid adaptation scale.
- double ** [v0](#)
Initial value function.
- double ** [v1](#)
Final value function.
- double ** [qpol](#)
Quantity policy.
- double ** [spol](#)
Effort policy.
- int [maxit](#)
Maximum number of iterations.
- double [tol](#)
Numerical error tolerance.
- double [acc](#)
Achieved numerical accuracy.
- int [it](#)
Iteration count.
- double [xbeg](#)
Execution start.
- double [xend](#)
Execution end.

6.10.1 Detailed Description

Solution structure.

Solution type.

Contains solution information. This involves discretized domain data in [grid_st](#) fields, approximations of the value function and the optimal controls, numerical method parameters, output accuracy and timing. Grids and approximation variables are allocated dynamically. Deallocation is the responsibility of the user.

The documentation for this struct was generated from the following file:

- [rad_types.h](#)

6.11 rad.Variable Class Reference

[Variable](#) class.

Public Member Functions

- `def __init__ (self, xgrid, rgrid, datafile=None, zvar=None)`
Constructor.
- `def __str__ (self)`
String representation of the variable object.
- `def surf_visual (self)`
Interactive surface visualization for Python Jupyter notebook.
- `def save_figs (self, fig_data)`
Create, show and save in pdf format a variable surface, its wealth and radius sections.

Static Public Attributes

- `datafile = None`
The filename of the binary variable data file.
- `x_grid = None`
The wealth grid data.
- `r_grid = None`
The radius grid data.
- `data = None`
The variable data.

6.11.1 Detailed Description

[Variable](#) class.

Loads and holds the binary data that approximate the optimal controls and the value functions generated by the c applications.

6.11.2 Constructor & Destructor Documentation

6.11.2.1 __init__()

```
def rad.Variable.__init__ (
    self,
    xgrid,
    rgrid,
    datafile = None,
    zvar = None )
```

Constructor.

Parameters

<i>xgrid</i>	The wealth grid data.
<i>rgrid</i>	The radius grid data.
<i>datafile</i>	The filename of the binary variable data file.
<i>zvar</i>	Variable data array.

6.11.3 Member Function Documentation**6.11.3.1 __str__()**

```
def rad.Variable.__str__ (
    self )
```

String representation of the variable object.

Returns

A string representation of the object.

6.11.3.2 save_figs()

```
def rad.Variable.save_figs (
    self,
    fig_data )
```

Create, show and save in pdf format a variable surface, its wealth and radius sections.

Parameters

<i>fig_data</i>	Figure data. Allowed keys are {angle, prefix, zlabel, rsections, rsections_points, xsections, xsections_points}
-----------------	---

6.11.3.3 surf_visual()

```
def rad.Variable.surf_visual (
    self )
```

Interactive surface visualization for Python Jupyter notebook.

6.11.4 Field Documentation

6.11.4.1 data

```
rad.Variable.data = None [static]
```

The variable data.

6.11.4.2 datafile

```
rad.Variable.datafile = None [static]
```

The filename of the binary variable data file.

6.11.4.3 r_grid

```
rad.Variable.r_grid = None [static]
```

The radius grid data.

6.11.4.4 x_grid

```
rad.Variable.x_grid = None [static]
```

The wealth grid data.

The documentation for this class was generated from the following file:

- rad.py

Chapter 7

File Documentation

7.1 cross_comp.h File Reference

Cross compilation functionality.

7.1.1 Detailed Description

Cross compilation functionality.

Contains compiler and OS specific functionality. The main implementation targets are unix based systems. This file contains the adjustments needed to support also windows targets.

7.2 grid_t.h File Reference

Grid type and grid functionality.

Data Structures

- struct [grid_t](#)
Grid structure.

