

# Package ‘snowfall’

April 19, 2009

**Type** Package

**Title** Easier cluster computing (based on snow).

**Version** 1.70

**Date** 2008-12-23

**Author** Jochen Knaus

**Maintainer** Jochen Knaus <jo@imbi.uni-freiburg.de>

**Description** Usability wrapper around snow for easier development of parallel R programs. This package offers e.g. extended error checks, and additional functions. All functions work in sequential mode, too, if no cluster is present or wished. Package is also designed as connector to the cluster management tool sfCluster, but can also used without it.

**Depends** snow

**Suggests** Rmpi

**License** GPL

**Repository** CRAN

**Date/Publication** 2008-12-30 16:54:53

## R topics documented:

snowfall-package . . . . .	2
snowfall-calculation . . . . .	4
snowfall-data . . . . .	6
snowfall-init . . . . .	7
snowfall-tools . . . . .	10

<b>Index</b>	<b>15</b>
--------------	-----------

---

snowfall-package *Toplevel useability wrapper for snow to make parallel programming even more easy and comfortable. All functions are able to run without cluster in sequential mode. Also snowfall works as connector to the cluster management program sfCluster, but can also run without it.*

---

## Description

**snowfall** is designed to make setup and usage of **snow** more easier. It also is made ready to work together with `sfCluster`, a ressource management and runtime observation tool for R-cluster usage.

## Details

Package: snowfall  
Type: Package  
Version: 1.61  
Date: 2008-11-01  
License: GPL

## Initialisation

Initialisation via `sfInit` must be called before the usage of any of the **snowfall** internal functions. `sfStop` stopps the current cluster. Some additional functions give access to build-in functions (like `sfParallel`, `sfCpus` etc.).

## Calculations

There are plenty of functions to execute parallel calculations via **snowfall**. Most of them are wrappers to the according **snow** functions, but there are additional functions as well. Most likely the parallel versions of the R-buildin applies are interesting: `sfLapply`, `sfSapply` and `sfApply`. For better cluster take a look at the load balanced `sfClusterApplyLB` and the function with restore possibilities: `sfClusterApplySR`.

## Tools

Various tools allow an easier access to parallel computing: `sfLibrary` and `sfSource` for loading code on the cluster, `sfExport`, `sfExportAll`, `sfRemoveAll` and `sfRemoveAll` for variable spreading on the cluster. And some more.

## sfCluster

**snowfall** is also the R-connector to the cluster management program `sfCluster`. Mostly all of the communication to this tool is done implicit and directly affecting the initialisation via `sfInit`.

Using `sfCluster` makes the parallel programming with **snowfall** even more practicable in real life environments.

For further informations about the usage of `sfCluster` look at its documentation.

### Author(s)

Jochen Knaus

Maintainer: Jochen Knaus <jo@imbi.uni-freiburg.de>,

### References

**snow** (Simple Network of Workstations):

<http://cran.r-project.org/src/contrib/Descriptions/snow.html>

`sfCluster` (Unix management tool for **snowfall** clusters):

<http://www.imbi.uni-freiburg.de/parallel>

### See Also

Snowfall Initialisation: [snowfall-init](#)

Snowfall Calculation: [snowfall-calculation](#)

Snowfall Tools: [snowfall-tools](#)

Optional links to other man pages, e.g. [snow](#)

### Examples

```
## Not run:
# Init Snowfall with settings from sfCluster
##sfInit()

# Init Snowfall with explicit settings.
sfInit( parallel=TRUE, nodes=2 )

if( sfParallel() )
  cat( "Running in parallel mode on", sfCpus(), "nodes.\n" )
else
  cat( "Running in sequential mode.\n" )

# Define some global objects.
globalVar1 <- c( "a", "b", "c" )
globalVar2 <- c( "d", "e" )
globalVar3 <- c( 1:10 )
globalNoExport <- "dummy"

# Define stupid little function.
calculate <- function( x ) {
  cat( x )
```

```

    return( 2 ^ x )
  }

# Export all global objects except globalNoExport
# List of exported objects is listed.
# Work both parallel and sequential.
sfExportAll( except=c( "globalNoExport" ) )

# List objects on each node.
sfClusterEvalQ( ls() )

# Calc something with parallel sfLapply
cat( unlist( sfLapply( globalVar3, calculate ) ) )

# Remove all variables from object.
sfRemoveAll( except=c( "calculate" ) )
## End(Not run)

```

---

snowfall-calculation

*Parallel calculation functions*

---

## Description

Parallel calculation functions. Execution is distributed automatically over the cluster. Most of this functions are wrappers for **snow** functions, but all can be used directly in sequential mode.

