Flow-Conditioned Parameter Grid Tools 0.9.0 documentation

[Flow-Conditioned Parameter Grid Tools](index.html#document-index)

0.9.0

Contents:

* [Cookbook](index.html#document-cookbook)
  + [Batch FCPG Creation on an HPC](index.html#document-batch_fcpgs_hpc)
  + [Cascade FCPG results between geospatial tiles](index.html#document-cascade_fcpg)
  + [Detect Missing FCPGs](index.html#document-detect_missing)
  + [Generate TauDEM Files From An ESRI Flow Direction Grid](index.html#document-make_TauDEM_files)
* [Function Documentation](index.html#document-functions)

[Flow-Conditioned Parameter Grid Tools](index.html#document-index)

* [Docs](index.html#document-index) »
* Flow-Conditioned Parameter Grid Tools 0.9.0 documentation

# Welcome to Flow-Conditioned Parameter Grid Tools’s documentation![¶](#welcome-to-flow-conditioned-parameter-grid-tools-s-documentation)

The FCPG Tools are a Python library to make flow-conditioned parameter grids (FCPGs) by either HUC2, HUC4, or other geospatial tiling schemes. These tools can be used in a Linux HPC environment or locally on your system. These tools are written for Linux and are tested for Windows 10 using the Window’s Subsystem for Linux Ubuntu 18 LTS.

# Quick Start[¶](#quick-start)

Load the FCPGtools using import FCPGtools as fcpg.

Please refer to the [Cookbook](index.html#cookbook-label) and [Function Documentation](index.html#function-label) for examples and usage.

# Installation[¶](#installation)

Clone the repository using git clone https://code.usgs.gov/StreamStats/FCPGtools.git.

Then cd into the repository and create an Anaconda environment using the supplied FCPGtools\_env.yml file by calling conda env create -f FCPGtools\_env.yml.

Then, install the repository using pip install git+file:<Full Path to the FCPGtools repository>

For example, pip install git+file:/home/<username>/projects/FCPGtools

On a HPC system you may need to load the correct Python module before building the Anaconda environment. This might be done with module load python/anaconda3

# Dependencies[¶](#dependencies)

Dependencies for this work are largely taken care of via the Anaconda environment specified by the yml file; however, the tools do rely on [TauDEM 5.3.8](https://github.com/dtarb/TauDEM/tree/v5.3.8), which needs to be installed and visible to your conda environment.

# Citation[¶](#citation)

Barnhart, T.B., Sando, T.R., Siefken, S.A., McCarthy, P.M., Rea, A.H. (2020). Flow-Conditioned Parameter Grid Tools. U.S. Geological Survey Software Release. DOI: <https://doi.org/10.5066/P9FPZUI0>

# Disclaimers[¶](#disclaimers)

Any use of trade, firm, or product names is for descriptive purposes only and does not imply endorsement by the U.S. Government.

This software has been approved for release by the U.S. Geological Survey (USGS). Although the software has been subjected to rigorous review, the USGS reserves the right to update the software as needed pursuant to further analysis and review. No warranty, expressed or implied, is made by the USGS or the U.S. Government as to the functionality of the software and related material nor shall the fact of release constitute any such warranty. Furthermore, the software is released on condition that neither the USGS nor the U.S. Government shall be held liable for any damages resulting from its authorized or unauthorized use.

## Cookbook[¶](#cookbook)

Example scripts and work flows for common FCPG tasks on local workstations and HPC environments. These scripts have not been turned into FCPGtools functions because of the the variability in HPC and local systems that the FCPGtools may be used with.

### Batch FCPG Creation on an HPC[¶](#batch-fcpg-creation-on-an-hpc)

This is an example of batch creating FCPG grids from a folder of parameter (precipitation, air temperature, land cover, etc.) grids and a flow direction grid. This example uses two Python scripts. The first script sorts through the parameter grids and submits a SLURM job for each one and the second script, makeFCPG.py, creates the FCPG within the SLURM job.

#### Parse parameter grids and submit SLURM jobs[¶](#parse-parameter-grids-and-submit-slurm-jobs)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 | import time import sys import os  hpcAccount = 'your\_hpc\_account\_here'  # Check if system arguments were provided if len(sys.argv) > 1:  inDir = sys.argv[1] # Input directory in which to search for parameter rasters  taufdr = sys.argv[2] # Flow direction grid in tauDEM format  taufac = sys.argv[3] # Flow accumulation grid in tauDEM format  workDir = sys.argv[4] # Working directory to save intermediate files  outDir = sys.argv[5] # Output directory to save CPGs  logDir = sys.argv[6] # Directory to save slurm log files  cores = sys.argv[7] # Number of cores to use for each slurm job  accumThresh = sys.argv[8] # Number of cells in flow accumulation grid below which CPG will be set to no data  overwrite = sys.argv[9] # Whether to overwrite existing CPGs  deleteTemp = sys.argv[10] # Whether to delete temporary files else:  print('No arguments provided.')  sys.exit(1)  covList = [] #Initialize list of parameter grids  if os.path.isdir(inDir):  #Get all parameter grid files in directory  for path, subdirs, files in os.walk(inDir):  for name in files:  #Check if file is .tif, and if so add it to parameter list  if os.path.splitext(name)[1] == ".tif":  covList.append(os.path.join(path, name)) elif os.path.isfile(inDir):  #Supplied path is a single parameter grid file  covList.append(inDir) else:  print("Invalid parameter grid directory")  print("The following parameter grids were located:") [print(cov) for cov in covList]  for cov in covList: #Iterate through the parameter grids   covname = os.path.splitext(os.path.basename(cov))[0] #Get the name of the parameter   #Create batch job which runs python script  jobfile = os.path.join(workDir, "{0}.slurm".format(str(covname))) # Create path to slurm job file   with open(jobfile, 'w+') as f:    #Write slurm job details  f.writelines("#!/bin/bash\n")  f.writelines("#SBATCH --job-name=%s\n" %covname) # set the name of the job  f.writelines("#SBATCH -c 1\n") # cpus per task  f.writelines("#SBATCH -n {0}\n".format(cores)) # number of tasks  f.writelines("#SBATCH --tasks-per-node=20\n") # Set number of tasks per node  f.writelines("#SBATCH -o {0}/slurm-%A.out\n".format(logDir)) # Set log file name   f.writelines("#SBATCH -p normal\n") # the partition you want to use, for this case prod is best  f.writelines("#SBATCH --account={0}\n".format(hpcAccount)) # your account  f.writelines("#SBATCH --time=01:00:00\n") # Overestimated guess at time  f.writelines("#SBATCH --mem=128000\n") #memory in MB  f.writelines("#SBATCH --mail-type=ALL\n") # Send email only for all events  f.writelines("#SBATCH --mail-user={0}@usgs.gov\n".format(os.getlogin()))  f.writelines("#SBATCH --exclusive\n") # Require exclusive use of nodes   #Set up python environment for job  f.writelines("module load gis/TauDEM-5.3.8-gcc-mpich\n") # load TauDEM  f.writelines("module load gdal/2.2.2-gcc\n") # load gdal for use with TauDEM  f.writelines("module load python/anaconda3") # load Python 3  f.writelines("source activate FCPGtools\n") # activate the correct Python environment   #Run the Python script  f.writelines("python -u ./makeFCPG.py {0} {1} {2} {3} {4} {5} {6} {7} {8}\n".format(cov, taufdr, taufac, workDir, outDir, cores, accumThresh, overwrite, deleteTemp))    print("Launching batch job for: " + str(covname))   os.system("sbatch {0}".format(jobfile)) #Send command to console   time.sleep(5) #Wait between submitting jobs |

