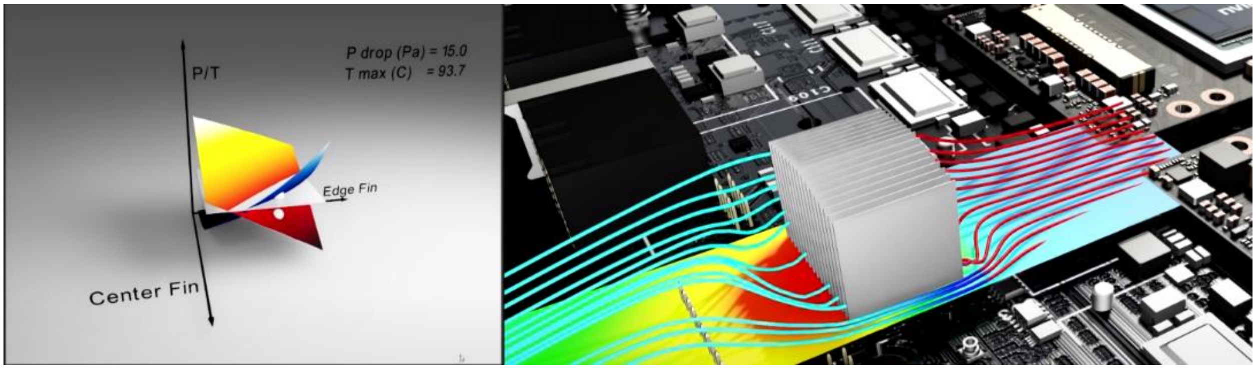
Modulus

A Neural Network Based Partial Differential Equation Solver



Source Code Documentation

Release v21.06 | November 9, 2021

Notice

The information provided in this specification is believed to be accurate and reliable as of the date provided. However, NVIDIA Corporation ("NVIDIA") does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This publication supersedes and replaces all other specifications for the product that may have been previously supplied.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and other changes to this specification, at any time and/or to discontinue any product or service without notice. Customer should obtain the latest relevant specification before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer. NVIDIA hereby expressly objects to applying any customer general terms and conditions with regard to the purchase of the NVIDIA product referenced in this specification.

NVIDIA products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on these specifications will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this specification. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this specification, or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this specification. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA. Reproduction of information in this specification is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

Trademarks

NVIDIA, the NVIDIA logo, CUDA, and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

Modulus

Release 21.06

Nov 03, 2021

CONTENTS:

class modulus.**Node**(*evaluate*, *name='Node'*)

Base class for computation nodes (PDEs, Networks, . . .). Modulus unrolles the tensorflow computational graph for all boundary conditions given to train on. To do this compactly we define our computational graph as a list of *Node's. Using the 'unroll_graph* function we can use these nodes to unroll our computational graph on the desired inputs and outputs. In some sense we are using a node based construction of our computational graph instead of an edge based construction like tensorflow.

Parameters

evaluate [function] A tensorflow function that takes in a dictionary of tensorflow tensors and outputs a dictionary of tensors. The inputs, outputs, and derivatives are automatically infered from this function.

name [str] This is used to name the node.

Methods

derivatives()	**Returns**
from_sympy(eq, out_name)	generates a Modulus Node from a sympy equation
inputs()	**Returns**
outputs()	**Returns**

derivatives()

Returns

derivatives [list of strings] derivative inputs of node.

classmethod from_sympy(*eq*, *out_name*)

generates a Modulus Node from a sympy equation

Parameters

eq [Sympy Symbol/Exp] the equation to convert to a Modulus Node. The inputs to this node consist of all Symbols, Functions, and derivatives of Functions. For example, *f(x,y) + f(x,y).diff(x) + k* will be converted to a node whose input is *f,k*, and derivatives *f__x*.

out_name [str] this will be the name of the output for the node.

Returns

node [Modulus Node]

inputs()

Returns

inputs [list of strings] inputs of node.

outputs()

Returns

outputs [list of strings] outputs of node.

class modulus.**Arch**(***config*)

Base class for all neural networks

Methods

make_node(name, inputs, outputs)	makes neural network node

add_options	
print_configs	
process_config	

make_node(*name*, *inputs*, *outputs*)

makes neural network node

Parameters

name [str] Used as namespace for all tensorflow variables.

inputs [list/tuple of strings or Keys] list of inputs to neural network (e.g. *['x','y']*).

outputs [list/tuple of strings or Keys] list of outputs to neural network (e.g. *['u','v','p']*).

Returns

node [Modulus Node]

class modulus.**Variables**(**args*, ***kwargs*)

Base class for variables. Modulus describes the computational graph as a list of nodes. These *Variables* are what are passed as inputs to nodes (Node.evaluate()). In some sense *Variables* is just a dictionary with extra functionality. Currently this class is used internally by Modulus and not exposed to users.

TODO this class needs to be restructured.

Parameters

input [dictionary] A dictionary of tensorflow tensors.

Methods

clear()	
concatenate(variables_list[, axis])	Concatenates a list of variables together.
copy()	
differentiate(outvar, invar, derivatives[, . . .])	Differentiates two variables with respect to eachother
from_tensor(tensor, keys)	Generate a Varaible from a tensorflow tensor.
fromkeys(keys[, v])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
join_dict(variables_dict_1, variables_dict_2)	combine two dictionaries of variables.
keys()	
l2_relative_error(true_var, pred_var)	Relative error between two variables.

Continued on next page

Table 3 – continued from previous page

`lambdify_np`(invar, outvar, sess)	Creates a function that takes in numpy dictionaries and computes requested outvar.
`loss`(true_var, pred_var, lambda_weighting[, . . .])	Loss between two variables.
`pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
`popitem`(/)	Remove and return a (key, value) pair as a 2-tuple.
`reduce_norm`(variables, name[, order])	Compute norm of all variables together.
`setdefault`(key[, default])	Insert key with a value of default if key is not in the dictionary.
`setdiff`(variables, keys)	set difference of variables
`split`(variables, keys, batch_sizes)	Splits a variable up by batch sizes.
`subset`(variables, keys)	Subset of variables
`tf_summary`(variables[, prefix])	Record all tensorflow tensors in tensorboard with tf.summary
`tf_variables`(keys[, batch_size])	Create Tensorflow placeholders
`to_tensor`(variables)	Concatenates Variables into a single tensorflow tensor.
`update`([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values`()	

callback	

classmethod concatenate(*variables_list*, *axis=0*)
Concatenates a list of variables together.

Parameters

variables_list [list of Variables] Variables to concatenate together.

axis [int] Axis to do concatenation.

Returns

variables [Variables]

copy() → a shallow copy of D

classmethod differentiate(*outvar*, *invar*, *derivatives*, *recursive=False*)
Differentiates two variables with respect to eachother

Parameters

outvar [Variables] Variables to be differentiated.

invar [Variables] Variables to differentiate with respect too.

derivatives [list of strs] A list of all requested derivatives. For example, ‘derivatives=[‘u__x’,’v__y’,u__x__x’]

recursive [bool] if False this will only compute first order derivatives in list. If false this will recursively compute all requested derivatives

Returns

diff_var [Variables] Dictionary of differentiated variables.

classmethod from_tensor(*tensor*, *keys*)
Generate a Varaible from a tensorflow tensor.

Parameters

tensor [tensorflow Tensor] Tensor of shape *[N, . . . , k]*.

keys [list of keys] List of keys whose length totals *k*.

Returns

variables [Variables] variables from the tensor.

classmethod fromkeys(*keys*, *v=None*)
Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*)
Return the value for key if key is in the dictionary, else default.

classmethod join_dict(*variables_dict_1*, *variables_dict_2*)
combine two dictionaries of variables.

static l2_relative_error(*true_var*, *pred_var*)
Relative error between two variables.

loss = sqrt(mean((true_var - pred_var)^2) / var(true_var))

Parameters

true_var [Variables] "true" variables in loss.

pred_var [Variables] "predicted" variables in loss.