Functions

- void [grid_init](#) (grid_t *g, short n, double m, double M, double w)
Grid initialization.
- void [grid_init_str](#) (grid_t *g, const char *init_str)
Grid initialization from string.
- void [grid_copy](#) (grid_t *dest, const grid_t *source)
Grid copy.
- void [grid_calc](#) (grid_t *g)
Calculates the grip points.
- int [grid_save](#) (const grid_t *g, const char *filename)
Binary save.
- int [grid_load](#) (grid_t *g, const char *filename)
Binary load.
- void [grid_free](#) (grid_t *g)
Grid deallocation.
- short [grid_liei](#) (const grid_t *g, double X)
Lower interpolation-extrapolation index.

7.2.1 Detailed Description

Grid type and grid functionality.

A grid is a structure that contains discretization data of a model's variables. It can be initialized by an `initialization string`, i.e. a string that describes how to construct the grid. It can also be initialized giving the boundary points, the weighting exponent and the number of grid points. Data allocation is dynamic and releasing memory is the responsibility of the caller.

7.2.2 Function Documentation

7.2.2.1 `grid_calc()`

```
void grid_calc (
    grid_t * g )
```

Calculates the grip points.

Grid calculation

The function expects that the data array is allocated. It also expects that $m < M$ and $w > 0$. If `GRID_T_SAFE_MODE` is true and any of the last two conditions is not satisfied, it prints a warning without breaking execution. The resulting behavior in such a case is undefined.

The distribution of grid points is calculated using a power function with exponent $g.w$. The weighting function is applied to an equidistant distribution on $[0, 1]$ and is then mapped to the grid's domain.

Parameters

<i>g</i>	Output grid object
----------	--------------------

7.2.2.2 `grid_copy()`

```
void grid_copy (
    grid_t * dest,
    const grid_t * source )
```

Grid copy.

Performs a deep copy of one grid to another.

Parameters

<i>dest</i>	Destination grid
<i>source</i>	Source grid

7.2.2.3 grid_free()

```
void grid_free (
    grid_t * g )
```

Grid deallocation.

Frees grid's data array.

Parameters

<i>g</i>	Output grid object
----------	--------------------

7.2.2.4 grid_init()

```
void grid_init (
    grid_t * g,
    short n,
    double m,
    double M,
    double w )
```

Grid initialization.

Sets up the output object's fields, allocates an array of *n* points to hold the data and calls `calc_grid()`.

Parameters

<i>g</i>	Output grid object
<i>n</i>	Number of grid points
<i>m</i>	Minimum grid point
<i>M</i>	Maximum grid point
<i>w</i>	Weighting exponent

7.2.2.5 grid_init_str()

```
void grid_init_str (
    grid_t * g,
    const char * init_str )
```

Grid initialization from string.

Parses the string using `GRID_T_INIT_STR_MASK` and calls `init_grid()` using the parsed data.

Parameters

<i>g</i>	Output grid object
<i>init_str</i>	Initialization string

7.2.2.6 `grid_liei()`

```
short grid_liei (
    const grid_t * g,
    double X )
```

Lower interpolation-extrapolation index.

Searches the grid array for the greatest domain value that is lower than the passed X value and returns the corresponding index. If the passed value is lower than the minimum grid value, then the function returns zero. If the passed value is greater than the greatest value, then it returns the previous to last index. The function assumes that the data array of the grid object has at least two values.

Parameters

<i>g</i>	Grid object
<i>X</i>	Domain value.

7.2.2.7 `grid_load()`

```
int grid_load (
    grid_t * g,
    const char * filename )
```

Binary load.

Populates the given grid structure from the grid binary file with the passed filename. The expected format of grid binary can be found in `save_grid()`.

Parameters

<i>g</i>	Grid object
<i>filename</i>	Output file name

Returns

Zero on success. If `GRID_T_SAFE_MODE`, it returns -1 if it fails to open the input file.

7.2.2.8 grid_save()

```
int grid_save (
    const grid_t * g,
    const char * filename )
```

Binary save.

Creates a grid binary file using the passed filename and stores the grid's data in it. The format of grid binary files is the following:

- The first sizeof(g->n) bytes represent the number of elements of the array as an integer.
- The next sizeof(g->w) bytes represent the weighting exponent of the array as a floating point number.
- The rest of the bytes represent the grid points as floating point numbers.