#### Generate a FCPG given a parameter and FDR grid[¶](#generate-a-fcpg-given-a-parameter-and-fdr-grid)

makeFCPG.py

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 | import FCPGtools as fc import os import datetime import sys  print("Starting {0}".format(datetime.datetime.now()))  # Set up Inputs paramRast = sys.argv[1] # Path to parameter raster with name in format of "source\_var\_dd\_mm\_yyyy.tif" taufdr = sys.argv[2] # Path to tauDEM flow direction grid with in format of "taufdrXXXX.tif", where XXXX is a HUC code of any length taufac = sys.argv[3] # Path to tauDEM flow accumulation grid workDir = sys.argv[4] # Path to working directory outDir = sys.argv[5] # Path to output directory for FCPG files cores = int(sys.argv[6]) # Number of cores to use  accumThresh = int(sys.argv[7]) # Number of cells in flow accumulation grid below which FCPG will be set to no data overwrite = fc.parsebool(sys.argv[8]) # Whether to overwrite CPGs or not  deleteTemp = fc.parsebool(sys.argv[9]) # Whether to delete temporary files  print("Starting FCPG process for:") print("Parameter Raster: {0}".format(paramRast)) print("Flow Driection Grid: {0}".format(taufdr)) print("Flow Accumulation Grid: {0}".format(taufac)) print("Working Directory: {0}".format(workDir)) print("Output Directory: {0}".format(outDir)) print("Number of Cores: {0}".format(cores)) print("Accumulation Threshold: {0} cells".format(accumThresh)) print("Overwrite Existing CPG: {0}".format(overwrite)) print("Delete Temporary Files: {0}".format(deleteTemp))  #Get name of input parameter without extention paramName = os.path.splitext(os.path.basename(paramRast))[0]   #Get HUC number from tau flow direction raster name try:  HUC = os.path.splitext(os.path.basename(taufdr))[0].split("taufdr")[1] except:  print("Error - Flow direction raster has inappropriate name")  #Prepare some file paths to things which will be created rprjFile = os.path.join(workDir, paramName + "\_HUC" + HUC + "rprj.tif") #Create filepath for reprojected parameter file accumFile = os.path.join(workDir, paramName + "\_HUC" + HUC + "accum.tif") #Create filepath for accumulated parameter file nodataFile = os.path.join(workDir, paramName + "\_HUC" + HUC + "nodata.tif") #Create filepath for parameter no data file nodataaccumFile = os.path.join(workDir, paramName + "\_HUC" + HUC + "accumnodata.tif") #Create filepath for parameter accumulated no data file zeronodataFile = os.path.join(workDir, paramName + "\_HUC" + HUC + "zeronodata.tif") #Create filepath for parameter zeroed no data file CPGFile = os.path.join(outDir, paramName + "\_HUC" + HUC +"\_FCPG.tif") #Create filepath for parameter FCPG file  if os.path.isfile(CPGFile) & (overwrite == False):  print("Error: Specified FCPG file exists and will not be overwritten") else:  #Run the FCPG tools  print("Calling resample function {0}".format(datetime.datetime.now()))  fc.resampleParam(paramRast, taufdr, rprjFile, resampleMethod="bilinear", cores=cores) #Resample and reprojected parameter raster  print("Calling flow accumulation function {0}".format(datetime.datetime.now()))  fc.accumulateParam(rprjFile, taufdr, accumFile, outNoDataRast=nodataFile, outNoDataAccum=nodataaccumFile, zeroNoDataRast=zeronodataFile, cores=cores) #Accumulate parameter  print("Calling make\_cpg function {0}".format(datetime.datetime.now()))  if os.path.isfile(nodataaccumFile):  #If no data accumulation file was created, use it in call to create FCPG  fc.make\_cpg(accumFile, taufac, CPGFile, noDataRast=nodataaccumFile, minAccum=accumThresh) #Create parameter FCPG with no data raster.  else:  fc.make\_cpg(accumFile, taufac, CPGFile, minAccum=accumThresh) #Create parameter FCPG without no data raster.    if deleteTemp:  try:  #Delete temporary files  os.remove(rprjFile)  os.remove(accumFile)  os.remove(nodataFile)  os.remove(nodataaccumFile)  except:  print("Warning: Unable to delete temporary files") print("Finished {0}".format(datetime.datetime.now())) |

### Cascade FCPG results between geospatial tiles[¶](#cascade-fcpg-results-between-geospatial-tiles)

Watersheds or hydrologic units, e.g. the [Watershed Boundary Dataset](https://www.usgs.gov/core-science-systems/ngp/national-hydrography/watershed-boundary-dataset?qt-science_support_page_related_con=4#qt-science_support_page_related_con), are often used as geospatial tiling schemes for digital elevation models. These pose a challenge for FCPGs as downstream accumulated area and parameter grids must be corrected with values from upstream geospatial tiles to be accurate. This section contains examples for cascading flow between 4-digit hydrologic units and larger 2-digit hydrologic units or other geospatial tiling schemes.