Returns

relative_error [Variables] Keys of these varaibles will have prefix *'l2_relative_error_'*.

static lambdify_np(*invar*, *outvar*, *sess*)
Creates a function that takes in numpy dictionaries and computes requested outvar. This handles all Tensorflow placeholders.

Parameters

invar [Variables] Input variable placeholders.

outvar [Variables] Variables to compute.

sess [tf.session] Tensorflow session.

Returns

np_function [Function] This function takes in numpy arrays in the form of invar and returns numpy arrays in the form of outvar.

static loss(*true_var*, *pred_var*, *lambda_weighting*, *order=2*, *prefix='loss'*)
Loss between two variables.

*loss = sum(lambda_weighting * (true_var - pred_var)^order)*

Parameters

true_var [Variables] "true" variables in loss.

pred_var [Variables] "predicted" variables in loss.

lambda_weighting [Variables] weighting for losses. This can be used to performed Monte Carlo integration

order [str, int] order of norm in loss.

prefix [str] prefix str in return keys.

Returns

loss [Variables] Keys of these varaibles will have prefix *'loss_L'+str(order)+'_'*

pop (*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

static reduce_norm (*variables*, *name*, *order=2*)
Compute norm of all variables together.

Parameters

variables [Variables] variables to norm.

name [str] key name of output Variable.

order [str, int] order of norm in loss.

Returns

norm [Variables] Variable with one key, "name".

setdefault (*key*, *default=None*)
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

classmethod setdiff (*variables*, *keys*)
set difference of variables

Parameters

variables [Variables] Variables to take set difference from.

keys [list of strs] list of keys to do subset of variable dict from.

Returns

variables [Variables]

classmethod split (*variables*, *keys*, *batch_sizes*)
Splits a variable up by batch sizes.

Parameters

variables [Variables] Variables to split up.

keys [list of strings] List of keys for return dictionary of Variables.

batch_sizes [list of ints] Each split size.

Returns

var_dict [Variables] Dictionary of variable with with keys, *keys*.

classmethod subset (*variables*, *keys*)
Subset of variables

Parameters

variables [Variables] Variables to take subset from

keys [list of Keys] list of keys to subset variable dict from.

Returns

variables [Variables]

static tf_summary(*variables*, *prefix=''*)

Record all tensorflow tensors in tensorboard with tf.summary

Parameters

variables [Variables] variables to add summary of.

prefix [str] prefix used by tensorboard.

classmethod tf_variables(*keys*, *batch_size=None*)

Create Tensorflow placeholders

Parameters

keys [list of strs, tuples, or Keys] If an element is a string tuple then it is converted into a Key. For each of the Key we will make a tensorflow placeholder.

batch_size [int] batch_size used for each tensor shape.

Returns

variables [Variables]

classmethod to_tensor(*variables*)

Concatenates Variables into a single tensorflow tensor.

Parameters

variables_list [list of Variables] Variables to concatenate together.

axis [int] Axis to do concatenation.

Returns

variables [Variables]

update([*E*], ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

class modulus.**BC**(*invar_fn*, *outvar_fn*, *batch_size*, *lambda_fn=None*)

Class describing boundary conditions.

This allows for generating or sampling mini-batches to train on.

Parameters

invar_fn [function] A Python function whose input is an int *batch_size* and whose output is a dictionary of numpy arrays for the positional points on the boundary condition.

outvar_fn [function] A Python function whose input is the output of dictionary of *invar_fn*. The output is another dictionary of numpy arrays corresponding to the boundary conditions.

batch_size [int] Batch size used for training.

lambda_fn [None, function] Function whose input is the output dictionary of *invar_fn* and *outvar_fn*. The output is a dictionary of numpy arrays corresponding to the weighting of the losses for each point. In other words, these weighting outputs will corrispond to *lambda_weighting* in,

*loss = sum(lambda_weighting * (true_var - pred_var)^2)*

The default None causes *lambda_fn* to output numpy arrays with value *1/batch_size*. This causes the above loss to become a standard *reduce_mean*.

Methods

`from_numpy`(invar_numpy, outvar_numpy[, …])	Make boundary condition from numpy arrays
`sample`()	Samples mini batch on boundary condition

invar_names	
lambda_names	
outvar_names	

classmethod from_numpy(*invar_numpy*, *outvar_numpy*, *batch_size=None*, *lambda_numpy=None*)

Make boundary condition from numpy arrays

Parameters

invar_numpy [dictionary of numpy arrays] dictionary of positional points on the boundary condition.

outvar_numpy [function] dictionary of boundary condition values.

batch_size [int] Batch size used for training. If None then the entire array size is used as the batch size.

lambda_numpy [None, dictionary of numpy arrays] dictionary of numpy arrays corresponding to the spacial weighting of the losses. In other words, these values will correspond to *lambda_weighting* in,

*loss = sum(lambda_weighting * (true_var - pred_var)^2)*

The default None causes *lambda_numpy* to equal numpy arrays full of value *1/batch_size*. This causes the above loss to become a standard *reduce_mean*.

sample()

Samples mini batch on boundary condition

Returns

variable [dictionary of numpy arrays] dictionary of numpy array that where sampled from the invar_fn, outvar_fn, and lambda_fn.

class modulus.**Monitor**(*invar_numpy*, *metrics*)

Class handling training monitors. This lets you track things like equation residuals.

Parameters

invar_numpy [dictionary of numpy arrays] A dictionary of positional points to run inference on.

metrics [dictionary of tensorflow functions] A dictionary where each value is a tensorflow function whose input is a *Variables* and whose output is a scalar value.

Methods

invar_names	
nodes	
outvar_names	
sample	

class modulus.**Validation**(*invar_fn*, *outvar_fn*, *batch_size=None*)

Class handling data to perform validation on.

Parameters

invar_numpy [dictionary of numpy arrays] A dictionary of positional points to run inference on.

outvar_names [list of str] A list of variable names that are requested to run inference on.

Methods

from_numpy(invar_numpy, outvar_numpy[, ...])	Make validation from numpy arrays
sample()	Samples mini batch on boundary condition

invar_names	
lambda_names	
outvar_names	

classmethod from_numpy(*invar_numpy*, *outvar_numpy*, *batch_size=None*)

Make validation from numpy arrays

Parameters

invar_numpy [dictionary of numpy arrays] dictionary of positional points on the boundary condition.

outvar_numpy [function] dictionary of boundary condition values.

batch_size [int] Batch size used for validating. If None then the entire array size is used as the batch size.

sample()

Samples mini batch on boundary condition

Returns

variable [dictionary of numpy arrays] dictionary of numpy array that where sampled from the invar_fn, outvar_fn, and lambda_fn.

class modulus.**Inference**(*invar_numpy*, *outvar_names*)

Class handling data to run inference on.

Parameters

invar_numpy [dictionary of numpy arrays] A dictionary of positional points to run inference on.

outvar_names [list of str] A list of variable names that are requested to run inference on.

Methods

invar_names	
outvar_names	
sample	

class modulus.**TrainDomain**(*nr_threads=8*, *samples_in_queue=100*, ***config*)

Train DataSet referred to as TrainDomain. This DataSet consists of a set of boundary conditions and is what the neural network will be optimized against.

Methods

`add`(data, name)	Adds data to dataset
`sample`()	Samples Dataset

add_options	
make_inputs	
plot	
print_configs	
process_config	
start_queue	

class modulus.**MonitorDomain**(*nr_threads=1*, *samples_in_queue=2*, ***config*)

Monitor DataSet referred to as MonitorDomain. This DataSet consists of a set of Monitor data and is used to monitor the solution from the solver.

Methods

`add`(data, name)	Adds data to dataset
`sample`()	Samples Dataset

add_options	
make_inputs	
plot	
print_configs	
process_config	
start_queue	

class modulus.**ValidationDomain**(*nr_threads=1*, *samples_in_queue=2*, ***config*)

Validation DataSet referred to as ValidationDomain. This DataSet consists of a set of Validation data and is used to easily validate the solver results to known solutions form other solvers or analytical solutions.

