torchquad

Release 0.5.0

ESA Advanced Concepts Team

Aug 03, 2025

OVERVIEW:

torchquad is a Python3 module for multidimensional numerical integration on the GPU. It uses autoray to support PyTorch and *other machine learning modules*.

You can see the latest code at https://github.com/esa/torchquad.

CHAPTER

ONE

GETTING STARTED

This is a brief introduction on how to set up *torchquad*.

1.1 Prerequisites

torchquad is built with

- autoray, which means the implemented quadrature supports NumPy and can be used for machine learning with modules such as PyTorch, JAX and Tensorflow, where it is fully differentiable
- conda, which will take care of all requirements for you

We recommend using conda, especially if you want to utilize the GPU. With PyTorch it will automatically set up CUDA and the cudatoolkit for you, for example. Note that *torchquad* also works on the CPU; however, it is optimized for GPU usage. torchquad's GPU support is tested only on NVIDIA cards with CUDA. We are investigating future support for AMD cards through ROCm.

For a detailed list of required packages and packages for numerical backends, please refer to the conda environment files environment.yml and environment_all_backends.yml. torchquad has been tested with JAX 0.2.25, NumPy 1.19.5, PyTorch 1.10.0 and Tensorflow 2.7.0; other versions of the backends should work as well.

1.2 Installation

First, we must make sure we have torchquad installed. The easiest way to do this is simply to

```
conda install torchquad -c conda-forge
```

Alternatively, it is also possible to use

```
pip install torchquad
```

The PyTorch backend with CUDA support can be installed with

```
conda install "cudatoolkit>=11.1" "pytorch>=1.9=*cuda*" -c conda-forge -c␣
↪pytorch
```

Note that since PyTorch is not yet on *conda-forge* for Windows, we have explicitly included it here using `-c pytorch`. Note also that installing PyTorch with *pip* may **not** set it up with CUDA support. Therefore, we recommend to use *conda*.

Here are installation instructions for other numerical backends:

```
conda install "tensorflow>=2.6.0=cuda*" -c conda-forge
pip install "jax[cuda]>=0.4.17" --find-links https://storage.googleapis.com/
↪jax-releases/jax_cuda_releases.html # linux only
conda install "numpy>=1.19.5" -c conda-forge
```

More installation instructions for numerical backends can be found in environment_all_backends.yml and at the backend documentations, for example https://pytorch.org/get-started/locally/, https://github.com/google/jax/#installation and https://www.tensorflow.org/install/gpu, and often there are multiple ways to install them.

1.3 Usage

Now you are ready to use *torchquad*. A brief example of how *torchquad* can be used to compute a simple integral can be found on our GitHub. For a more thorough introduction, please refer to the tutorial.

CHAPTER

TWO

TUTORIAL

torchquad is a dedicated module for numerical integration in arbitrary dimensions. This tutorial gives a more detailed look at its functionality and explores some performance considerations.

2.1 Minimal working example

```
# To avoid copying things to GPU memory,
# ideally allocate everything in torch on the GPU
# and avoid non-torch function calls
import torch
from torchquad import MonteCarlo, set_up_backend

# Enable GPU support if available and set the floating point precision
set_up_backend("torch", data_type="float32")

# The function we want to integrate, in this example
# f(x0,x1) = sin(x0) + e^x1 for x0=[0,1] and x1=[-1,1]
# Note that the function needs to support multiple evaluations at once (first
# dimension of x here)
# Expected result here is ~3.2698
def some_function(x):
    return torch.sin(x[:, 0]) + torch.exp(x[:, 1])

# Declare an integrator;
# here we use the simple, stochastic Monte Carlo integration method
mc = MonteCarlo()

# Compute the function integral by sampling 10000 points over domain
integral_value = mc.integrate(
    some_function,
    dim=2,
    N=10000,
    integration_domain=[[0, 1], [-1, 1]],
    backend="torch",
)
```

To set the default logger verbosity, change the `TORCHQUAD_LOG_LEVEL` environment variable; for example `export TORCHQUAD_LOG_LEVEL=DEBUG`. A *later section* in this tutorial shows how to choose a different numerical backend.

For information on using multiple GPUs, see the *Multi-GPU Usage* section.

2.2 Detailed Introduction

The main problem with higher-dimensional numerical integration is that the computation simply becomes too costly if the dimensionality, n, is large, as the number of evaluation points increases exponentially - this problem is known as the *curse of dimensionality*. This especially affects grid-based methods, but is, to some degree, also present for Monte Carlo methods, which also require larger numbers of points for convergence in higher dimensions.

At the time, *torchquad* offers the following integration methods for abritrary dimensionality.