Parameters

<i>g</i>	Grid object
<i>filename</i>	Output file name

Returns

Zero on success. If GRID_T_SAFE_MODE, it returns -1 if it fails to open the output file.

7.3 logger.h File Reference

Logging macros.

7.3.1 Detailed Description

Logging macros.

To reset the logging level in a file, (re)define the macro RAD_LOG_LEVEL and include this file.

7.4 pmap_t.h File Reference

Parameter map type and functionality.

Data Structures

- struct [pmap_t](#)
Parameter file structure.

Functions

- int [pmap_init](#) (pmap_t *pmap, const char *pfilename)
Initialize parameter map.
- void [pmap_add](#) (pmap_t *pmap, const char *key, const char *val)
Add pair.
- void [pmap_add_int](#) (pmap_t *pmap, const char *key, int val)
Add pair from int value.
- void [pmap_add_double](#) (pmap_t *pmap, const char *key, double val)
Add pair from double value.
- void [pmap_save](#) (const pmap_t *pmap, const char *pfilename)
Save to file.
- void [pmap_free](#) (pmap_t *pmap)
Deallocate parameter map.
- const char * [pmap_find](#) (const pmap_t *pmap, const char *key)
Find in parameter map.
- const char * [pmap_gkey](#) (const pmap_t *pmap, int i)
Get key.
- const char * [pmap_gvalue](#) (const pmap_t *pmap, int i)
Get value.
- void [pmap_cvalue](#) (char **valbuf, const pmap_t *pmap, int i)
Copy value.

7.4.1 Detailed Description

Parameter map type and functionality.

Models parameter maps. Parameter maps are arrays of key-value parameter values. They are constructed by parsing parameter files. Parameter files are plain text files, the lines of which are key-value pairs. These pairs are parsed based on the mask defined by PMAP_PARAM_MASK.

7.4.2 Function Documentation

7.4.2.1 pmap_add()

```
void pmap_add (
    pmap_t * pmap,
    const char * key,
    const char * val )
```

Add pair.

Adds a key value pair into the parameter map.

Parameters

<i>pmap</i>	Parameter map
<i>key</i>	New key
<i>val</i>	New value

7.4.2.2 pmap_add_double()

```
void pmap_add_double (
    pmap_t * pmap,
    const char * key,
    double val )
```

Add pair from double value.

Adds a key value pair into the parameter map. The passed value is of double data type. The function converts it to string and calls [pmap_add\(\)](#). The conversion may result to data loss.

Parameters

<i>pmap</i>	Parameter map
<i>key</i>	New key
<i>val</i>	New value

7.4.2.3 pmap_add_int()

```
void pmap_add_int (
    pmap_t * pmap,
    const char * key,
    int val )
```

Add pair from int value.

Adds a key value pair into the parameter map. The passed value is of integer data type. The function converts it to string and calls [pmap_add\(\)](#).

Parameters

<i>pmap</i>	Parameter map
<i>key</i>	New key
<i>val</i>	New value

7.4.2.4 pmap_cvalue()

```
void pmap_cvalue (
    char ** valbuf,
    const pmap_t * pmap,
    int i )
```

Copy value.

Copies the value saved at the passed index to the given output buffer.

Parameters

<i>valbuf</i>	Pointer to output buffer.
<i>pmap</i>	Parameter map
<i>i</i>	Index

7.4.2.5 pmap_find()

```
const char* pmap_find (
    const pmap_t * pmap,
    const char * key )
```

Find in parameter map.

Searches for a pair with key value that matches the given key and returns the value. If no match is found, the function returns NULL.

Parameters

<i>pmap</i>	Parameter map
<i>key</i>	Key

Returns

The value that corresponds to the passed key

7.4.2.6 pmap_free()

```
void pmap_free (
    pmap_t * pmap )
```

Deallocate parameter map.

Frees the key-value allocated memory, sets the pointer to NULL and the number of pairs to zero.