#### Cascade Four-Digit Hydrologic Regions[¶](#cascade-four-digit-hydrologic-regions)

This script provides an example work flow of how to cascade the last (maximum) accumulated area values from an upstream basin, e.g. HUC 1002 – Upper Missouri, to the first cell of a downstream basin, e.g. HUC 1003 – Missouri-Marias. The process creates a json file containing the values to be cascaded, creates a weighting grid of ones for the downstream region, inserts the upstream value into the weighting grid at the correct location, and accumulates the weighting grid using the downstream flow direction grid to produce an adjusted flow accumulation grid for the downstream region. This process can be repeated for each parameter grid and then used with make\_fcpg() to create FCPGs corrected for upstream basins.

##### Example HUC-4 Cascading Work Flow[¶](#example-huc-4-cascading-work-flow)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 | import FCPGtools as fcpg # Import the tools.  import geopandas as gpd  # Define the file input and output paths and the upstream region. upstreamFACfltau = '' # Upstream flow accumulation grid from TauDEM. upstreamFDRfltau = '' # Upstream flow direction grid in TauDEM format. downstreamFACfltau = '' # Downstream flow accumuation grid from TauDEM. downstreamFDRfltau = '' # Downstream flow direction grid in TauDEM format. region = '' # Hydrologic region to make the update dictionary for. updateDict = '' # Path to the update dictionary to create. downstreamFACWeight = '' # Path to the downstream FAC weighting grid. This is where the upstream value will be inserted. downstreamAdjFAC = '' # Path to output the adjusted FAC weighting grid. wbdPth = '' # Path to the watershed boundary dataset for the HUC-2 region you are working in. dstCRS = '' # Proj4 represenation of the projection that the FCPGs will be in.  # Define upstream and downstream basins. upstream = '' downstream = ''  # Load the WBD. layer = 'WBDHU12' # HUC12 WBD layer with ToHUC codes. wbd = gpd.read\_file(wbdPth, layer = layer) # Load the WBD to a geodataframe.  # Convert HUC12 and ToHUC codes to 4-digit codes., wbd['HUC4'] = wbd.HUC12.map(fcpg.getHUC4) wbd['ToHUC4'] = wbd.ToHUC.map(fcpg.getHUC4)  # Find basins that contain pour points. pourBasins = fcpg.makePourBasins(wbd,upstream,downstream)  # Find pour points between the upstream and downstream basins. pourPoints = fcpg.findPourPoints(pourBasins, upstreamFACfltau, upstreamFDRfltau, plotBasins = True)  # create update dictionary from the pour points. newX,newY,maxFAC = zip(\*pourPoints) fcpg.createUpdateDict(newX,newY,maxFAC,upstream,updateDict)  # Update the downstream basin using the update dictionary. fcpg.adjustFAC(downstreamFACfltau,downstreamFACWeight,updateDict,downstreamFDRfltau,downstreamAdjFAC, cores=6) # note that this tool will create downstreamFACWeight if it does not exist based on downstreamFACfltau, see function documentation. |

#### Cascade Two-Digit Hydrologic Regions[¶](#cascade-two-digit-hydrologic-regions)

This script provides an example work flow of how to cascade the last (maximum) accumulated area value from an upstream region, e.g. Region 14 – Upper Colorado, to the first cell of a downstream region, e.g. Region 15 – Lower Colorado. The process creates a json file containing the value to be cascaded, creates a weighting grid of ones for the downstream region, inserts the upstream value into the weighting grid at the correct location, and accumulates the weighting grid using the downstream flow direction grid to produce an adjusted flow accumulation grid for the downstream region. This process can be repeated for each parameter grid and then used with make\_fcpg() to create FCPGs corrected for upstream regions.

##### Example HUC-2 Cascading Work Flow[¶](#example-huc-2-cascading-work-flow)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | import FCPGtools as fcpg # Import the tools.   # Define the file input and output paths and the upstream region. upstreamFDRflesri = '' # Upstream flow direction grid in ESRI format. downstreamFDRflesri = '' # Downstream flow direction grid in ESRI format. upstreamFACfltau = '' # Upstream flow accumulation grid from TauDEM. upstreamFDRfltau = '' # Upstream flow direction grid in TauDEM format. downstreamFACfltau = '' # Downstream flow accumuation grid from TauDEM. downstreamFDRfltau = '' # Downstream flow direction grid in TauDEM format. region = '' # Hydrologic region to make the update dictionary for. updateDict = '' # Path to the update dictionary to create. downstreamFACWeight = '' # Path to the downstream FAC weighting grid. This is where the upstream value will be inserted. downstreamAdjFAC = '' # Path to output the adjusted FAC weighting grid.  # convert ESRI flow directions to TauDEM flow directions. fcpg.tauDrainDir(upstreamFDRflesri, upstreamFDRfltau) fcpg.tauDrainDir(downstreamFDRflesri, downstreamFDRfltau)  # Accumulate upstream FDR grid. fcpg.tauFlowAccum(upstreamFDRfltau, upstreamFACfltau)  # Create the update dictionary to move data between HUC2 regions. x,y,d,w = fcpg.findLastFACFD(upstreamFACfltau, upstreamFACfltau) fcpg.createUpdateDict(x,y,d,region,updateDict)  # Make a grid of ones based on the downstream grid. fcpg.makeFACweight(downstreamFDRfltau,downstreamFACWeight)  # Create the updated flow accumulation grid with information from the upstream HUC inserted into the source grid. fcpg.adjustFAC(downstreamFDRfltau,downstreamFACWeight,  updateDict,downstreamFDRfltau,downstreamAdjFAC,cores=8) |