Name	How it works	Spacing
Trapezoid rule	Creates a linear interpolant between two neighbouring points	Equal
Simpson's rule	Creates a quadratic interpolant between three neighbouring point	Equal
Boole's rule	Creates a more complex interpolant between five neighbouring points	Equal
Gaussian Quadrature	Uses orthogonal polynomials to generate a grid of sample points along with corresponding weights. A *GaussLegendre* implementation is provided as is a base *Gaussian* class that can be extended e.g., to other polynomials.	Unequal
Monte Carlo	Randomly chooses points at which the integrand is evaluated	Random
VEGAS Enhanced (VEGAS+)	Adaptive multidimensional Monte Carlo integration (VEGAS with adaptive stratified sampling)	Stratified sampling

2.3 Outline

This tutorial is a guide for new users to *torchquad* and is structured in the following way:

1. Example integration in one dimension (1-D) with PyTorch
2. Example integration in ten dimensions (10-D) with PyTorch
3. Some accuracy / runtime comparisons with scipy
4. Information on how to select a numerical backend
5. Example showing how gradients can be obtained w.r.t. the integration domain with PyTorch
6. Methods to speed up the integration
7. Multidimensional/Vectorized Integrands
8. Parametric Integration with Variable Domains
9. Custom Integrators

Feel free to test the code on your own computer as we go along.

2.4 Imports

Now let's get started! First, the general imports:

```
import scipy
import numpy as np
```

(continues on next page)

(continued from previous page)

```
# For benchmarking
import time
from scipy.integrate import nquad

# For plotting
import matplotlib.pyplot as plt

# To avoid copying things to GPU memory,
# ideally allocate everything in torch on the GPU
# and avoid non-torch function calls
import torch
torch.set_printoptions(precision=10) # Set displayed output precision to 10 digits

from torchquad import set_up_backend  # Necessary to enable GPU support
from torchquad import Trapezoid, Simpson, Boole, MonteCarlo, VEGAS # The available␣
↪integrators
from torchquad.utils.set_precision import set_precision
import torchquad
```

```
# Use this to enable GPU support and set the floating point precision
set_up_backend("torch", data_type="float32")
```

2.5 One-dimensional integration

To make it easier to understand the methods used in this notebook, we will start with an example in one dimension. If you are new to these methods or simply want a clearer picture, feel free to check out Patrick Walls' nice Python introduction to the Trapezoid rule and Simpson's rule in one dimension. Similarly, Tirthajyoti Sarkar has made a nice visual explanation of Monte Carlo integration in Python.

Let `f(x)` be the function $f(x) = e^x \cdot x^2$. Over the domain $[0, 2]$, the integral of `f(x)` is $\int_0^2 f(x)dx = \int_0^2 e^x \cdot x^2 dx = 2(e^2 - 1) = 12.778112197861300454...$

Let's declare the function and a simple function to print the absolute error, as well as remember the correct result.

```
def f(x):

    return torch.exp(x) * torch.pow(x, 2)

def print_error(result, solution):
    print("Results:", result.item())
    print(f"Abs. Error: {(torch.abs(result - solution).item()):.8e}")
    print(f"Rel. Error: {(torch.abs((result - solution) / solution).item()):.8e}")

solution = 2 * (torch.exp(torch.tensor([2.0])) - 1)
```

Note that we are using the torch versions of functions like ``exp`` to ensure that all variables are and stay on the GPU. Also, note: the unit imaginary number i is written as `j` in Python.

Let's plot the function briefly.

```
points = torch.linspace(0, 2, 100)
# Note that for plotting we have to move the values to the CPU first
plt.plot(points.cpu(), f(points).cpu())
plt.xlabel("$x$", fontsize=14)
plt.ylabel("f($x$)", fontsize=14)
```

Let's define the integration domain, set the precision to double, and initialize the integrator - let's start with the trapezoid rule.

```
# Integration domain is a list of lists to allow arbitrary dimensionality.
integration_domain = [[0, 2]]
# Initialize a trapezoid solver
tp = Trapezoid()
```

Now we are all set to compute the integral. Let's try it with just 101 sample points for now.

```
result = tp.integrate(f, dim=1, N=101, integration_domain=integration_domain)
print_error(result, solution)
```

Output: Results: 12.780082702636719
Abs. Error: 1.97029114e-03
Rel. Error: 1.54192661e-04

This is quite close already, as 1-D integrals are comparatively easy. Let's see what type of value we get for different integrators.

```
simp = Simpson()
result = simp.integrate(f, dim=1, N=101, integration_domain=integration_domain)
print_error(result, solution)
```