Methods

add(data, name)	Adds data to dataset
sample()	Samples Dataset

add_options	
make_inputs	
plot	
print_configs	
process_config	
start_queue	

class modulus.**InferenceDomain**(*nr_threads=1*, *samples_in_queue=2*, ***config*)

Inference DataSet referred to as InferenceDomain. This DataSet consists of a set of Inference data and is used to easily evaluate the solver on desired quantities.

Methods

add(data, name)	Adds data to dataset
sample()	Samples Dataset

add_options	
make_inputs	
plot	
print_configs	
process_config	
start_queue	

class modulus.**DataSet**(*nr_threads*, *samples_in_queue*)

DataSet classes for threading data

Parameters

nr_threads [int] Number of threads to que data with.

samples_in_queue [int] Max number of samples in que.

Methods

add(data, name)	Adds data to dataset
sample()	Samples Dataset

add_options	
make_inputs	
plot	
print_configs	
process_config	
start_queue	

add(*data*, *name*)

Adds data to dataset

Parameters

data [member of Data Class] data to add to DataSet.

name [str] The data will use this indexed with this name

sample()

Samples Dataset

Returns

data_samples [dictionary of Variables] For every Data class in DataSet this returns a sample from that Data. The keys of the dictionary are the names given for each Data.

class modulus.**Solver**(***config*)

Trains and Evaluates solver.

Attributes

train_domain [TrainDomain] Defines the boundary conditions to train on.

inference_domain [InferenceDomain] Used to define a collection of points to run inference on.

val_domain [ValidationDomain] Used to compare Modulus solutions to validation data.

monitor_domain [MonitorDomain] Used to make monitors to check convergence of simulation.

arch [Arch] Standard fully connected network

optimizer [Optimizer] Adam optimizer

lr [LR] Exponentialy decayed learning rate

Methods

arch	alias of `modulus.architecture.fully_connected.FullyConnectedArch`
lr	alias of `modulus.learning_rate.ExponentialDecayLR`
optimizer	alias of `modulus.optimizer.AdamOptimizer`

add_global_step	
add_options	
base_dir	
broadcast_initialization	
custom_loss	
custom_update_op	
eval	
initialize_optimizer	
load_iteration_step	
load_network	
make_domain_dirs	
plot_data	
print_configs	
process_config	
record_inference	
record_monitor	
record_train	
record_validation	
run_initialization	
save_checkpoint	
save_iteration_step	
solve	
start_session	
stream	
train_domain_loss	
unroll_train_domain	
update_network_dir	
val_domain_error	

arch

alias of `modulus.architecture.fully_connected.FullyConnectedArch`

config = None

initialize domains for i, train_domain in enumerate(self.seq_train_domain):

if type(train_domain) == type: self.seq_train_domain[i] = train_domain(**config)

if type(self.inference_domain) == type: self.inference_domain = self.inference_domain(**config)

if type(self.val_domain) == type: self.val_domain = self.val_domain(**config)

if type(self.monitor_domain) == type: self.monitor_domain = self.monitor_domain(**config)

self.monitor_outvar_store = { }

lr

alias of `modulus.learning_rate.ExponentialDecayLR`

optimizer

alias of `modulus.optimizer.AdamOptimizer`

class `modulus.PDES.`**`AdvectionDiffusion`**(*T='T'*, *D='D'*, *Q=0*, *rho='rho'*, *dim=3*, *time=False*)

Advection diffusion equation

Parameters

T [str] The dependent variable.

D [float, Sympy Symbol/Expr, str] Diffusivity. If *D* is a str then it is converted to Sympy Function of form 'D(x,y,z,t)'. If 'D' is a Sympy Symbol or Expression then this is substituted into the equation.

Q [float, Sympy Symbol/Expr, str] The source term. If *Q* is a str then it is converted to Sympy Function of form 'Q(x,y,z,t)'. If 'Q' is a Sympy Symbol or Expression then this is substituted into the equation. Default is 0.

rho [float, Sympy Symbol/Expr, str] The density. If *rho* is a str then it is converted to Sympy Function of form 'rho(x,y,z,t)'. If 'rho' is a Sympy Symbol or Expression then this is substituted into the equation to allow for compressible Navier Stokes.

dim [int] Dimension of the diffusion equation (1, 2, or 3). Default is 3.

time [bool] If time-dependent equations or not. Default is False.

Examples

```
>>> ad = AdvectionDiffusion(D=0.1, rho=1.)
>>> ad.pprint(preview=False)
  advection_diffusion: u*T__x + v*T__y + w*T__z - 0.1*T__x__x - 0.1*T__y__y - 0.
↪1*T__z__z
>>> ad = AdvectionDiffusion(D='D', rho=1, dim=2, time=True)
>>> ad.pprint(preview=False)
  advection_diffusion: -D*T__x__x - D*T__y__y + u*T__x + v*T__y - D__x*T__x - D__
↪y*T__y + T__t
```

Methods

`make_node`([stop_gradients])	Make a list of nodes from PDE.
`pprint`([preview])	Print partial differential equation

subs	

name = 'AdvectionDiffusion'

class modulus.PDES.**IntegralAdvection**(*T*, *D='D'*, *dim=2*)

Imposes the conservation of heat flux in an integral form.

Parameters

T [str] The dependent variable.

D [float, Sympy Symbol/Expr, str] Diffusivity. If *D* is a str then it is converted to Sympy Function of form 'D(x,y,z,t)'. If 'D' is a Sympy Symbol or Expression then this is substituted into the equation.

dim [int] Dimension of the equation (1, 2, or 3). Default is 2.

Methods

make_node	

make_node()

class modulus.PDES.**Diffusion**(*T='T'*, *D='D'*, *Q=0*, *dim=3*, *time=True*)

Diffusion equation

Parameters

T [str] The dependent variable.

D [float, Sympy Symbol/Expr, str] Diffusivity. If *D* is a str then it is converted to Sympy Function of form 'D(x,y,z,t)'. If 'D' is a Sympy Symbol or Expression then this is substituted into the equation.

Q [float, Sympy Symbol/Expr, str] The source term. If *Q* is a str then it is converted to Sympy Function of form 'Q(x,y,z,t)'. If 'Q' is a Sympy Symbol or Expression then this is substituted into the equation. Default is 0.

dim [int] Dimension of the diffusion equation (1, 2, or 3). Default is 3.

time [bool] If time-dependent equations or not. Default is True.

Examples

```
>>> diff = Diffusion(D=0.1, Q=1, dim=2)
>>> diff.pprint(preview=False)
  diffusion_T: T__t - 0.1*T__x__x - 0.1*T__y__y - 1
>>> diff = Diffusion(T='u', D='D', Q='Q', dim=3, time=False)
>>> diff.pprint(preview=False)
  diffusion_u: -D*u__x__x - D*u__y__y - D*u__z__z - Q - D__x*u__x - D__y*u__y - D_
↪_z*u__z
```

Methods

make_node([stop_gradients])	Make a list of nodes from PDE.
pprint([preview])	Print partial differential equation

subs	

name = 'Diffusion'

class modulus.PDES.**DiffusionInterface**(*T_1*, *T_2*, *D_1*, *D_2*, *dim=3*, *time=True*)

Matches the boundary conditions at an interface

Parameters

T_1, T_2 [str] Dependent variables to match the boundary conditions at the interface.

D_1, D_2 [float] Diffusivity at the interface.

dim [int] Dimension of the equations (1, 2, or 3). Default is 3.

time [bool] If time-dependent equations or not. Default is True.

Methods

`make_node`([stop_gradients])	Make a list of nodes from PDE.
`pprint`([preview])	Print partial differential equation

subs	

name = 'DiffusionInterface'

class modulus.PDES.**IntegralDiffusion**(*T*, *dim=2*)

Implementation of the integral diffusion

Parameters

T [str] The dependent variable.

dim [int] Dimension of the equations (2 or 3). Default is 2.