Parameters

<i>pmap</i>	Parameter map
-------------	---------------

7.4.2.7 pmap_gkey()

```
const char* pmap_gkey (
```

```
const pmap_t * pmap,  
int i )
```

Get key.

Returns the key saved at the passed index.

Parameters

<i>pmap</i>	Parameter map
<i>i</i>	Index

Returns

The key string

7.4.2.8 pmap_gvalue()

```
const char* pmap_gvalue (  
    const pmap_t * pmap,  
    int i )
```

Get value.

Returns the value saved at the passed index.

Parameters

<i>pmap</i>	Parameter map
<i>i</i>	Index

Returns

The value string

7.4.2.9 pmap_init()

```
int pmap_init (  
    pmap_t * pmap,  
    const char * filename )
```

Initialize parameter map.

Initializes a parameter map object using the given file. If the function fails to open the parameter file, it returns a zero sized map. The function parses key-value pairs using PMAP_T_PARAM_MASK. The key-value array is dynamically initialized and should be deallocated by the user.

Parameters

<i>pmap</i>	Parameter map
<i>pfilename</i>	Parameter filename

Returns

Zero on success, an error code otherwise.

7.4.2.10 pmap_save()

```
void pmap_save (
    const pmap_t * pmap,
    const char * pfilename )
```

Save to file.

Creates or rewrites a file from the passed filename. The function saves the pairs of the parameter map line by line using the PMAP_T_STR_MASK.

Parameters

<i>pmap</i>	Parameter map
<i>pfilename</i>	Filename

7.5 rad_setup.h File Reference

Radial attention model solver.

Data Structures

- struct [setup_t](#)
Execution consolidating structure.

Functions

- int [setup_init](#) (setup_t *u, const char *parameter_filename, const struct [objpart_st](#) *obhparts)
Setup initialization.
- int [setup_solve](#) (setup_t *u)
Model solver.
- int [setup_resume](#) (setup_t *u)
Resume model solver.
- void [setup_load](#) (setup_t *u, const char *setup_path, const struct [objpart_st](#) *obhparts)

Load setup.

- void `setup_save` (const setup_t *u, const char *setup_path)

Save setup.

- void `setup_free` (setup_t *u)

Setup deallocation.

- int `setup_find_last_saved` (char *save_point)

Automatic last save point acquisition.

7.5.1 Detailed Description

Radial attention model solver.

7.5.2 Function Documentation

7.5.2.1 `setup_find_last_saved()`

```
int setup_find_last_saved (
    char * save_point )
```

Automatic last save point acquisition.

This is an auxiliary function that located the last setup save point in a particular folder. It is intended to be used in conjunction with the periodic iteration backup functionality of the applications. When `RAD_SAVE_CYCLE` is positive, the applications create backup binary files of the execution status in the file system every `RAD_SAVE_CYCLE` iterations. The files are stored in directories of the form 'itN', where N is replaced by the iteration count. This function searches into the applications' data path for the greatest saved iteration. It stores the located path into the passed character pointer.

Parameters

<code>save_point</code>	An output string that stores the last save point directory
-------------------------	--

Returns

Zero on success and non-zero otherwise

7.5.2.2 `setup_free()`

```
void setup_free (
    setup_t * u )
```

Setup deallocation.

Frees setup's dynamically allocated memory.

Parameters

<i>u</i>	Setup object to be destroyed.
----------	-------------------------------

7.5.2.3 setup_init()

```
int setup_init (
    setup_t * u,
    const char * parameter_filename,
    const struct objpart_st * obhparts )
```

Setup initialization.

The function is responsible for setting up a model using initialization values taken from the passed parameter file. The parameter file given should contain values for model parameter as well as parameterization for solution data, such as for instance grid specifications. The functional specification of the attentional costs, temporal utility, radius and wealth dynamics is passed separately using the obhparts variable. Pipeline calculations are performed here. Threads initialization is not performed here.