## Usage

```

sfClusterApply( x, fun, ... )
sfClusterApplyLB( x, fun, ... )
sfClusterApplySR( x, fun, ..., name="default", perUpdate=NULL, restore=sfRestore() )

sfClusterMap( fun, ..., MoreArgs = NULL, RECYCLE = TRUE )

sfLapply( x, fun, ... )
sfSapply( x, fun, ..., simplify = TRUE, USE.NAMES = TRUE )
sfApply( x, margin, fun, ... )
sfRapply( x, fun, ... )
sfCapply( x, fun, ... )

sfMM( a, b )

sfRestore()

```

**Arguments**

x	vary depending on function. See function details below.
fun	function to call
margin	vector specifying the dimension to use
...	additional arguments to pass to standard function
simplify	logical; see <code>sapply</code>
USE.NAMES	logical; see <code>sapply</code>
a	matrix
b	matrix
RECYCLE	see snow documentation
MoreArgs	see snow documentation
name	a character string indicating the name of this parallel execution. Naming is only needed if there are more than one call to <code>sfClusterApplySR</code> in a program.
perUpdate	a numerical value indicating the progress printing. Values range from 1 to 100 (no printing). Value means: any X percent of progress status is printed. Default (on given value 'NULL') is 5).
restore	logical indicating whether results from previous runs should be restored or not. Default is coming from <code>sfCluster</code> . If running without <code>sfCluster</code> , default is FALSE, if yes, it is set to the value coming from the external program.

**Details**

`sfClusterApply` calls each index of a given list on a separate node, so length of given list must be smaller than nodes. Wrapper for **snow** function `clusterApply`.

`sfClusterApplyLB` is a load balanced version of `sfClusterApply`. If a node finished its list segment it immediately starts with the next segment. Use this function in infrastructures with machines with different speed. Wrapper for **snow** function `clusterApplyLB`.

`sfClusterApplySR` saves intermediate results and is able to restore them on a restart. Use this function on very long calculations or it is (however) foreseeable that cluster will not be able to finish its calculations (e.g. because of a shutdown of a node machine). If your program use more than one parallelised part, argument `name` must be given with a unique name for each loop. Intermediate data is saved depending on R-filename, so restore of data must be explicit given for not confusing changes on your R-file (it is recommended to only restore on fully tested programs). If restores, `sfClusterApplySR` continues calculation after the first non-null value in the saved list. If your parallelised function can return null values, you probably want to change this.

`sfLapply`, `sfSapply` and `sfApply` are parallel versions of `lapply`, `sapply` and `apply`. The first two use an list or vector as argument, the latter an array.

`parMM` is a parallel matrix multiplication. Wrapper for **snow** function `parMM`.

*`sfRapply` and `sfCapply` are not implemented atm.*

**See Also**

See snow documentation for details on commands: [snow](#)

**Examples**

```

## Not run:
restoreResults <- TRUE

sfInit(parallel=FALSE)

## Execute in cluster or sequential.
sfLapply(1:10, exp)

## Execute with intermediate result saving and restore on wish.
sfClusterApplySR(1:100, exp, name="CALC_EXP", restore=restoreResults)
sfClusterApplySR(1:100, sum, name="CALC_SUM", restore=restoreResults)

sfStop()

##
## Small bootstrap example.
##
sfInit(parallel=TRUE, cpus=2)

require(mvna)
data(sir.adm)

sfExport("sir.adm", local=FALSE)
sfLibrary(cmpsrk)

wrapper <- function(a) {
  index <- sample(1:nrow(sir.adm), replace=TRUE)
  temp <- sir.adm[index, ]
  fit <- crr(temp$time, temp$status, temp$pnue, failcode=1, cencode=0)
  return(fit$coef)
}

result <- sfLapply(1:100, wrapper)

mean( unlist( rbind( result ) ) )
sfStop()
## End(Not run)

```

---

snowfall-data

*Internal configuration and test data*


---

**Description**

Internal configuration and test data. Only used for internal setup and testing.

**Usage**

```
config
```

```
f1
f2
sfOption
```

## Format

A matrix containing basic predefined configuration informations.