Methods

make_node	

make_node()

class modulus.PDES.**MaxwellFreqReal**(*ux='ux'*, *uy='uy'*, *uz='uz'*, *k=1.0*)

Frequency domain Maxwell's equation

Parameters

ux [str] Ex

uy [str] Ey

uz [str] Ez

k [float, Sympy Symbol/Expr, str] Wave number. If *k* is a str then it is converted to Sympy Function of form 'k(x,y,z,t)'. If 'k' is a Sympy Symbol or Expression then this is substituted into the equation.

Methods

`make_node`([stop_gradients])	Make a list of nodes from PDE.
`pprint`([preview])	Print partial differential equation

subs	

name = 'MaxwellFreqReal'

class modulus.PDES.**SommerfeldBC**(*ux='ux'*, *uy='uy'*, *uz='uz'*)

Frequency domain ABC, Sommerfeld radiation condition Only for real part Equation: 'n x _curl(E) = 0'

Parameters

ux [str] Ex

uy [str] Ey

uz [str] Ez

Methods

make_node([stop_gradients])	Make a list of nodes from PDE.
pprint([preview])	Print partial differential equation

subs	

name = 'SommerfeldBC'

class modulus.PDES.**PEC**(*ux='ux'*, *uy='uy'*, *uz='uz'*, *dim=3*)

Perfect Electric Conduct BC for

Parameters

ux [str] Ex

uy [str] Ey

uz [str] Ez

dim [int] Dimension of the equations (2, or 3). Default is 3.

Methods

make_node([stop_gradients])	Make a list of nodes from PDE.
pprint([preview])	Print partial differential equation

subs	

name = 'PEC_3D'

class modulus.PDES.**EnergyFluid**(*cp='cp'*, *kappa='kappa'*, *rho='rho'*, *nu='nu'*, *visc_heating=False*, *dim=3*, *time=False*)

Energy equation Supports compressible flow. For Ideal gases only (uses the assumption that beta*T = 1). No heat/energy source added.

Parameters

cp [str] The specific heat.

kappa [str] The conductivity.

rho [Sympy Symbol/Expr, str] The density. If *rho* is a str then it is converted to Sympy Function of form 'rho(x,y,z,t)'. If 'rho' is a Sympy Symbol or Expression then this is substituted into the equation.

nu [Sympy Symbol/Expr, str] The kinematic viscosity. If *nu* is a str then it is converted to Sympy Function of form 'nu(x,y,z,t)'. If 'nu' is a Sympy Symbol or Expression then this is substituted into the equation.

visc_heating [bool] If viscous heating is applied or not. Default is False.

dim [int] Dimension of the energy equation (2 or 3). Default is 3.

time [bool] If time-dependent equations or not. Default is False.

Examples

```
>>> ene = EnergyFluid(nu=0.1, rho='rho', cp=2.0, kappa=5, dim=2,  time=False,␣
↪visc_heating=False)
>>> ene.pprint(preview=False)
  [('temperauture_fluid', '2.0 \left(u{\left(x,y \right)}
  \frac{\partial}{\partial x} T{\left(x,y \right)}
  + v{\left(x,y \right)} \frac{\partial}{\partial y}
  T{\left(x,y \right)}\right) \rho{\left(x,y \right)}
  - u{\left(x,y \right)} \frac{\partial}{\partial x} p{\left(x,y \right)}
  - v{\left(x,y \right)} \frac{\partial}{\partial y} p{\left(x,y \right)}
  - 5 \frac{\partial^{2}}{\partial x^{2}} T{\left(x,y \right)}
  - 5 \frac{\partial^{2}}{\partial y^{2}} T{\left(x,y \right)}')]
```

Methods

`make_node`([stop_gradients])	Make a list of nodes from PDE.
`pprint`([preview])	Print partial differential equation

subs	

class `modulus.PDES.`**`NavierStokes`**(*nu*, *rho=1*, *dim=3*, *time=True*)

Compressible Navier Stokes equations

Parameters

nu [float, Sympy Symbol/Expr, str] The kinematic viscosity. If *nu* is a str then it is converted to Sympy Function of form *nu(x,y,z,t)*. If *nu* is a Sympy Symbol or Expression then this is substituted into the equation. This allows for variable viscosity.

rho [float, Sympy Symbol/Expr, str] The density of the fluid. If *rho* is a str then it is converted to Sympy Function of form 'rho(x,y,z,t)'. If 'rho' is a Sympy Symbol or Expression then this is substituted into the equation to allow for compressible Navier Stokes. Default is 1.

dim [int] Dimension of the Navier Stokes (2 or 3). Default is 3.

time [bool] If time-dependent equations or not. Default is True.

Examples

```
>>> ns = NavierStokes(nu=0.01, rho=1, dim=2)
>>> ns.pprint(preview=False)
  continuity: u__x + v__y
  momentum_x: u*u__x + v*u__y + p__x + u__t - 0.01*u__x__x - 0.01*u__y__y
  momentum_y: u*v__x + v*v__y + p__y + v__t - 0.01*v__x__x - 0.01*v__y__y
>>> ns = NavierStokes(nu='nu', rho=1, dim=2, time=False)
>>> ns.pprint(preview=False)
  continuity: u__x + v__y
```

(continues on next page)

(continued from previous page)

```
momentum_x: -nu*u__x__x - nu*u__y__y + u*u__x + v*u__y - nu__x*u__x - nu__y*u__
↪y + p__x
momentum_y: -nu*v__x__x - nu*v__y__y + u*v__x + v*v__y - nu__x*v__x - nu__y*v__
↪y + p__y
```

Methods

make_node([stop_gradients])	Make a list of nodes from PDE.
pprint([preview])	Print partial differential equation

subs	

name = 'NavierStokes'

class modulus.PDES.**IntegralContinuity**(*dim=2*)

Implementation of the integral continuity condition

Parameters

dim [int] Dimension of the equations (2 or 3). Default is 2.

Methods

make_node	

make_node()

class modulus.PDES.**GradNormal**(*T, dim=3, time=True*)

Implementation of the gradient boundary condition

Parameters

T [str] The dependent variable.

dim [int] Dimension of the equations (1, 2, or 3). Default is 3.

time [bool] If time-dependent equations or not. Default is True.

Examples

```
>>> gn = ns = GradNormal(T='T')
>>> gn.pprint(preview=False)
  normal_gradient_T: normal_x*T__x + normal_y*T__y + normal_z*T__z
```

Methods

make_node([stop_gradients])	Make a list of nodes from PDE.
pprint([preview])	Print partial differential equation

subs	

name = 'GradNormal'

class modulus.PDES.**ScreenedPoissonDistance**(*distance='normal_distance'*, *tau=0.1*, *dim=3*)

Screened Poisson Distance

Parameters

distance [str] A user-defined variable for distance. Default is "normal_distance".

tau [float] A small, positive parameter. Default is 0.1.

dim [int] Dimension of the Screened Poisson Distance (1, 2, or 3). Default is 3.

Methods

make_node([stop_gradients])	Make a list of nodes from PDE.
pprint([preview])	Print partial differential equation

subs	

name = 'ScreenedPoissonDistance'

class modulus.PDES.**KEpsilon**(*nu*, *rho=1*, *dim=3*, *time=True*)

k-epsilon Turbulence model

Parameters

nu [float] The kinematic viscosity of the fluid.

rho [float, Sympy Symbol/Expr, str] The density. If *rho* is a str then it is converted to Sympy Function of form 'rho(x,y,z,t)'. If 'rho' is a Sympy Symbol or Expression then this is substituted into the equation.

dim [int] Dimension of the k-epsilon turbulence model (2 or 3). Default is 3.

time [bool] If time-dependent equations or not. Default is True.

Methods

make_node([stop_gradients])	Make a list of nodes from PDE.
pprint([preview])	Print partial differential equation

subs	

name = 'ZeroEquation'

class modulus.PDES.**ZeroEquation**(*nu*, *max_distance*, *rho=1*, *dim=3*, *time=True*)

Zero Equation Turbulence model

Parameters

nu [float] The kinematic viscosity of the fluid.

max_distance [float] The maximum wall distance in the flow field.

rho [float, Sympy Symbol/Expr, str] The density. If *rho* is a str then it is converted to Sympy Function of form 'rho(x,y,z,t)'. If 'rho' is a Sympy Symbol or Expression then this is substituted into the equation. Default is 1.

dim [int] Dimension of the Zero Equation Turbulence model (2 or 3). Default is 3.

time [bool] If time-dependent equations or not. Default is True.