Parameters

<i>u</i>	Setup structure to be initialized.
<i>parameter_filename</i>	Input key-value, text file
<i>obhparts</i>	Function pointers to model's functional specifications

Returns

Zero on success, non-zero otherwise

See also

[pmap_t.h](#), [objpart_st](#).

7.5.2.4 setup_load()

```
void setup_load (
    setup_t * u,
    const char * setup_path,
    const struct objpart_st * obhparts )
```

Load setup.

Consolidates model and solution loading functionality. The format and naming conventions of the binary data are set in the [model_load\(\)](#) and [solution_load\(\)](#) functions. The order of the functions in the model's Inherits the convention used in model loading.

Parameters

<i>u</i>	Setup to be populated
<i>setup_path</i>	Path with stored binary model and solution data
<i>obhparts</i>	Model's functional specification

See also

[objpart_st](#)

7.5.2.5 `setup_resume()`

```
int setup_resume (
    setup_t * u )
```

Resume model solver.

The functionality is similar to [setup_solve\(\)](#). The function is intended to be used for resuming execution from a point stored in the file system. This functionality is helpful when the application is executed in environments with execution-time constraints. It assumes that the passed setup is populated using the [setup_load\(\)](#) function. In contrast to [setup_solve\(\)](#), this function does not call setup initialization routines. It rather jumps directly to iteration functionality execution.

Parameters

<i>u</i>	Model setup
----------	-------------

Returns

Zero on success, non-zero otherwise

7.5.2.6 `setup_save()`

```
void setup_save (
    const setup_t * u,
    const char * setup_path )
```

Save setup.

Consolidates model and solution saving functionality. The format and naming conventions of the saved binary data are set in the [model_save\(\)](#) and [solution_save\(\)](#) functions.

Parameters

<i>u</i>	Setup to be saved
<i>setup_path</i>	Path where binary model and solution data are to be stored

7.5.2.7 setup_solve()

```
int setup_solve (
    setup_t * u )
```

Model solver.

This is the top-level main functionality call. The function expects an initialized model setup (see [setup_init\(\)](#)). If multi-threading mode is enabled, the function initializes threading based on the pipeline allocations calculated in the setup. The function initializes the iterative solution procedure. It performs the fixed point calculation steps and checks for convergence. The iterations stops if the maximum number of iterations is reached, or if the convergence criterion is met. Then the function deallocates threads and returns.

Parameters

<i>u</i>	Model setup
----------	-------------

Returns

Zero on success, non-zero otherwise

7.6 rad_specs.h File Reference

Radial attention model functional specification.

Functions

- double [radt](#) (const struct [objvar_st](#) *v)
Radius transition.
- double [util](#) (const struct [objvar_st](#) *v)
Temporal utility.
- double [cost](#) (const struct [objvar_st](#) *v)
Attentional costs.
- double [wltt](#) (const struct [objvar_st](#) *v)
Wealth transition.

7.6.1 Detailed Description

Radial attention model functional specification.

This file contains the declarations of the radial attention example of the article. It uses exponential specification for effort costs. If you want to change the specification you have to redefine the corresponding macros of this file and re-compile.

7.6.2 Function Documentation

7.6.2.1 cost()

```
double cost (
    const struct objvar_st * v )
```

Attentional costs.

Calculates the attentional costs by

$$c(s, r') = (e^{\alpha s} - 1)(1 - \gamma r'(s, r)).$$

Parameters

<i>v</i>	Objective function data
----------	-------------------------

Returns

Calculated costs.

See also

radt(const objvar_t*)

7.6.2.2 radt()

```
double radt (
    const struct objvar_st * v )
```

Radius transition.

Calculates the next date's radius by

$$r'(s, r) = 1 - (1 - \delta r)e^{-s}.$$

Parameters

<i>v</i>	Objective function data
----------	-------------------------

Returns

Calculated next date's radius.

7.6.2.3 util()

```
double util (
    const struct objvar_st * v )
```

Temporal utility.

Calculates the temporal utility by

$$u(q, r) = r'(s, r)(1 - e^{-q}).$$

Parameters

<i>v</i>	Objective function data
----------	-------------------------

Returns

Calculated temporal utility.