---

```
snowfall-init      Initialisation of cluster usage
```

---

## Description

Initialisation and organisation code to use **snowfall**.

## Usage

```
sfInit( parallel=NULL, cpus=NULL, type=NULL, socketHosts=NULL, restore=NULL, slaveOutfile=NULL )
sfStop( nostop=FALSE )
```

```
sfParallel()
sfCpus()
sfNodes()
sfGetCluster()
sfType()
sfSession()
sfSocketHosts()
sfSetMaxCPUs( number=32 )
```

## Arguments

<code>parallel</code>	Logical determining parallel or sequential execution. If not set values from commandline are taken.
<code>cpus</code>	Numerical amount of CPUs requested for the cluster. If not set, values from the commandline are taken.
<code>nostart</code>	Logical determining if the basic cluster setup should be skipped. Needed for nested use of <b>snowfall</b> and usage in packages.
<code>type</code>	Type of cluster. Can be 'SOCK', 'MPI', 'PVM' or 'NWS'. Default is 'SOCK'.
<code>socketHosts</code>	Host list for socket clusters. Only needed for socketmode (SOCK) and if using more than one machines (if using only your local machine (localhost) no list is needed).
<code>restore</code>	Globally set the restore behavior in the call <code>sfClusterApplySR</code> to the given value.
<code>slaveOutfile</code>	Write R slave output to this file. Default: no output (Unix: <code>/dev/null</code> , Windows: <code>:nul</code> ). If using <code>sfCluster</code> this argument has no function, as slave logs are defined using <code>sfCluster</code> .

nostop	Same as noStart for ending.
number	Amount of maximum CPUs useable.

## Details

`sfInit` initialise the usage of the **snowfall** functions and - if running in parallel mode - setup the cluster and **snow**. If using `sfCluster` management tool, call this without arguments. If `sfInit` is called with arguments, these overwrite `sfCluster` settings. If running parallel, `sfInit` set up the cluster by calling `makeCluster` from **snow**. If using with `sfCluster`, the initialisation also contains management of lockfiles. If this function is called more than once and current cluster is yet running, `sfStop` is called automatically.

Note that you should call `sfInit` before using any other function from **snowfall**, with the only exception `sfSetMaxCPUs`. If you don't call `sfInit` first, on calling an **snowfall** function `sfInit` is called without any parameters, which is equal to sequential mode in **snowfall** only mode or the settings from `sfCluster` if used with `sfCluster`.

If you use **snowfall** in a package argument `nostart` is very handy if mainprogram uses **snowfall** as well. If set, cluster setup will be skipped and both parts (package and main program) use the same cluster.

If you call `sfInit` more than one time in a program without explicit calling `sfStop`, stopping of the cluster will be executed automatically. If your R-environment does not cover required libraries, `sfInit` automatically switches to sequential mode (with a warning). Required libraries for parallel usage are **snow** and depending on argument `type` the libraries for the cluster mode (none for socket clusters, **Rmpi** for MPI clusters, **rpvm** for PVM clusters and **nws** for NetWorkSpaces).

If using Socket or NetWorkSpaces, `socketHosts` can be used to specify the hosts you want to have your workers running. Basically this is a list, where any entry can be a plain character string with IP or hostname (depending on your DNS settings). Also for real heterogenous clusters for any host paths are settable. Please look to the according **snow** documentation for details. If you are not giving an socketlist, a list with the required amount of CPUs on your local machine (localhost) is used. This would be the easiest way to use parallel computing on a single machine, like a laptop.

Note there is limit on CPUs used in one program (which can be configured on package installation). The current limit are 32 CPUs. If you need a higher amount of CPUs, call `sfSetMaxCPUs` before the first call to `sfInit`. The limit is set to prevent inadvertently request by single users affecting the cluster as a whole.

Use `slaveOutfile` to define a file where to write the log files. The file location must be available on all nodes. Beware of taking a location on a shared network drive! Under \*nix systems, most likely the directories `/tmp` and `/var/tmp` are not shared between the different machines. The default is no output file. If you are using `sfCluster` this argument have no meaning as the slave logs are always created in a location of `sfClusters` choice (depending on it's configuration).

`sfStop` stop cluster. If running in parallel mode, the LAM/MPI cluster is shut down.