Methods

make_node([stop_gradients])	Make a list of nodes from PDE.
pprint([preview])	Print partial differential equation

subs	

name = 'ZeroEquation'

class modulus.PDES.**WaveEquation**(*u='u'*, *c='c'*, *dim=3*, *time=True*)

Wave equation

Parameters

u [str] The dependent variable.

c [float, Sympy Symbol/Expr, str] Wave speed coefficient. If *c* is a str then it is converted to Sympy Function of form 'c(x,y,z,t)'. If 'c' is a Sympy Symbol or Expression then this is substituted into the equation.

dim [int] Dimension of the wave equation (1, 2, or 3). Default is 2.

time [bool] If time-dependent equations or not. Default is True.

Examples

```
>>> we = WaveEquation(c=0.8, dim=3)
>>> we.pprint(preview=False)
  wave_equation: u__t__t - 0.64*u__x__x - 0.64*u__y__y - 0.64*u__z__z
>>> we = WaveEquation(c='c', dim=2, time=False)
>>> we.pprint(preview=False)
  wave_equation: -c**2*u__x__x - c**2*u__y__y - 2*c*c__x*u__x - 2*c*c__y*u__y
```

Methods

make_node([stop_gradients])	Make a list of nodes from PDE.
pprint([preview])	Print partial differential equation

subs	

name = 'WaveEquation'

class modulus.architecture.**FullyConnectedArch**(***config*)

Standard fully connected network

Methods

make_node(name, inputs, outputs)	makes neural network node

add_options	
print_configs	
process_config	

class modulus.architecture.**FourierNetArch**(***config*)

Fourier Net

Methods

make_node(name, inputs, outputs)	makes neural network node
set_frequencies(frequencies[, . . .])	Set frequencies four first layer of Fourier neural network.

add_options	
print_configs	
process_config	

set_frequencies(*frequencies*, *frequencies_params=None*)

Set frequencies four first layer of Fourier neural network. This can be used to set arbitrary frequency ranges as well as default frequency ranges

Parameters

frequencies [numpy array or tuple with name and list] If frequencies is a numpy array then this is used directly as the frequencies. In this case the array shape must be *[nr_freq, dims]* where *nr_freq* is the number of frequencies you are using and *dims* is the dimension *[1, 2, 3, 4]*. If frequencies is given as a tuple then it defines one of the default frequency ranges. The first element of the tuple is the frequency type and the second element is a list frequencies to use. The possibilities for frequency types are, *full* - All terms in frequency space *axis* - Just the axis terms. *diagonal* - spectrum on the diagonal of the frequency

space.

You can also speficy multiple types of frequencies. In particular, you can use *axis,diagonal* if you want to use both the axis and diagonal of the frequency space.

frequencies_params [numpy array or tuple with name and list] Same types and functionality as *frequencies* however this is used for params that are not *x*, *y*, *z*, or *t*. If None is given then same frequencies as *frequencies*.

class modulus.architecture.**DGMArch**(***config*)

A variation of the fully connected network. Reference: Sirignano, J. and Spiliopoulos, K., 2018. DGM: A deep learning algorithm for solving partial differential equations. Journal of computational physics, 375, pp.1339-1364.

Methods

`make_node`(name, inputs, outputs)	makes neural network node

add_options	
print_configs	
process_config	

class `modulus.architecture.`**`ModifiedFourierNetArch`**(***config*)

A modified Fourier Network which enables multiplicative interactions betweeen the Fourier features and hidden layers. References: (1) Tancik, M., Srinivasan, P.P., Mildenhall, B., Fridovich-Keil, S., Raghavan, N., Singhal, U., Ramamoorthi, R., Barron, J.T. and Ng, R., 2020. Fourier features let networks learn high frequency functions in low dimensional domains. arXiv preprint arXiv:2006.10739. (2) Wang, S., Teng, Y. and Perdikaris, P., 2020. Understanding and mitigating gradient pathologies in physics-informed neural networks. arXiv preprint arXiv:2001.04536.

Methods

`make_node`(name, inputs, outputs)	makes neural network node
`set_frequencies`(frequencies[, ...])	Set frequencies four first layer of Fourier neural network.

add_options	
print_configs	
process_config	

`set_frequencies`(*frequencies*, *frequencies_params=None*)

Set frequencies four first layer of Fourier neural network. This can be used to set arbitrary frequency ranges as well as default frequency ranges

Parameters

frequencies [numpy array or tuple with name and list] If frequencies is a numpy array then this is used directly as the frequencies. In this case the array shape must be *[nr_freq, dims]* where *nr_freq* is the number of frequencies you are using and *dims* is the dimension *[1, 2, 3, 4]*. If frequencies is given as a tuple then it defines one of the default frequency ranges. The first element of the tuple is the frequency type and the second element is a list frequencies to use. The possibilities for frequency types are, *full* - All terms in frequency space *axis* - Just the axis terms. *diagonal* - spectrum on the diagonal of the frequency

space.

You can also speficy multiple types of frequencies. In particular, you can use *axis,diagonal* if you want to use both the axis and diagonal of the frequency space.

frequencies_params [numpy array or tuple with name and list] Same types and functionality as *frequencies* however this is used for params that are not *x*, *y*, *z*, or *t*. If None is given then same frequencies as *frequencies*.

class `modulus.architecture.`**`RadialBasisArch`**(***config*)

Methods

`make_node`(name, inputs, outputs)	makes neural network node

add_options	
print_configs	
process_config	
set_bounds	

class modulus.architecture.**SirenArch**(***config*)

Siren fully connected network Reference: Sitzmann, Vincent, et al. Implicit Neural Representations with Periodic Activation Functions. arXiv preprint arXiv:2006.09661 (2020).

Methods

`make_node`(name, inputs, outputs)	makes neural network node

add_options	
print_configs	
process_config	

Defines base class for all geometries

class modulus.sympy_utils.geometry.**Geometry**(*curves*, *sdf*)

Constructive Geometry Module that allows sampling on surface and interior

Parameters

curve [list of Curves] These curves with define the surface or perimiter of geometry.

sdf [SymPy Exprs] SymPy expresion of signed distance function.

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, . . . [, . . .])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, . . .])	Samples the surface or perimiter of the geometry.
sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry

Continued on next page

Table 31 – continued from previous page

translate(xyz)	translate geometry

copy	

boundary_bc(*outvar_sympy*, *batch_size_per_area*, *criteria=True*, *lambda_sympy=None*, *param_ranges={}*, *fixed_var=True*, *batch_per_epoch=1000*, *quasirandom=False*)

Create a Boundary Condition of surface or perimiter of the geometry.

Parameters

outvar_sympy [dictionary of SymPy Symbols/Expr, floats or ints] This is used to describe the boundary condition. For example, *outvar_sympy={'u': 0}* would specify *'u'* to be zero on this boundary.

batch_size_per_area [int] Batch size per unit area of perimeter/area.

criteria [None, SymPy boolean exprs] Only sample points that satisfy this criteria.

lambda_sympy [dictionary of SymPy Symbols/Expr, floats or ints] This is used to describe the weighting of the boundary condition. The weighting will be multiplied by the default weighting of *area / batch_size*. For example, *lambda_sympy={'lambda_u': 0.1}* would result in loss *l_u = sum((0.1*area/batch_size)*(true_u-pred_u)**2*, where *area* is the surface area of the surface or perimiter, *true_u* is specified in *outvar_sympy* and *pred_u* is the predicted *u* from the network.

param_ranges: dict with of SymPy Symbols and their ranges If the geometry is parameterized then you can provide ranges for the parameters with this.

fixed_var [bool] If False then the sample points will be generated on the fly. If it is True then points will be generated ahead of time once.

batch_per_epoch [int] If *fixed_var* is True then the number of points generated ahead of time will be *perimeter * batch_size_per_area * batch_per_epoch*. In other words the batch size times *batch_per_epoch*.

quasirandom [bool] If true then sample the points using the Halton sequences. Default is False.