See also

radt(const objvar_t*)

7.6.2.4 wltt()

```
double wltt (
    const struct objvar_st * v )
```

Wealth transition.

Calculates the next date's wealth

$$d(x, r') = R(x - r'(s, r)q).$$

Parameters

<i>v</i>	Objective function data
----------	-------------------------

Returns

Calculated costs.

See also

radt(const objvar_t*)

7.7 rad_types.h File Reference

Data type definitions and model functionality.

Data Structures

- struct [objvar_t](#)
Objective function input structure.
- struct [objpart_t](#)
Objective function part structure.
- struct [model_t](#)
Model structure.
- struct [sol_t](#)
Solution structure.

Functions

- void [model_init](#) (model_t *m, const struct [pmap_st](#) *pmap, const objpart_t *objparts)
Model initialization.
- void [model_load](#) (model_t *m, const char *model_path, const objpart_t *objparts)
Load model.
- void [model_save](#) (const model_t *m, const char *model_path)
Save model.
- void [solution_init](#) (sol_t *s, const struct [pmap_st](#) *pmap)
Initialize solution structure.
- void [solution_load](#) (sol_t *s, const char *model_path)
Load solution.
- void [solution_save](#) (const sol_t *s, const char *model_path)
Save solution.
- void [solution_free](#) (sol_t *s)
Free solution.

7.7.1 Detailed Description

Data type definitions and model functionality.

The two basic application data types are the model and solution structures. The first models the radial attention example parameters and functions. The second models the numerical approximation of solutions. The indented initialization method for types is through using parameter files (see [pmap_t](#)).

7.7.2 Function Documentation

7.7.2.1 model_init()

```
void model_init (
    model_t * m,
    const struct pmap_st * pmap,
    const objpart_t * objparts )
```

Model initialization.

Set the parameters using the passed parameter file and hooks the given function to the corresponding objective function parts. The expected order of the parts is

- util (utility function),
- cost (mental costs function),
- radt (radius transition function) and
- wltt (wealth transition function).

Parameters

<i>m</i>	Model
<i>pmap</i>	Parameter map
<i>objparts</i>	Objective function parts

See also

[solution_init\(\)](#)

7.7.2.2 `model_load()`

```
void model_load (
    model_t * m,
    const char * model_path,
    const objpart_t * objparts )
```

Load model.

Performs a binary loads of the model's parameters. The parameters are stored to the passed model object. They are read from the passed model path. The function expects that a binary model file name `model` exists in the passed directory. The model functional specification is load by the the passed objective parts objects and uses the same rules as the [model_init\(\)](#) function.

Parameters

<i>m</i>	Model object were the loaded data are stored
<i>model_path</i>	The path that contains binary saved data
<i>objparts</i>	The model's functional specification

See also

`objpart_t`, [model_init\(\)](#)

7.7.2.3 `model_save()`

```
void model_save (
    const model_t * m,
    const char * model_path )
```

Save model.

Performs a binary save of the model to he file system. The resulting resulting binary files are intended to be used by [model_load\(\)](#) to replicated the saved data. The function creates the model directory if it does not exist. Then it saves

- a binary dump file of the object and
- a text file with the model's functional specification.

Parameters

<i>m</i>	Model object to be saved
<i>model_path</i>	Save path

See also

[solution_load\(\)](#)

7.7.2.4 solution_free()

```
void solution_free (
    sol_t * s )
```

Free solution.

Disallocates a solution object. Object created using [solution_init\(\)](#) and [solution_load\(\)](#) are expected to be finalized using this function.

Parameters

<i>s</i>	Solution structure to be destroyed.
----------	-------------------------------------

7.7.2.5 solution_init()

```
void solution_init (
    sol_t * s,
    const struct pmap_st * pmap )
```

Initialize solution structure.

The function is responsible for assigning the passed values of the parameter map to solution parameters. It constructs the state and control grids. It also allocates memory to hold the value function and policy approximations.