### Detect Missing FCPGs[¶](#detect-missing-fcpgs)

Helper script to iterate through geospatial tiles, HUC4 basins here, and to check for completed FCPGs. This is useful if you have many FCPGs to process.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 | import os  HUClist = ["1002", "1003", "1004"] # HUC4 geospatial tiles to search over.  inDir = "../data/cov/static" # Source parameter grid folder.  CPGdir = "../FCPGs" # Output FCPG folder.  covList = [] #Initialize list of parameter grids.  # iterate through all source parameter grids. if os.path.isdir(inDir):   for path, subdirs, files in os.walk(inDir):  for name in files:  #Check if file is .tif, and if so add it to covariate list  if os.path.splitext(name)[1] == ".tif" or os.path.splitext(name)[1] == ".vrt":  covList.append(os.path.join(path, name))   print("The following covariate files were located in the specified directory:")  print(covList)   missingList = [] #Initialize list of missing files   # iterate through source parameter grids and test if FCPGs have been created.  for cov in covList:   covname = os.path.splitext(os.path.basename(cov))[0] #Get the name of the parameter grid   if os.path.isdir(CPGdir):  for HUC in HUClist:  #Create the file name corresponding to the HUC and parameter grid  CPGFile = os.path.join(CPGdir, HUC,covname + "\_HUC" + HUC +"\_FCPG.tif") #Create filepath for parameter CPG file   if not os.path.isfile(CPGFile):  print("Missing File: {0}".format(CPGFile))  missingList.append(CPGFile)     else:  print("Error FCPG directory does not exist: {0}".format(CPGdir))    print("{0} missing files found".format(len(missingList)))  else:  print("Error input directory does not exist: {0}".format(inDir)) |

### Generate TauDEM Files From An ESRI Flow Direction Grid[¶](#generate-taudem-files-from-an-esri-flow-direction-grid)

Helper script to reclassify an ESRI flow direction grid to TauDEM flow directions and generate a TauDEM flow accumulation grid.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | import fcpgtools as fcpg  cores = 6 # Computer cores to use. fdr = '' # Input ESRI flow direction grid. fdrTau = '' # Output TauDEM formatted flow direction grid. facTau = '' # Output flow accumulation grid.   fcpg.tauDrainDir(fdr, taufdr) # Reclassify flow directions. fcpg.tauFlowAccum(taufdr, taufac, cores=cores) # Compute flow accumulation. |

## Function Documentation[¶](#module-tools)

ExtremeUpslopeValue(*fdr*, *param*, *output*, *accum\_type='MAX'*, *cores=1*, *fac=None*, *thresh=None*)[¶](#tools.ExtremeUpslopeValue)

Wrapper for the TauDEM extreme upslope value function.

|  |  |
| --- | --- |
| Parameters: | **fdr** : str  Path to a flow direction grid in TauDEM format.  **param** : str  Path to parameter raster to run through the D8 Extreme Upslope Value tool  **output** : str  Path to output raster file.  **accum\_type** : str (optional)  Either “MAX” or “MIN.” Defaults to “MAX.”  **cores** : int (optional)  Number of cores to run this process on. Defaults to 1.  **fac** : str (optional)  Path to a flow accumulation raster. Defaults to None.  **thresh** : int (optional)  Threshold values, in the same units as fac to mask output to stream channels. Defaults to None. |
| Returns: | **output** : raster  Raster of either the maximum or minumum upslope value of the parameter supplied to the function. |

FindDownstreamCellTauDir(*d*, *x*, *y*, *w*)[¶](#tools.FindDownstreamCellTauDir)

Find downstream cell given the flow direction of a cell using TauDEM flow directions.

|  |  |
| --- | --- |
| Parameters: | **d** : int  Flow direction of cell to find downstream cell of.  **x** : float  Horizontal coordinate (either projected or unprojected).  **y** : float  Vertical coordinate (either projected or unprojected).  **w** : float  Cell size in map units. |
| Returns: | **x** : float  Horizontal coordinate of the downstream cell.  **y** : float  Verital coordinate of the downstream cell. |

accumulateParam(*paramRast*, *fdr*, *accumRast*, *outNoDataRast=None*, *outNoDataAccum=None*, *zeroNoDataRast=None*, *cores=1*)[¶](#tools.accumulateParam)

|  |  |
| --- | --- |
| Parameters: | **paramRast** : str  Raster of parameter values to acumulate, this file is modified by the function.  **fdr** : str  Flow direction raster in TauDEM format.  **accumRast** : str  File location to store accumulated parameter values.  **outNoDataRast** : str (optional)  File location to store parameter no data raster.  **outNoDataAccum** : str (optional)  File location to store accumulated no data raster.  **zeroNoDataRast** : str (optional)  File location to store the no data raster filled with zeros.  **cores** : int (optional)  The number of cores to use for parameter accumulation. Defaults to 1. |
| Returns: | **accumRast** : raster  Raster of accumulated parameter values.  **outNoDataRast** : raster  Raster of no data values.  **outNoDataRast** : raster  Raster of accumulated no data values. |

accumulateParams(*paramRasts*, *fdr*, *outWorkspace*, *cores=1*, *appStr='accum'*)[¶](#tools.accumulateParams)

Batch version of [accumulateParam()](#tools.accumulateParam).

|  |  |
| --- | --- |
| Parameters: | **paramRasts** : list  List of input parameter raster paths to accumulate along the supplied fdr.  **fdr** : str  Path to the flow direction raster.  **outWorkspace** : str  Path to the output directory for accumulation rasters.  **cores** : int (optional)  Number of cores to use. Defaults to 1.  **appStr :str (optional)**  String of text to append to accumulated parameter filenames. Defaults to “accum.” |
| Returns: | **fileList** : list  List of file paths to accumulated parameter rasters. |

adjustFAC(*facWeighttemplate*, *downstreamFACweightFl*, *updateDictFl*, *downstreamFDRFl*, *adjFACFl*, *cores=1*)[¶](#tools.adjustFAC)

Generate an updated flow accumulation grid (FAC) given an update dictionary produced by [createUpdateDict()](#tools.createUpdateDict).

|  |  |
| --- | --- |
| Parameters: | **facWeighttemplate** : str  Path to a FDR or FAC grid used to make the FAC weighting grid.  **downstreamFACweightFl** : str  Path to output the FAC weighting grid.  **updateDictFl** : str  Path to update dictionary used to update the FAC weighting grid.  **downstreamFDRFl** : str  Path to downstream FDR to use when computing the adjusted FAC grid.  **adjFACFl** : str  Path to output the adjusted FAC raster.  **cores** : int (Optional)  Number of cores to use. Defaults to 1. |
| Returns: | **adjFACFl** : raster  Adjusted flow accumulation raster at adjFACFl |

adjustParam(*updatedParam*, *downstreamParamFL*, *updateDictFl*, *adjParamFl*)[¶](#tools.adjustParam)