Returns

BC [BC as defined in *data.py*.]

dims()

Returns

dims [list of strings] output can be ['x'], ['x','y'], or ['x','y','z']

interior_bc(*outvar_sympy*, *bounds*, *batch_size_per_area*, *criteria=True*, *lambda_sympy=None*, *param_ranges={}*, *fixed_var=True*, *batch_per_epoch=1000*, *quasirandom=False*)

Create a Boundary Condition of interior of the geometry (where SDF is positive).

Parameters

outvar_sympy [dictionary of SymPy Symbols/Expr, floats or ints] This is used to describe the boundary condition. For example, *outvar_sympy={'u': 0}* would specify *'u'* to be zero on this boundary.

bounds [dict] bounds to sample points from. For example, *bounds = {'x': (0, 1), 'y': (0, 1)}*.

batch_size_per_area [int] Batch size per unit area of volume.

criteria [None, SymPy boolean exprs] Only sample points that satisfy this criteria.

lambda_sympy [dictionary of SymPy Symbols/Expr, floats or ints] This is used to describe the weighting of the boundary condition. The weighting will be multiplied by the default weighting of *area / batch_size*. For example, *lambda_sympy={'lambda_u': 0.1}* would result in loss *l_u = sum((0.1*area/batch_size)*(true_u-pred_u)**2*, where *area* is the surface area of the surface or perimiter, *true_u* is specified in *outvar_sympy* and *pred_u* is the predicted *u* from the network.

param_ranges: dict with of SymPy Symbols and their ranges If the geometry is parameterized then you can provide ranges for the parameters with this.

fixed_var [bool] If False then the sample points will be generated on the fly. If it is True then points will be generated ahead of time once.

batch_per_epoch [int] If *fixed_var* is True then the number of points generated ahead of time will be *volume * batch_size_per_area * batch_per_epoch*. In other words the batch size times *batch_per_epoch*.

quasirandom [bool] If true then sample the points using the Halton sequences. Default is False.

Returns

BC [BC as defined in *data.py*.]

`perimeter`(*param_ranges={}*)
Perimeter or surface area of geometry.

Parameters

param_ranges: dict with of SymPy Symbols and their ranges If the geometry is parameterized then you can provide ranges for the parameters with this.

Returns

perimeter [float] total perimeter or surface area of geometry.

`repeat`(*spacing*, *repeat_lower*, *repeat_higher*, *center=None*)
finite repetition of geometry

Parameters

spacing [float, int or SymPy Symbol/Exp] spacing between each repetition.

repeat_lower [tuple of ints] How many repetitions going in negative direction.

repeat_higher [tuple of ints] How many repetitions going in positive direction.

center [None, list of floats] Do repetition with this center.

`rotate`(*angle*, *axis='z'*, *center=None*)
rotates geometry

Parameters

angle [float, int, SymPy Symbol/Exp] Angle to rotate geometry.

axis [str] Rotate around this axis.

center [None, list of floats] Do rotation around this center

`sample_boundary`(*nr_points_per_area*, *criteria=None*, *param_ranges={}*, *discard_criteria=True*, *quasirandom=False*)
Samples the surface or perimiter of the geometry.

Parameters

nr_points_per_area [int] number of points per unit area to sample.

criteria [None, SymPy boolean exprs] Only sample points that satisfy this criteria.

param_ranges: dict with of SymPy Symbols and their ranges If the geometry is parameterized then you can provide ranges for the parameters with this.

discard_criteria [bool] If true then return points that don't satify criteria or *sdf(x,y,z)=0*. This is used to keep constant return

batch size.

quasirandom [bool] If true then sample the points using the Halton sequences. Default is False.

`sample_interior`(*nr_points_per_volume*, *bounds*, *criteria=None*, *param_ranges={}*, *discard_criteria=True*, *quasirandom=False*)

Samples the interior of the geometry.

Parameters

nr_points_per_volume [int] number of points per unit volume to sample.

bounds [dict] bounds to sample points from. For example, *bounds = {'x': (0, 1), 'y': (0, 1)}*.

criteria [None, SymPy boolean exprs] Only sample points that satisfy this criteria.

param_ranges: dict with of SymPy Symbols and their ranges If the geometry is parameterized then you can provide ranges for the parameters with this.

discard_criteria [bool] If true then return points that don't satify criteria or *sdf(x,y,z)=0*. This is used to keep constant return

batch size.

quasirandom [bool] If true then sample the points using the Halton sequences. Default is False.

`scale`(*x*, *center=None*)

scale geometry

Parameters

x [float, int, SymPy Symbol/Exp] Scale factor.

center [None, list of floats] Do scaling with around this center.

`translate`(*xyz*)

translate geometry

Parameters

xyz [tuple of float, int or SymPy Symbol/Exp] translate geometry by these values.

class `modulus.sympy_utils.geometry_1d.`**`Line1D`**(*point_1*, *point_2*)

1D Line along x-axis

Parameters

point_1 [int or float] lower bound point of line

point_2 [int or float] upper bound point of line

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, … [, …])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, …])	Samples the surface or perimiter of the geometry.
sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry
translate(xyz)	translate geometry

copy	

`class modulus.sympy_utils.geometry_1d.`**`Point1D`**(*point*)

1D Point along x-axis

Parameters

point [int or float] x coordinate of the point

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, … [, …])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, …])	Samples the surface or perimiter of the geometry.
sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry
translate(xyz)	translate geometry

copy	

Primitives for 2D geometries see https://www.iquilezles.org/www/articles/distfunctions/distfunctions.html

`class modulus.sympy_utils.geometry_2d.`**`Channel2D`**(*point_1*, *point_2*)

2D Channel (no bounding curves in x-direction)

Parameters

point_1 [tuple with 2 ints or floats] lower bound point of channel

point_2 [tuple with 2 ints or floats] upper bound point of channel

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, ... [, ...])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, ...])	Samples the surface or perimiter of the geometry.
sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry
translate(xyz)	translate geometry

copy	

class modulus.sympy_utils.geometry_2d.**Circle**(*center*, *radius*)

2D Circle

Parameters

center [tuple with 2 ints or floats] center point of circle

radius [int or float] radius of circle

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, ... [, ...])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, ...])	Samples the surface or perimiter of the geometry.

Continued on next page

Table 35 – continued from previous page

sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry
translate(xyz)	translate geometry

copy

class modulus.sympy_utils.geometry_2d.**Ellipse**(*center*, *major*, *minor*)
2D Ellipse

Parameters

center [tuple with 2 ints or floats] center point of circle

radius [int or float] radius of circle

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, ...[, ...])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, ...])	Samples the surface or perimiter of the geometry.
sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry
translate(xyz)	translate geometry

copy

class modulus.sympy_utils.geometry_2d.**Line**(*point_1*, *point_2*, *normal*)
2D Line parallel to y-axis

Parameters

point_1 [tuple with 2 ints or floats] lower bound point of line segment

point_2 [tuple with 2 ints or floats] upper bound point of line segment

normal [int or float] normal direction of line (+1 or -1)

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, …[, …])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, …])	Samples the surface or perimiter of the geometry.
sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry
translate(xyz)	translate geometry

copy	

class modulus.sympy_utils.geometry_2d.**Rectangle**(*point_1*, *point_2*)

2D Rectangle

Parameters

point_1 [tuple with 2 ints or floats] lower bound point of rectangle

point_2 [tuple with 2 ints or floats] upper bound point of rectangle

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, …[, …])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, …])	Samples the surface or perimiter of the geometry.
sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry
translate(xyz)	translate geometry

copy	

class modulus.sympy_utils.geometry_2d.**Triangle**(*center*, *base*, *height*)

2D Isosceles Triangle Symmetrical axis parallel to y-axis

Parameters

center [tuple with 2 ints or floats] center of base of triangle

base [int or float] base of triangle

height [int or float] height of triangle

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, ...[, ...])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, ...])	Samples the surface or perimiter of the geometry.
sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry
translate(xyz)	translate geometry

copy	

Primitives for 3D geometries see https://www.iquilezles.org/www/articles/distfunctions/distfunctions.html

class modulus.sympy_utils.geometry_3d.**Box**(*point_1*, *point_2*)

3D Box/Cuboid

Parameters

point_1 [tuple with 3 ints or floats] lower bound point of box

point_2 [tuple with 3 ints or floats] upper bound point of box

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, ...[, ...])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, ...])	Samples the surface or perimiter of the geometry.