Parameters

<i>s</i>	An uninitialized solution structure
<i>pmap</i>	A parameter map with initialization values

See also

[model_init\(\)](#), [solution_free\(\)](#)

7.7.2.6 `solution_load()`

```
void solution_load (
    sol_t * s,
    const char * model_path )
```

Load solution.

Loads binary saved solution data from the file system. The resulting solution structure creates a binary equivalent data object with the one that called the [solution_save\(\)](#) function. The function expect to find

- a global solution binary file,
- four grid binary files (wealth, radius, quantity and effort) and
- two optimal control binary files and two (current and next date's) value function binary files in the passed directory.

Parameters

<i>s</i>	Solution structure to be populated
<i>model_path</i>	Base file system directory containing saved execution data

See also

[solution_free\(\)](#)

7.7.2.7 `solution_save()`

```
void solution_save (
    const sol_t * s,
    const char * model_path )
```

Save solution.

Saves the given solution structure in binary format to the file system. The created binary files are intended to be used by [solution_load\(\)](#) to create a binary replica of the saved object. The function creates

- a text file with the model's function specifications (see [rad_specs.h](#))
- a head file with creation information in key-value pairs
- a global solution binary dump file of the solution structure,
- four grid binary dump files (wealth, radius, quantity and effort) and
- two optimal control and two (current and next date's) value function binary dump files in the passed directory.

Parameters

<i>s</i>	Solution structure to be saved
<i>model_path</i>	Base file system save directory

Index

- `__init__`
 - `rad.Grid`, [12](#)
 - `rad.ParameterDependence`, [18](#)
 - `rad.RadialAttentionModel`, [20](#)
 - `rad.Variable`, [25](#)
 - `__parse_fnc__`
 - `rad.RadialAttentionModel`, [20](#)
 - `__str__`
 - `rad.Grid`, [12](#)
 - `rad.Variable`, [26](#)
- `cost`
 - `rad_specs.h`, [44](#)
- `cross_comp.h`, [29](#)
- `data`
 - `rad.Grid`, [13](#)
 - `rad.Variable`, [27](#)
- `data_files`
 - `rad.ParameterDependence`, [18](#)
- `datafile`
 - `rad.Grid`, [13](#)
 - `rad.Variable`, [27](#)
- `eqpart`
 - `rad.Grid`, [12](#)
- `fnc`
 - `objpart_t`, [16](#)
- `get_radius_dynamics`
 - `rad.RadialAttentionModel`, [20](#)
- `get_wealth_dynamics`
 - `rad.RadialAttentionModel`, [21](#)
- `grid_calc`
 - `grid_t.h`, [30](#)
- `grid_copy`
 - `grid_t.h`, [30](#)
- `grid_free`
 - `grid_t.h`, [31](#)
- `grid_init`
 - `grid_t.h`, [31](#)
- `grid_init_str`
 - `grid_t.h`, [31](#)
- `grid_liei`
 - `grid_t.h`, [32](#)
- `grid_load`
 - `grid_t.h`, [32](#)
- `grid_save`
 - `grid_t.h`, [32](#)
- `grid_t`, [14](#)
- `grid_t.h`, [29](#)
 - `grid_calc`, [30](#)
 - `grid_copy`, [30](#)
 - `grid_free`, [31](#)
 - `grid_init`, [31](#)
 - `grid_init_str`, [31](#)
 - `grid_liei`, [32](#)
 - `grid_load`, [32](#)
 - `grid_save`, [32](#)
- `grids`
 - `rad.RadialAttentionModel`, [22](#)
- `logger.h`, [33](#)
- `lower_bound_index`
 - `rad.Grid`, [13](#)
- `model_init`
 - `rad_types.h`, [46](#)
- `model_load`
 - `rad_types.h`, [48](#)
- `model_save`
 - `rad_types.h`, [48](#)
- `model_sting`
 - `rad.RadialAttentionModel`, [21](#)
- `model_t`, [14](#)
- `objpart_t`, [15](#)
 - `fnc`, [16](#)
 - `str`, [16](#)
- `objvar_t`, [17](#)
- `parameter_string`
 - `rad.ParameterDependence`, [18](#)
- `parameters`
 - `rad.RadialAttentionModel`, [22](#)
- `pmap_add`
 - `pmap_t.h`, [34](#)
- `pmap_add_double`
 - `pmap_t.h`, [35](#)
- `pmap_add_int`
 - `pmap_t.h`, [35](#)
- `pmap_cvalue`
 - `pmap_t.h`, [35](#)
- `pmap_find`
 - `pmap_t.h`, [37](#)
- `pmap_free`
 - `pmap_t.h`, [37](#)
- `pmap_gkey`
 - `pmap_t.h`, [37](#)
- `pmap_gvalue`