`sfParallel`, `sfCpus` and `sfSession` grant access to the internal state of the currently used cluster. All three can be configured via commandline and especially with `sfCluster` as well, but given arguments in `sfInit` always overwrite values on commandline. The commandline options are `'--parallel'` (empty option. If missing, sequential mode is forced), `'--cpus=X'` (for nodes, where X is a numerical value) and `'--session=X'` (with X a string).

`sfParallel` returns a logical if program is running in parallel/cluster-mode or sequential on a single processor.

`sfCpus` returns the size of the cluster in CPUs (equals the CPUs which are useable). In sequential mode `sfCpus` returns one. `sfNodes` is a deprecated similar to `sfCpus`.

`sfSession` returns a string with the session-identification. It is mainly important if used with the `sfCluster` tool.

`sfGetCluster` gets the **snow**-cluster handler. Use for direct calling of **snow** functions.

`sfType` returns the type of the current cluster backend (if used any). The value can be `SOCK`, `MPI`, `PVM` or `NWS` for parallel modes or "- sequential -" for sequential execution.

`sfSocketHosts` gives the list with currently used hosts for socket clusters. Returns empty list if not used in socket mode (means: `sfType() != 'SOCK'`).

`sfSetMaxCPUs` enables to set a higher maximum CPU-count for this program. If you need higher limits, call `sfSetMaxCPUs` before `sfInit` with the new maximum amount.

## See Also

See snow documentation for details on commands: [snow](#)

## Examples

```
## Not run:
# Run program in plain sequential mode.
sfInit( parallel=FALSE )
stopifnot( sfParallel() == FALSE )
sfStop()

# Run in parallel mode overwriting probably given values on
# commandline.
# Executes via Socket-cluster with 4 worker processes on
# localhost.
# This is probably the best way to use parallel computing
# on a single machine, like a notebook, if you are not
# using sfCluster.
# Uses Socketcluster (Default) - which can also be stated
# using type="SOCK".
sfInit( parallel=TRUE, nodes=4 )
stopifnot( sfCpus() == 4 )
stopifnot( sfParallel() == TRUE )
sfStop()

# Run parallel mode (socket) with 4 workers on 3 specific machines.
sfInit( parallel=TRUE, nodes=4, type="SOCK",
        socketHosts=c( "biom7", "biom7", "biom11", "biom12" ) )
stopifnot( sfCpus() == 4 )
stopifnot( sfParallel() == TRUE )
sfStop()

# Hook into MPI cluster.
# Note: you can use any kind MPI cluster Rmpi supports.
sfInit( parallel=TRUE, nodes=4, type="MPI" )
sfStop()
```

```

# Hook into PVM cluster.
sfInit( parallel=TRUE, nodes=4, type="PVM" )
sfStop()

# Run in sfCluster-mode: settings are taken from commandline:
# Runmode (sequential or parallel), amount of nodes and hosts which
# are used.
sfInit()

# Session-ID from sfCluster (or XXXXXXXX as default)
session <- sfSession()

# Calling a snow function: cluster handler needed.
parLapply( sfGetCluster(), 1:10, exp )

# Same using snowfall wrapper, no handler needed.
sfLapply( 1:10, exp )

sfStop()
## End(Not run)

```

---

snowfall-tools      *Cluster tools*

---

## Description

Tools for cluster usage. Allow easier handling of cluster programming.

## Usage

```

sfLibrary( package, pos=2,
           lib.loc=NULL, character.only=FALSE,
           warn.conflicts=TRUE,
           keep.source=getOption("keep.source.pkgs"),
           verbose=getOption("verbose"), version,
           stopOnError=TRUE )
sfSource( file, encoding = getOption("encoding"), stopOnError = TRUE )
sfExport( ..., list=NULL, local=TRUE, debug=FALSE )
sfExportAll( except=NULL, debug=FALSE )

sfRemove( ..., list=NULL, master=FALSE, debug=FALSE )
sfRemoveAll( except=NULL, debug=FALSE )

sfCat( ..., sep=" ", master=TRUE )

sfClusterSplit( seq )
sfClusterCall( fun, ..., stopOnError=TRUE )
sfClusterEval( expr, stopOnError=TRUE )

```

```

sfClusterSetupRNG( type="RNGstream", ... )
sfClusterSetupRNGstream( seed=rep(12345,6), ... )
sfClusterSetupSPRNG( seed=round(2^32*runif(1)), prngkind="default", para=0, ... )

sfTest()