Continued on next page

Table 40 – continued from previous page

`sample_interior`(nr_points_per_volume, bounds)	Samples the interior of the geometry.
`scale`(x[, center])	scale geometry
`translate`(xyz)	translate geometry

copy	

class `modulus.sympy_utils.geometry_3d.`**`Channel`**(*point_1*, *point_2*)

3D Channel (no bounding surfaces in x-direction)

Parameters

point_1 [tuple with 3 ints or floats] lower bound point of channel

point_2 [tuple with 3 ints or floats] upper bound point of channel

Methods

`boundary_bc`(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
`dims`()	**Returns**
`interior_bc`(outvar_sympy, bounds, . . . [, . . .])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
`perimeter`([param_ranges])	Perimeter or surface area of geometry.
`repeat`(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
`rotate`(angle[, axis, center])	rotates geometry
`sample_boundary`(nr_points_per_area[, . . .])	Samples the surface or perimiter of the geometry.
`sample_interior`(nr_points_per_volume, bounds)	Samples the interior of the geometry.
`scale`(x[, center])	scale geometry
`translate`(xyz)	translate geometry

copy	

class `modulus.sympy_utils.geometry_3d.`**`Cone`**(*center*, *radius*, *height*)

3D Cone Axis parallel to z-axis

Parameters

center [tuple with 3 ints or floats] base center of cone

radius [int or float] base radius of cone

height [int or float] height of cone

Methods

`boundary_bc`(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
`dims`()	**Returns**
`interior_bc`(outvar_sympy, bounds, …[, …])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
`perimeter`([param_ranges])	Perimeter or surface area of geometry.
`repeat`(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
`rotate`(angle[, axis, center])	rotates geometry
`sample_boundary`(nr_points_per_area[, …])	Samples the surface or perimiter of the geometry.
`sample_interior`(nr_points_per_volume, bounds)	Samples the interior of the geometry.
`scale`(x[, center])	scale geometry
`translate`(xyz)	translate geometry

copy	

class `modulus.sympy_utils.geometry_3d.`**`Cylinder`**(*center*, *radius*, *height*)

3D Cylinder Axis parallel to z-axis

Parameters

center [tuple with 3 ints or floats] center of cylinder

radius [int or float] radius of cylinder

height [int or float] height of cylinder

Methods

`boundary_bc`(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
`dims`()	**Returns**
`interior_bc`(outvar_sympy, bounds, …[, …])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
`perimeter`([param_ranges])	Perimeter or surface area of geometry.
`repeat`(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
`rotate`(angle[, axis, center])	rotates geometry
`sample_boundary`(nr_points_per_area[, …])	Samples the surface or perimiter of the geometry.
`sample_interior`(nr_points_per_volume, bounds)	Samples the interior of the geometry.
`scale`(x[, center])	scale geometry
`translate`(xyz)	translate geometry

copy	

class `modulus.sympy_utils.geometry_3d.`**`ElliCylinder`**(*center*, *a*, *b*, *height*)

3D Elliptical Cylinder Axis parallel to z-axis

Approximation based on 4-arc ellipse construction https://www.researchgate.net/publication/241719740_Approximating_an_ellipse_with_four_circular_arcs

Please manually ensure a>b

Parameters

center [tuple with 3 ints or floats] center of base of ellipse

a [int or float] semi-major axis of ellipse

b [int or float] semi-minor axis of ellipse

height [int or float] height of elliptical cylinder

Methods

`boundary_bc`(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
`dims`()	**Returns**
`interior_bc`(outvar_sympy, bounds, … [, …])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
`perimeter`([param_ranges])	Perimeter or surface area of geometry.
`repeat`(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
`rotate`(angle[, axis, center])	rotates geometry
`sample_boundary`(nr_points_per_area[, …])	Samples the surface or perimiter of the geometry.
`sample_interior`(nr_points_per_volume, bounds)	Samples the interior of the geometry.
`scale`(x[, center])	scale geometry
`translate`(xyz)	translate geometry

copy	

class `modulus.sympy_utils.geometry_3d.`**`IsoTriangularPrism`**(*center*, *base*, *height*, *height_prism*)

2D Isosceles Triangular Prism Symmetrical axis parallel to y-axis

Parameters

center [tuple with 3 ints or floats] center of base of triangle

base [int or float] base of triangle

height [int or float] height of triangle

height_prism [int or float] height of triangular prism

Methods

`boundary_bc`(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.

Continued on next page

Table 45 – continued from previous page

dims()	**Returns**
interior_bc(outvar_sympy, bounds, … [, …])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, …])	Samples the surface or perimiter of the geometry.
sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry
translate(xyz)	translate geometry

copy	

class modulus.sympy_utils.geometry_3d.**Plane**(*point_1*, *point_2*, *normal*)

3D Plane perpendicular to x-axis

Parameters

point_1 [tuple with 3 ints or floats] lower bound point of plane

point_2 [tuple with 3 ints or floats] upper bound point of plane

normal [int or float] normal direction of plane (+1 or -1)

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, … [, …])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, …])	Samples the surface or perimiter of the geometry.
sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry
translate(xyz)	translate geometry

copy	

class modulus.sympy_utils.geometry_3d.**Sphere**(*center*, *radius*)

3D Sphere

Parameters

center [tuple with 3 ints or floats] center of sphere

radius [int or float] radius of sphere

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, . . . [, . . .])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, . . .])	Samples the surface or perimiter of the geometry.
sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry
translate(xyz)	translate geometry

copy	

class modulus.sympy_utils.geometry_3d.**Tetrahedron**(*center*, *radius*)

3D Tetrahedron The 4 symmetrically placed points are on a unit sphere. Centroid of the tetrahedron is at origin and lower face is parallel to x-y plane Reference: https://en.wikipedia.org/wiki/Tetrahedron

Parameters

center [tuple with 3 ints or floats] centroid of tetrahedron

radius [int or float] radius of circumscribed sphere

Methods

boundary_bc(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
dims()	**Returns**
interior_bc(outvar_sympy, bounds, . . . [, . . .])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
perimeter([param_ranges])	Perimeter or surface area of geometry.
repeat(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
rotate(angle[, axis, center])	rotates geometry
sample_boundary(nr_points_per_area[, . . .])	Samples the surface or perimiter of the geometry.
sample_interior(nr_points_per_volume, bounds)	Samples the interior of the geometry.
scale(x[, center])	scale geometry

Continued on next page

Table 48 – continued from previous page

`translate`(xyz)	translate geometry

copy

class `modulus.sympy_utils.geometry_3d.`**`Torus`**(*center*, *radius*, *radius_tube*)

3D Torus

Parameters

center [tuple with 3 ints or floats] center of torus

radius [int or float] distance from center to center of tube (major radius)

radius_tube [int or float] radius of tube (minor radius)

Methods

`boundary_bc`(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.
`dims`()	**Returns**
`interior_bc`(outvar_sympy, bounds, . . . [, . . .])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
`perimeter`([param_ranges])	Perimeter or surface area of geometry.
`repeat`(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
`rotate`(angle[, axis, center])	rotates geometry
`sample_boundary`(nr_points_per_area[, . . .])	Samples the surface or perimiter of the geometry.
`sample_interior`(nr_points_per_volume, bounds)	Samples the interior of the geometry.
`scale`(x[, center])	scale geometry
`translate`(xyz)	translate geometry

copy

class `modulus.sympy_utils.geometry_3d.`**`TriangularPrism`**(*center*, *side*, *height*)

3D Uniform Triangular Prism Axis parallel to z-axis

Parameters

center [tuple with 3 ints or floats] center of prism

side [int or float] side of equilateral base

height [int or float] height of prism

Methods

`boundary_bc`(outvar_sympy, batch_size_per_area)	Create a Boundary Condition of surface or perimiter of the geometry.