- pmap_t.h, 38
- pmap_init
 - pmap_t.h, 38
- pmap_save
 - pmap_t.h, 39
- pmap_t, 19
- pmap_t.h, 33
 - pmap_add, 34
 - pmap_add_double, 35
 - pmap_add_int, 35
 - pmap_cvalue, 35
 - pmap_find, 37
 - pmap_free, 37
 - pmap_gkey, 37
 - pmap_gvalue, 38
 - pmap_init, 38
 - pmap_save, 39
- r_grid
 - rad.Variable, 27
- rad, 9
- rad.Grid, 11
 - __init__, 12
 - __str__, 12
 - data, 13
 - datafile, 13
 - eqpart, 12
 - lower_bound_index, 13
 - weight, 13
- rad.ParameterDependence, 17
 - __init__, 18
 - data_files, 18
 - parameter_string, 18
- rad.RadialAttentionModel, 19
 - __init__, 20
 - __parse_fnc__, 20
 - get_radius_dynamics, 20
 - get_wealth_dynamics, 21
 - grids, 22
 - model_sting, 21
 - parameters, 22
 - save_cost_rsections, 21
 - save_latex_table, 21
 - save_learning_rsections, 21
 - specification, 22
 - variables, 22
- rad.Variable, 25
 - __init__, 25
 - __str__, 26
 - data, 27
 - datafile, 27
 - r_grid, 27
 - save_figs, 26
 - surf_visual, 26
 - x_grid, 27
- rad_setup.h, 39
 - setup_find_last_saved, 40
 - setup_free, 40
 - setup_init, 41
- setup_load, 41
 - setup_resume, 42
 - setup_save, 42
 - setup_solve, 43
- rad_specs.h, 43
 - cost, 44
 - radt, 44
 - util, 44
 - wlft, 45
- rad_types.h, 45
 - model_init, 46
 - model_load, 48
 - model_save, 48
 - solution_free, 49
 - solution_init, 49
 - solution_load, 49
 - solution_save, 50
- radt
 - rad_specs.h, 44
- save_cost_rsections
 - rad.RadialAttentionModel, 21
- save_figs
 - rad.Variable, 26
- save_latex_table
 - rad.RadialAttentionModel, 21
- save_learning_rsections
 - rad.RadialAttentionModel, 21
- setup_find_last_saved
 - rad_setup.h, 40
- setup_free
 - rad_setup.h, 40
- setup_init
 - rad_setup.h, 41
- setup_load
 - rad_setup.h, 41
- setup_resume
 - rad_setup.h, 42
- setup_save
 - rad_setup.h, 42
- setup_solve
 - rad_setup.h, 43
- setup_t, 23
- sol_t, 23
- solution_free
 - rad_types.h, 49
- solution_init
 - rad_types.h, 49
- solution_load
 - rad_types.h, 49
- solution_save
 - rad_types.h, 50
- specification
 - rad.RadialAttentionModel, 22
- str
 - objpart_t, 16
- surf_visual
 - rad.Variable, 26

util

rad_specs.h, [44](#)

variables

rad.RadialAttentionModel, [22](#)

weight

rad.Grid, [13](#)

wlft

rad_specs.h, [45](#)

x_grid

rad.Variable, [27](#)