```

### Arguments

<code>expr</code>	expression to evaluate
<code>seq</code>	vector to split
<code>fun</code>	function to call
<code>list</code>	character vector with names of objects to export
<code>local</code>	a logical indicating if variables should taken from local scope(s) or only from global.
<code>debug</code>	a logical indicating extended information is given upon action to be done (e.g. print exported variables, print context of local variables etc.).
<code>except</code>	character vector with names of objects not to export/remove
<code>sep</code>	a character string separating elements in x
<code>master</code>	a logical indicating if executed on master as well
<code>...</code>	additional arguments to pass to standard function
<code>package</code>	name of the package. Check <code>library</code> for details.
<code>pos</code>	position in search path to load library.
<code>warn.conflicts</code>	warn on conflicts (see "library").
<code>keep.source</code>	see "library".
<code>verbose</code>	enable verbose messages.
<code>version</code>	version of library to load (see "library").
<code>encoding</code>	encoding of library to load (see "library").
<code>lib.loc</code>	a character vector describing the location of the R library trees to search through, or 'NULL'. Check <code>library</code> for details.
<code>character.only</code>	a logical indicating package can be assumed to be a character string. Check <code>library</code> for details.
<code>file</code>	filename of file to read. Check <code>source</code> for details
<code>stopOnError</code>	a logical indicating if function stops on failure or still returns. Default is TRUE.
<code>type</code>	a character determine which random number generator should be used for clusters. Allowed values are "RNGstream" for L'Ecuyer's RNG or "SPRNG" for Scalable Parallel Random Number Generators.
<code>para</code>	additional parameters for the RNGs.
<code>seed</code>	Seed for the RNG.
<code>prngkind</code>	type of RNG, see snow documentation.

## Details

The current functions are little helpers to make cluster programming easier. All of these functions also work in sequential mode without any further code changes.

`sfLibrary` loads an R-package on all nodes, including master. Use this function if slaves need this library, too. Parameters are identically to the R-build in function `library`. If a relative path is given in `lib.loc`, it is converted to an absolute path. As default `sfLibrary` stops on any error, but this can be prevented by setting `stopOnError=FALSE`, the function is returning `FALSE` then. On success `TRUE` is returned.

`sfSource` loads a sourcefile on all nodes, including master. Use this function if the slaves need the code as well. Make sure the file is accessible on all nodes under the same path. The loading is done on slaves using `source` with fixed parameters: `local=FALSE`, `chdir=FALSE`, `echo=FALSE`, so the file is loaded global without changing of directory. As default `sfSource` stops on any error, but this can be prevented by setting `stopOnError=FALSE`, the function is returning `FALSE` then. On success `TRUE` is returned.

`sfExport` exports variables from the master to all slaves. Use this function if slaves need access to these variables as well. `sfExport` features two execution modes: local and global. If using local mode (default), variables for export are searched backwards from current environment to `globalenv()`. Use this mode if you want to export local variables from functions or other scopes to the slaves. In global mode only global variables from master are exported. *Note: all exported variables are global on the slaves!* If you have many identical named variables in different scopes, use argument `debug=TRUE` to view the context the exported variable is coming from. Variables are given as their names or as a character vector with their names using argument `list`.

`sfExportAll` exports all global variables from the master to all slaves with exception of the given `list`. Use this function if you want to export mostly all variables to all slaves. Argument `list` is a character vector with names of the variables *not* to export.

`sfRemove` removes a list of global (previous exported or generated) variables from slaves and (optional) master. Use this function if there are large further unused variables left on slave. Basically this is only interesting if you have more than one explicit parallel task in your program - where the danger is slaves memory usage exceed. If argument `master` is given, the variables are removed from master as well (default is `FALSE`). Give names of variables as arguments, or use argument `list` as a character vector with the names. For deep cleaning of slave memory use `sfRemoveAll`.

`sfRemoveAll` removes all global variables from the slaves. Use this function if you want to remove mostly all variables on the slaves. Argument `list` is a character vector with names of the variables *not* to remove.

`sfCat` is a debugging function printing a message on all slaves (which appear in the logfiles).

`sfClusterSplit` splits a vector into one consecutive piece for each cluster and returns as a list with length equal to the number of cluster nodes. Wrapper for **snow** function `clusterSplit`.