Generate an updated parameter grid given an update dictionary from [createUpdateDict()](#tools.createUpdateDict).

|  |  |
| --- | --- |
| Parameters: | **updatedParam** : str  Name of the parameter to update.  **downstreamParamFL** : str  Path to downstream parameter grid to update.  **updateDictFl** : str  Path to update dictionary to use.  **adjParamFl** : str  Path to output adjusted parameter file. |
| Returns: | **adjParamFl** : raster  Adjusted parameter raster that can be accumulated prior to FCPG creation. |

applyMult(*inRast*, *mult*, *outRast*)[¶](#tools.applyMult)

Multiply input raster by mult.

|  |  |
| --- | --- |
| Parameters: | **inRast** : str  Path to input raster.  **mult** : str  Path to multiplier raster.  **outRast** : str  Path to output raster. |
| Returns: | **outRast** : raster  Input raster multiplied by the multiplier raster. |

binarizeCat(*val*, *data*, *nodata*, *outWorkspace*, *baseName*, *ext*, *profile*)[¶](#tools.binarizeCat)

Turn a categorical raster (e.g. land cover type) into a set of binary rasters, one for each category in the supplied raster, zero for areas where that class is not present, and -1 for regions of no data in the supplied raster. See also [cat2bin()](#tools.cat2bin).

|  |  |
| --- | --- |
| Parameters: | **data** : np.array  Numpy arrary of raster data to convert to binary.  **val** : int  Raster value to extract binary for from data.  **nodata** : int or float  Raster no data value.  **outWorkspace** : str  Path to folder to save binary output rasters to.  **baseName** : str  Base name for the output rasters.  **ext** : str  File extension for output rasters. |
| Returns: | **catRaster** : str  Filepath to the binary raster created. |

cat2bin(*inCat*, *outWorkspace*)[¶](#tools.cat2bin)

Turn a categorical raster (e.g. land cover type) into a set of binary rasters, one for each category in the supplied raster, zero for areas where that class is not present, and -1 for regions of no data in the supplied raster. Wrapper on [binarizeCat()](#tools.binarizeCat).

|  |  |
| --- | --- |
| Parameters: | **inCat** : str  Input catagorical parameter raster.  **outWorkspace** : str  Workspace to save binary raster output files. |
| Returns: | **fileList** : list  List of filepaths to output files. |

changeNoData(*inRast*, *newNoData*, *updateDict={'compress': 'LZW'*, *'profile': 'GeoTIFF'*, *'tiled': True*, *'sparse\_ok': True*, *'num\_threads': 'ALL\_CPUS'*, *'bigtiff': 'IF\_SAFER'}*)[¶](#tools.changeNoData)

Update raster no data value to a new value.

|  |  |
| --- | --- |
| Parameters: | **inRast** : str  Path to input raster file.  **newNoData** : str  New no data value for the raster.  **updateDict** : dict (optional)  Dictionary of rasterio parameters used to create the updated raster. |
| Returns: | None |

createUpdateDict(*x*, *y*, *upstreamFACmax*, *fromHUC*, *outfl*)[¶](#tools.createUpdateDict)

Create a dictionary for updating downstream FAC and parameter grids using values pulled from the next grid upstream.

|  |  |
| --- | --- |
| Parameters: | **x** : list  Horizontal coordinate(s) for where the update needs to happen in the downstream grid.  **y** : list  Vertical coordinate(s) for where the update needs to happen in the downstream grid.  **upstreamFACmax** : list  Value(s) to insert into the downstream FAC grid.  **fromHUC** : str  The upstream HUC that the values are coming from.  **outfl** : str (path)  Path to where to save the json of this dictionary. The convention is to name this by the downstream HUC. |
| Returns: | **updateDict** : dict  Update dictionary that is also written to outfl. |

d8todinfinity(*inRast*, *outRast*, *updateDict={'dtype': 'float32'*, *'compress': 'LZW'*, *'profile': 'GeoTIFF'*, *'tiled': True*, *'sparse\_ok': True*, *'num\_threads': 'ALL\_CPUS'*, *'nodata': -1*, *'bigtiff': 'IF\_SAFER'}*)[¶](#tools.d8todinfinity)

Convert TauDEM D-8 flow directions to D-Infinity flow directions.

|  |  |
| --- | --- |
| Parameters: | **inRast** : str  Path to a TauDEM D-8 flow direction raster.  **updateDict** : dict (optional)  Dictionary of rasterio parameters used to write out the GeoTiff. |
| Returns: | **outRast** : str  Path to output the TauDEM D-Infinity flow direction raster. |

decayAccum(*ang*, *mult*, *outRast*, *paramRast=None*, *cores=1*)[¶](#tools.decayAccum)

Decay the accumulation of a parameter raster.

|  |  |
| --- | --- |
| Parameters: | **ang** : str  Path to flow angle raster from the TauDEM Dinfinity flow direction tool.  **mult** : str  Path to raster of multiplier values applied to upstream accumulations, 1 corresponds to no decay, 0 corresponds to complete decay.  **outRast** : str  Path to output raster for decayed accumulation raster.  **paramRast** : str (optional)  Raster of parameter values to accumulate. If not supplied area will be accumulated. Defaults to None.  **cores** : int (optional)  Number of cores to use. Defaults to 1. |
| Returns: | **outRast** : raster  Decayed accumulation raster, either area or parameter depending on what is supplied to the function. |

dist2stream(*fdr*, *fac*, *thresh*, *outRast*, *cores=1*)[¶](#tools.dist2stream)

Compute distance to streams.

|  |  |
| --- | --- |
| Parameters: | **fdr** : str  Path to flow direction raster in TauDEM format.  **fac** : str  Path to flow accumulation raster.  **thresh** : int  Accumulation threshold for stream formation in number of grid cells.  **outRast** : str  Path to output the distance raster.  **cores** : int (optional)  The number of cores to use. Defaults to 1. |
| Returns: | **outRast** : raster  Raster with values of d8 flow distance from each cell to the nearest stream. |

findLastFACFD(*facfl*, *fl=None*)[¶](#tools.findLastFACFD)