Continued on next page

Table 50 – continued from previous page

`dims()`	**Returns**
`interior_bc`(outvar_sympy, bounds, …[, …])	Create a Boundary Condition of interior of the geometry (where SDF is positive).
`perimeter`([param_ranges])	Perimeter or surface area of geometry.
`repeat`(spacing, repeat_lower, repeat_higher)	finite repetition of geometry
`rotate`(angle[, axis, center])	rotates geometry
`sample_boundary`(nr_points_per_area[, …])	Samples the surface or perimiter of the geometry.
`sample_interior`(nr_points_per_volume, bounds)	Samples the interior of the geometry.
`scale`(x[, center])	scale geometry
`translate`(xyz)	translate geometry

copy	

simple Sympy helper functions

`modulus.sympy_utils.functions.`**`line`**(*x*, *point_x_1*, *point_y_1*, *point_x_2*, *point_y_2*)

line function from point intercepts

Parameters

x [Sympy Symbol/Exp] the *x* in equation *y=a*x+b*

point_x_1 [Sympy Symbol/Exp, float, int] first intercept x position

point_y_1 [Sympy Symbol/Exp, float, int] first intercept y position

point_x_2 [Sympy Symbol/Exp, float, int] second intercept x position

point_y_2 [Sympy Symbol/Exp, float, int] second intercept y position

Returns

y [Sympy Expr] *y=slope*x+intercept*

`modulus.sympy_utils.functions.`**`parabola`**(*x*, *inter_1*, *inter_2*, *height*)

parabola from point intercepts

Parameters

x [Sympy Symbol/Exp] the *x* in equation *y=a*x*2+b*x+c*

inter_1 [Sympy Symbol/Exp, float, int] first intercept such that *y=0* when *x=inter_1*

inter_2 [Sympy Symbol/Exp, float, int] second intercept such that *y=0* when *x=inter_1*

height [Sympy Symbol/Exp, float, int] max height of parabola

Returns

y [Sympy Expr] *y=factor*(x-inter_1)*(x-+inter_2)*

`modulus.sympy_utils.functions.`**`parabola2D`**(*x*, *y*, *inter_1_x*, *inter_2_x*, *inter_1_y*, *inter_2_y*, *height*)

square parabola from point intercepts

Parameters

x [Sympy Symbol/Exp] the *x* in equation *z=parabola(x)*parabola(y)*

y [Sympy Symbol/Exp] the *y* in equation *z=a*x**2+b*y**2+c*xy+d*y+e*x+f*

inter_1_x [Sympy Symbol/Exp, float, int] first intercept such that *z=0* when *x=inter_1_x*

inter_2_x [Sympy Symbol/Exp, float, int] second intercept such that *z=0* when *x=inter_2_x*

inter_1_y [Sympy Symbol/Exp, float, int] first intercept such that *z=0* when *y=inter_1_y*

inter_2_y [Sympy Symbol/Exp, float, int] second intercept such that *z=0* when *y=inter_2_y*

height [Sympy Symbol/Exp, float, int] max height of parabola

Returns

y [Sympy Expr] *y=factor*(x-inter_1)*(x-+inter_2)*

Defines parameterized curves in SymPy

class `modulus.sympy_utils.curves.`**`Curve`**(*functions*, *ranges*, *area*, *criteria=True*)

Parameterized curves that keeps track of normals and area/perimeter

Parameters

functions [dictionary of SymPy Exprs] Parameterized curve in 1, 2 or 3 dimensions. For example, a circle might have ‘functions = {‘x’: cos(theta),

‘y’: sin(theta), ‘normal_x’: cos(theta), ‘normal_y’: sin(theta)}‘.

#TODO refactor to remove normals.

ranges [dictionary of Sympy Symbols and ranges] This gives the ranges for the parameters in the parameterized curve. For example, a circle might have *ranges = {theta: (0, 2*pi)}*.

area [float, int, SymPy Exprs] The surface area/perimeter of the curve.

criteria [SymPy Boolean Function] If this boolean expression is false then we do not sample their on curve. This can be used to enforce uniform sample probability.

Methods

approx_area([param_ranges, approx_nr])	**Parameters**
dims()	**Returns**
rotate(angle[, axis])	rotate curve
sample(nr_points[, param_ranges, quasirandom])	sample curve
scale(x)	scale curve
translate(xyz)	translate curve

compile_criteria	
compile_functions	
copy	
sample_ranges	
to_invar_fn	

`approx_area`(*param_ranges={}*, *approx_nr=100*)

Parameters

param_ranges: dict with of SymPy Symbols and their ranges If the curve is parameterized then you can provide ranges for the parameters with this.

approx_nr [int] Area might be difficult to compute it parameterized. In this case w approximate it by sampleing *self.area*, *approx_nr* number of times.

Returns

area [float] area of curve

compile_criteria(*criteria*, *param_ranges={}*)

compile_functions(*param_ranges={}*)

copy()

dims()

Returns

dims [list of strings] output can be ['x'], ['x','y'], or ['x','y','z']

rotate(*angle*, *axis='z'*)
rotate curve

Parameters

angle [float, SymPy Symbol/Exprs] angle of rotation

axis [str] axis of rotation

sample(*nr_points*, *param_ranges={}*, *quasirandom=False*)
sample curve

Parameters

nr_points [int] Number of points per area to sample.

param_ranges: dict with of SymPy Symbols and their ranges If the curve is parameterized then you can give the ranges for these parameters with this.

quasirandom [bool] If true then sample the points using the Halton sequences. Default is False.

sample_ranges(*batch_size*, *param_ranges={}*, *quasirandom=False*)

scale(*x*)
scale curve

Parameters

x [float, SymPy Symbol/Exprs] scale factor.

to_invar_fn(*criteria_fn*, *param_ranges={}*, *discard_criteria=True*, *quasirandom=False*)

translate(*xyz*)
translate curve

Parameters

xyz [tuple of floats, ints, SymPy Symbol/Exprs] translate curve by these values.

Helper functions for converting sympy equations to tensorflow

class modulus.sympy_utils.tf_printer.**CustomDerivativePrinter**(*settings=None*)

Attributes

order

Methods

`doprint`(expr)	Returns printer's representation for expr (as a string)
`emptyPrinter`(expr)	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
`set_global_settings`(**settings)	Set system-wide printing settings.

parenthesize	
stringify	

`modulus.sympy_utils.tf_printer.`**`tf_lambdify`**(*f*, *r*)
generates a tensorflow function from a sympy equation

Parameters

f [Sympy Exp, float, int, bool] the equation to convert to tensorflow. If float, int, or bool this gets converted to a constant function of value *f*.

r [list, dict] A list of the arguments for *f*. If dict then the keys of the dict are used.

Returns

tf_f [tensorflow function]

`modulus.csv_utils.`**`csv_to_dict`**(*filename*, *mapping=None*, *delimiter=', '*)
reads a csv file to a dictionary of columns

Parameters

filename [str] The file name to load from

mapping [None, dict] If None load entire csv file and store every column as a key in the dict. If *mapping* is not none use this to map keys from CSV to keys in dict.

delimiter: str The string used for separating values.

Returns

data [dict of numpy arrays] numpy arrays have shape [N, 1].

`modulus.csv_utils.`**`dict_to_csv`**(*dictonary*, *filename*)
saves a dict of numpy arrays to csv file

Parameters

dictionary [dict] dictionary of numpy arrays. The numpy arrays have a shape of [N, 1].

filename [str] The file name to save too

CHAPTER

ONE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

J

K

L

M

N

O

P

R

S

T

U

V

W

Z