`sfClusterCall` calls a function on each node and returns list of results. Wrapper for **snow** function `clusterCall`.

`sfClusterEvalQ` evaluates a literal expression on all nodes. Wrapper for **snow** function `clusterEvalQ`.

`sfTest` is a simple unit-test for most of the build in functions. It runs tests and compares the results for the correct behavior. Note there are some warnings if using, this is intended (as behavior for some errors is tested, too). use this if you are not sure all nodes are running your R-code correctly (but mainly it is implemented for development).

**See Also**

See **snow** documentation for details on wrapper-commands: [snow](#)

**Examples**

```
## Not run:
sfInit( parallel=FALSE )

## Now works both in parallel as in sequential mode without
## explicit cluster handler.
sfClusterEval( cat( "yummie\n" ) );

## Load a library on all slaves. Stop if fails.
sfLibrary( tools )
sfLibrary( "tools", character.only=TRUE ) ## Alternative.

## Execute in cluster or sequential.
sfLapply( 1:10, exp )

## Export global Var
gVar <- 99
sfExport( "gVar" )

## If there are local variables with same name which shall not
## be exported.
sfExport( "gVar", local=FALSE )

## Export local variables
var1 <- 1 ## Define global
var2 <- "a"

f1 <- function() {
  var1 <- 2
  var3 <- "x"

  f2 <- function() {
    var1 <- 3

    sfExport( "var1", "var2", "var3", local=TRUE )
    sfClusterCall( var1 ) ## 3
    sfClusterCall( var2 ) ## "a"
    sfClusterCall( var3 ) ## "x"
  }

  f2()
}

f1()

## Init random number streams (snows functions, build upon
## packages rlecuyer/rsprng).
sfClusterCall( runif, 4 )
```

```
sfClusterSetupRNG()          ## L'Ecuyer is default.
sfClusterCall( runif, 4 )

sfClusterSetupRNG( type="SPRNG", seed = 9876)
sfClusterCall( runif, 4 )

## Run unit-test on main functions.
sfTest()
## End(Not run)
```

# Index

## \*Topic **datasets**

snowfall-data, 6

## \*Topic **package**

snowfall-calculation, 4

snowfall-init, 7

snowfall-package, 2

snowfall-tools, 10

config (snowfall-data), 6

f1 (snowfall-data), 6

f2 (snowfall-data), 6

library, 12

sfApply (snowfall-calculation), 4

sfCapply (snowfall-calculation), 4

sfCat (snowfall-tools), 10

sfClusterApply  
(snowfall-calculation), 4

sfClusterApplyLB  
(snowfall-calculation), 4

sfClusterApplySR  
(snowfall-calculation), 4

sfClusterCall (snowfall-tools), 10

sfClusterEval (snowfall-tools), 10

sfClusterEvalQ (snowfall-tools),  
10

sfClusterMap  
(snowfall-calculation), 4

sfClusterSetupRNG  
(snowfall-tools), 10

sfClusterSetupRNGstream  
(snowfall-tools), 10

sfClusterSetupSPRNG  
(snowfall-tools), 10

sfClusterSplit (snowfall-tools),  
10

sfCpus (snowfall-init), 7

sfExport (snowfall-tools), 10

sfExportAll (snowfall-tools), 10

sfGetCluster (snowfall-init), 7

sfInit (snowfall-init), 7

sfLapply (snowfall-calculation), 4

sfLibrary (snowfall-tools), 10

sfMM (snowfall-calculation), 4

sfNodes (snowfall-init), 7

sfOption (snowfall-data), 6

sfParallel (snowfall-init), 7

sfRapply (snowfall-calculation), 4

sfRemove (snowfall-tools), 10

sfRemoveAll (snowfall-tools), 10

sfRestore (snowfall-calculation),  
4

sfSapply (snowfall-calculation), 4

sfSession (snowfall-init), 7

sfSetMaxCPUs (snowfall-init), 7

sfSocketHosts (snowfall-init), 7

sfSource (snowfall-tools), 10

sfStop (snowfall-init), 7

sfTest (snowfall-tools), 10

sfType (snowfall-init), 7

snow, 3, 5, 9, 13

snowfall (snowfall-package), 2

snowfall-calculation, 3

snowfall-init, 3

snowfall-tools, 3

snowfall-calculation, 4

snowfall-data, 6

snowfall-init, 7

snowfall-package, 2

snowfall-tools, 10