Find the coordinate of the greatest cell in facfl, return the value from fl at that point.

|  |  |
| --- | --- |
| Parameters: | **facfl** : str  Path to a flow accumulation grid.  **fl** : str (optional)  Path to an accumulated parameter file. Defaults to None. If None, the facfl is queried. |
| Returns: | **x** : float  Horizontal coordinate of the greatest FAC cell.  **y** : float  Vertical coordinate of the greatest FAC cell.  **d** : int or float  Value from the parameter grid queried.  **w** : float  Cell size of the grid. |

Notes

This can be used to find the flow direction of the FAC cell with the greatest accumulation value or the parameter value of the cell with the greatest accumulation value.

findPourPoints(*pourBasins*, *upfacfl*, *upfdrfl*, *plotBasins=False*)[¶](#tools.findPourPoints)

Finds unique pour points between two HUC4s.

|  |  |
| --- | --- |
| Parameters: | **pourBasins** : GeoDataframe  GeoDataframe of the HUC12 basins that flow into the downstream HUC4. Used to clip the upstream FAC grid to identify pour points.  **upfacfl** : str  Path to the upstream flow accumulation grid.  **upfdrfl** : str  Path to the upstream tauDEM flow direction grid.  **plotBasins** : bool (Optional)  Boolean to make plots of upstream HUC12s and identified pour points. Defaults to False. |
| Returns: | **finalPoints** : list  List of tuples containing (x,y,w). These pour points have not been incremented downstream and can be used to query accumulated (but not FCPGed) upstream parameter grids for information to cascade down to |

getFeatures(*gdf*)[¶](#tools.getFeatures)

Helper function to parse features from a GeoPandas GeoDataframe in such a manner that Rasterio can handle them.

|  |  |
| --- | --- |
| Parameters: | **gdf** : GeoDataframe  GeoPandas GeoDataframe with a geometry column. |
| Returns: | **features** : geoJSON  GeoJSON representation of geometry features from the input GeoDataFrame. |

getHUC4(*HUC12*)[¶](#tools.getHUC4)

Helper function to return HUC4 representation from a HUC12 identifier.

|  |  |
| --- | --- |
| Parameters: | **HUC12** : str  Text representation of the 12-digit HUC12 code. |
| Returns: | **HUC4** : str  HUC4 identifier. |

loadJSON(*infl*)[¶](#tools.loadJSON)

Load dictionary stored in a JSON file.

|  |  |
| --- | --- |
| Parameters: | **infl** : str  Path to the JSON to be loaded. |
| Returns: | **dictionary** : dict  Dictionary that was loaded. |

loadRaster(*fl*, *returnMeta=False*, *band=1*)[¶](#tools.loadRaster)

Helper function to load raster data and metadata.

|  |  |
| --- | --- |
| Parameters: | **fl** : str  Path to the raster file to load.  **returnMeta** : bool (Optional)  Return the raster metadata. Defaults to False.  **band** : int (Optional)  Band to read from the raster. Defaults to one. |
| Returns: | **dat** : np.array  Numpy array of the data in the selected raster band.  **meta** : dict  Dictionary of raster metadata. |

makeDecayGrid(*d2strm*, *k*, *outRast*)[¶](#tools.makeDecayGrid)

Create a decay raster where grid cell values are computed as the inverse number of grid cells, \(\frac{1}{n+k\*dx}\), where n is the distance from the d2strm raster and from each grid cell to the nearest stream, k is the constant applied to the stream distance values, and dx is the cell size of the raster.

|  |  |
| --- | --- |
| Parameters: | **d2strm** : str  Path to raster of flow distances from each grid cell to the nearest stream.  **k** : float  Constant applied to decay factor denominator, this has units equal to the horizontal map units in the d2strm raster.  **outRast** : str  Output file path for decay grid. |
| Returns: | **outRast** : raster  Raster file with grid cells values representing weights decaying as they ove further from the stream. |

makeFACweight(*ingrd*, *outWeight*)[¶](#tools.makeFACweight)

Make FAC weighting grid of ones based on the extents of the input grid. No data cells are persisted.

|  |  |
| --- | --- |
| Parameters: | **ingrd** : str  Path to input raster from which to generate the weighting grid from.  **outWeight** : str  Path to the output weighting raster generated. |
| Returns: | **outWeight** : raster  Raster of the same extent and resolution as the input grid, but filled with ones where data exist. No data cells are persisted. |

makePourBasins(*wbd*, *fromHUC4*, *toHUC4*)[¶](#tools.makePourBasins)

Make geodataframe of HUC12 basis flowing from fromHUC4 to toHUC4.

|  |  |
| --- | --- |
| Parameters: | **wbd** : GeoDataframe  HUC12-level geodataframe projected to the same coordinate reference system (CRS) as the flow accumulation (FAC) and flow direction (FDR) grids being used.  **fromHUC4** : str  HUC4 string for the upstream basin.  **toHUC4** : str  HUC string for the downstream basin. |
| Returns: | **pourBasins** : GeoDataframe  HUC12-level geodataframe of units that drain from fromHUC4 to toHUC4. |

make\_fcpg(*accumParam*, *fac*, *outRast*, *noDataRast=None*, *minAccum=None*)[¶](#tools.make_fcpg)

Create a flow-conditioned parameter grid using accumulated parameter and area rasters. See also [make\_fcpgs()](#tools.make_fcpgs).

|  |  |
| --- | --- |
| Parameters: | **accumParam :**  File location of the accumulated parameter data raster.  **fac** : str  File location of the flow accumulation raster.  **outRast** : str  File location of the output flow-conditioned parameter grid.  **noDataRast** : str  File location of the accumulated parameter no data raster.  **minAccum** : float  Value of flow accumulation below which the CPG values will be set to no data |
| Returns: | **outRast** : raster  Flow-conditioned parameter grid file where grid cell values represent the mean upstream value of the paramter. |

make\_fcpgs(*accumParams*, *fac*, *outWorkspace*, *minAccum=None*, *appStr='FCPG'*)[¶](#tools.make_fcpgs)

Batch version of [make\_fcpg()](#tools.make_fcpg).

|  |  |
| --- | --- |
| Parameters: | **accumParams** : list  List of accumulated parameter rasters to create FCPGs from.  **fac** : str  Path to the flow accumulation raster.  **outWorkspace** : str  Path to an oiutput directory for produced FCPGs.  **minAccum** : int (optional)  Minimum accumulation value below which the output FCPG will be turned to no data values. Defaults to None.  **appStr** : str (optional)  String of text to append to filenames of the produced FCPG grids. |
| Returns: | **fileList** : list  List of file paths to the produced FCPGs. |

maskStreams(*inRast*, *streamRast*, *outRast*)[¶](#tools.maskStreams)

Mask areas not on the stream network.

|  |  |
| --- | --- |
| Parameters: | **inRast** : str  Path to the input raster to mask.  **streamRast** : str  Path to the stream raster where all non-stream pixels set to no data.  **outRast** : str  Path to output raster file. |
| Returns: | **outRast** : raster  Raster with non-stream cells set to the no data value from inRast. |

parsebool(*b*)[¶](#tools.parsebool)

Parse a boolean argument from the command line.

|  |  |
| --- | --- |
| Parameters: | **b** : str  String of either True or False. |
| Returns: | **res** : bool  True if b is “True” or False if b is not “True.” |

queryPoint(*x*, *y*, *grd*)[¶](#tools.queryPoint)

Query grid based on a supplied point.

|  |  |
| --- | --- |
| Parameters: | **x** : float or int  Horizontal coordinate in grd projection.  **y** : float or int  Vertical coordinate in grd projection.  **grd** : str  Path to raster to query based on the supplied x and y coordinates. |
| Returns: | **value** : float or int  Value queried from the supplied raster. |

resampleParam(*inParam*, *fdr*, *outParam*, *resampleMethod='bilinear'*, *cores=1*, *forceProj=False*, *forceProj4='"+proj=aea +lat\_1=29.5 +lat\_2=45.5 +lat\_0=23 +lon\_0=-96 +x\_0=0 +y\_0=0 +ellps=GRS80 +datum=NAD83 +units=m +no\_defs"'*)[¶](#tools.resampleParam)

Resample, re-project, and clip the parameter raster based on the resolution, projection, and extent of the of the flow direction raster supplied. See also [resampleParams()](#tools.resampleParams).

|  |  |
| --- | --- |
| Parameters: | **inParam** : str  Path to the input parameter data raster  **fdr** : str  Path to the flow direction raster  **outParam** : str  Path to the output file for the resampled parameter raster.  **resampleMethod** : str (optional)  resampling method, either ‘bilinear’ or ‘nearest neighbor’. Bilinear should generally be used for continuous data sets such as precipitation while nearest neighbor should generally be used for categorical datasets such as land cover type. Defaults to bilinear.  **cores** : int (optional)  The number of cores to use. Defaults to 1.  **forceProj** : bool (optional)  Force the projection of the flow direction raster. This can be useful if the flow direction raster has an unusual projection. Defaults to False.  **forceProj4** : str (optional)  Proj4 string used to force the flow direction raster. This defaults to USGS Albers, but is not used unless the forceProj parameter is set to True. |
| Returns: | **outParam** : raster  Resampled, reprojected, and clipped parameter raster. |

resampleParams(*inParams*, *fdr*, *outWorkspace*, *resampleMethod='bilinear'*, *cores=1*, *appStr='rprj'*)[¶](#tools.resampleParams)

Batch version of [resampleParam()](#tools.resampleParam).

|  |  |
| --- | --- |
| Parameters: | **inParam** : list  List of input parameter raster paths.  **fdr** : str  Path to the flow direction raster.  **outWorkspace :**  Path to the output directory for the resampled rasters.  **resampleMethod** : str (optional)  Resampling method, either bilinear or nearest neighbor. Defaults to bilinear.  **cores :**  Number of cores to use. Defaults to 1.  **appStr** : str (optional)  String of text to append to the input parameter filenames. Defaults to “rprj.” |
| Returns: | **fileList** : list  Paths to resampled, reprojected, and clipped parameter rasters. |

saveJSON(*dictionary*, *outfl*)[¶](#tools.saveJSON)

Save dictionary to JSON file.

|  |  |
| --- | --- |
| Parameters: | **dictionary** : dict  Dictionary to be saved.  **outfl** : str  Path for where to generate the JSON |
| Returns: | None |

tauDrainDir(*inRast*, *outRast*, *updateDict={'compress': 'LZW'*, *'sparse': True*, *'tiled': True*, *'blockysize': 256*, *'blockxsize': 256*, *'driver': 'GTiff'*, *'nodata': 0}*)[¶](#tools.tauDrainDir)

Reclassifies ESRI drainage directions into tauDEM drainage directions.

|  |  |
| --- | --- |
| Parameters: | **inRast** : str  Path to a raster encoded with ESRI flow direction values.  **outRast** : str  Path to output a raster with flow directions encoded for TauDEM.  **updateDict** : dict (optional)  Dictionary of rasterio raster options used to create outRast. Defaults have been supplied, but may not work in all situations and input file formats. |
| Returns: | **outRast** : raster  Reclassified flow direction raster at the path specified above. |

tauFlowAccum(*fdr*, *accumRast*, *cores=1*)[¶](#tools.tauFlowAccum)

Accumulate flow direction grid using TauDEM.

|  |  |
| --- | --- |
| Parameters: | **fdr** : str  Path to a flow direction raster in TauDEM format.  **accumRast** : str  Path to output the flow accumulation raster.  **cores** : int (optional)  Number of cores to use. Defaults to 1. |
| Returns: | **accumRast** : raster  Raster of accumulated parameter values at the path specified above. |

updateDict(*ud*, *upHUC*, *varName*, *val*)[¶](#tools.updateDict)

Update dictionary created using [createUpdateDict()](#tools.createUpdateDict) with a parameter value.

|  |  |
| --- | --- |
| Parameters: | **ud** : str  Path to the update dictionary to add a variable to.  **upHUC** : str  Name of the upstream HUC that the variable cooresponds to.  **varName** : str  Name to use for the variable.  **val** : list, int or float  Value to add to the upstream dictonary. |
| Returns: | **ud** : json  Update dictionary written back out to ud. |

updateRaster(*x*, *y*, *val*, *grd*, *outgrd*)[¶](#tools.updateRaster)

Insert val into grd at location specified by x,y, writes new raster to outgrd.

|  |  |
| --- | --- |
| Parameters: | **x** : list or float  Horizontal coordinate in map units.  **y** : list of float  Vertical coordinate in map units.  **val** : int or float  Value to insert into raster at grd.  **grd** : str  Path to raster to be updated.  **outgrd** : str (path)  Path to write updated raster to. |
| Returns: | **dat** : raster  Writes raster dataset to supplied grdout destination. |

buildNC(*inDir*, *outFile*, *metaDict*, *cl=9*)[¶](#netCDFmaker.buildNC)

Build netCDF file from a stack of geotiffs.

|  |  |
| --- | --- |
| Parameters: | **inDir** : str  Directory with geotiff files to be converted, specified as ‘/dir/here/\*.tif’  **outFile** : str  Output filename with ‘.nc’ included.  **metaDict** : dict  Metadata dictionary used to populate fields in the netCDF. See Notes below for a description of fields to include.  **cl** : int  Compression level, 1-9. A higher value will result in a smaller output file, but will take longer. |
| Returns: | **NetCDF** : file  A NetCDF file at *outFile*. |

Notes

*Metadata Dictionary*

The metadata dictionary is expecting a particular set of keys to specify the metadata fields within the netCDF file being generated. These fields have been chose to make the resulting netCDF file compliant with the Climate and Forecast and the Data Discovery metadata conventions. Fields are described below, many descriptions are the same as <http://cfconventions.org/Data/cf-conventions/cf-conventions-1.7/cf-conventions.html#description-of-file-contents> or <http://wiki.esipfed.org/index.php/Attribute_Convention_for_Data_Discovery_1-3>.

title

A succinct description of what is in the dataset.

institution

Specifies where the original data was produced.

source

The method of production of the original data. If it was model-generated, source should name the model and its version, as specifically as could be useful. If it is observational, source should characterize it (e.g., “surface observation” or “radiosonde”).

id

An identifier for the data set, provided by and unique within its naming authority. The combination of the “naming authority” and the “id” should be globally unique, but the id can be globally unique by itself also. IDs can be URLs, URNs, DOIs, meaningful text strings, a local key, or any other unique string of characters. The id should not include white space characters.

naming\_authority

The organization that provides the initial id (see above) for the dataset. The naming authority should be uniquely specified by this attribute. We recommend using reverse-DNS naming for the naming authority; URIs are also acceptable. Example: ‘edu.ucar.unidata’.

references

Published or web-based references that describe the data or methods used to produce it.

comment

Miscellaneous information about the data or methods used to produce it.

history

Provides an audit trail for modifications to the original data. Well-behaved generic netCDF filters will automatically append their name and the parameters with which they were invoked to the global history attribute of an input netCDF file. We recommend that each line begin with a timestamp indicating the date and time of day that the program was executed.

license

Provide the URL to a standard or specific license, enter “Freely Distributed” or “None”, or describe any restrictions to data access and distribution in free text.

acknowledgement

A place to acknowledge various types of support for the project that produced this data.

metadata\_link

A URL that gives the location of more complete metadata. A persistent URL is recommended for this attribute.

date\_created

The date on which this version of the data was created. (Modification of values implies a new version, hence this would be assigned the date of the most recent values modification.) Metadata changes are not considered when assigning the date\_created. The ISO 8601:2004 extended date format is recommended, as described in the Attribute Content Guidance section.

creator\_type

Specifies type of creator with one of the following: ‘person’, ‘group’, ‘institution’, or ‘position’. If this attribute is not specified, the creator is assumed to be a person.

creator\_email

The email address of the person (or other creator type specified by the creator\_type attribute) principally responsible for creating this data.

creator\_name

The name of the person (or other creator type specified by the creator\_type attribute) principally responsible for creating this data.

creator\_url

The URL of the person (or other creator type specified by the creator\_type attribute) principally responsible for creating this data.

creator\_institution

The institution of the creator; should uniquely identify the creator’s institution. This attribute’s value should be specified even if it matches the value of publisher\_institution, or if creator\_type is institution.

publisher\_type

Specifies type of publisher with one of the following: ‘person’, ‘group’, ‘institution’, or ‘position’. If this attribute is not specified, the publisher is assumed to be a person.

publisher\_name

The name of the person (or other entity specified by the publisher\_type attribute) responsible for publishing the data file or product to users, with its current metadata and format.

publisher\_email

The email address of the person (or other entity specified by the publisher\_type attribute) responsible for publishing the data file or product to users, with its current metadata and format.

publisher\_url

The URL of the person (or other entity specified by the publisher\_type attribute) responsible for publishing the data file or product to users, with its current metadata and format.

publisher\_institution

The institution that presented the data file or equivalent product to users; should uniquely identify the institution. If publisher\_type is institution, this should have the same value as publisher\_name.

var\_name

Variable name given to the dataset. For example minimum temperature would be Tmin.

units

Unit value associated with quantity described by the dataset.

add\_offset

Offset value use with the data, usually 0.0 if no offset is used with the data.

standard\_name

Short name associated with the dataset.

long\_name

More descriptive name associated with the dataset.

grid\_mapping

The way data values are mapped to a grid, usually ‘crs’.

scale\_factor

Numeric value the data are scaled by to save space, usually 1.0 if data are not scaled.

coverage\_content\_type

An ISO 19115-1 code to indicate the source of the data (image, thematicClassification, physicalMeasurement, auxiliaryInformation, qualityInformation, referenceInformation, modelResult, or coordinate).

# Indices and tables[¶](#indices-and-tables)

* [Index](genindex.html)
* [Module Index](py-modindex.html)
* [Search Page](search.html)

© Copyright 2020, Theodore Barnhart, Roy Sando, Seth Siefkin, Peter McCarthy, and Al Rea

Built with [Sphinx](http://sphinx-doc.org/) using a [theme](https://github.com/rtfd/sphinx_rtd_theme) provided by [Read the Docs](https://readthedocs.org).