Output: Results: 12.778112411499023
Abs. Error: 0.00000000e+00
Rel. Error: 0.00000000e+00

```
mc = MonteCarlo()
result = mc.integrate(f, dim=1, N=101, integration_domain=integration_domain)
print_error(result, solution)
```

Output: Results: 13.32831859588623
Abs. Error: 5.50206184e-01
Rel. Error: 4.30584885e-02

```
vegas = VEGAS()
result = vegas.integrate(f, dim=1, N=101, integration_domain=integration_domain)
print_error(result, solution)
```

Output: Results: 21.83991813659668
Abs. Error: 9.06180573e+00
Rel. Error: 7.09166229e-01

Notably, Simpson's method is already sufficient for a perfect solution here with 101 points. Monte Carlo methods do not perform so well; they are more suited to higher-dimensional integrals. VEGAS currently requires a larger number of samples to function correctly (as it performs several iterations).

Let's step things up now and move to a ten-dimensional problem.

2.6 High-dimensional integration

Now, we will investigate the following ten-dimensional problem:

Let `f_2` be the function $f_2(x) = \sum_{i=1}^{10} \sin(x_i)$.

Over the domain $[0,1]^{10}$, the integral of `f_2` is $\int_0^1 \dots \int_0^1 \sum_{i=1}^{10} \sin(x_i) = 20 \sin^2(1/2) = 4.596976941318602825990633925570\dots$

Plotting this is tricky, so let's directly move to the integrals.

```
def f_2(x):
    return torch.sum(torch.sin(x), dim=1)


solution = 20 * (torch.sin(torch.tensor([0.5])) * torch.sin(torch.tensor([0.5])))
```

Let's start with just 3 points per dimension, i.e., $3^{10} = 59,049$ sample points.

Note: *torchquad* currently only supports equal numbers of points per dimension. We are working on giving the user more flexibility on this point.

```
# Integration domain is a list of lists to allow arbitrary dimensionality
integration_domain = [[0, 1]] * 10
N = 3 ** 10
```

```
tp = Trapezoid()  # Initialize a trapezoid solver
result = tp.integrate(f_2, dim=10, N=N, integration_domain=integration_domain)
print_error(result, solution)
```

```
Output: Results: 4.500804901123047
        Abs. Error: 9.61723328e-02
        Rel. Error: 2.09207758e-02
```

```
simp = Simpson()  # Initialize Simpson solver
result = simp.integrate(f_2, dim=10, N=N, integration_domain=integration_domain)
print_error(result, solution)
```

```
Output: Results: 4.598623752593994
        Abs. Error: 1.64651871e-03
        Rel. Error: 3.58174206e-04
```

```
boole = Boole()  # Initialize Boole solver
result = boole.integrate(f_2, dim=10, N=N, integration_domain=integration_domain)
print_error(result,solution)
```

```
Output: Results: 4.596974849700928
        Abs. Error: 2.38418579e-06
        Rel. Error: 5.18642082e-07
```

```
mc = MonteCarlo()
result = mc.integrate(f_2, dim=10, N=N, integration_domain=integration_domain, seed=42)
print_error(result, solution)
```

```
Output: Results: 4.598303318023682
        Abs. Error: 1.32608414e-03
        Rel. Error: 2.88468727e-04
```

```
vegas = VEGAS()
result = vegas.integrate(f_2, dim=10, N=N, integration_domain=integration_domain)
print_error(result, solution)
```

```
Output: Results: 4.598696708679199
        Abs. Error: 1.71947479e-03
        Rel. Error: 3.74044670e-04
```

Note that the Monte Carlo methods are much more competitive in this case. The bad convergence properties of the trapezoid method are visible while Simpson's and Boole's rule are still OK given the comparatively smooth integrand.

If you have been repeating the examples from this tutorial on your own computer, you might get `RuntimeError: CUDA out of memory` if you have a small GPU. In that case, you could also try to reduce the number of sample points (e.g., 3 per dimension). You can really see the curse of dimensionality fully at play here, since $5^{10} = 9,765,625$ but $3^{10} = 59,049$, reducing the number of sample points by a factor of 165. Note, however, that Boole's method cannot work for only 3 points per dimension, so the number of sample points is therefore automatically increased to 5 per dimension for this method.

2.7 Comparison with scipy

Let's explore how *torchquad*'s performance compares to scipy, the go-to tool for numerical integration. A more detailed exploration of this topic might be done as a side project at a later time. For simplicity, we will stick to a 5-D version of the $\sin(x)$ of the previous section. Let's declare it with numpy and torch. NumPy arrays will remain on the CPU and torch tensor on the GPU.

```
dimension = 5
integration_domain = [[0, 1]] * dimension
ground_truth = 2 * dimension * np.sin(0.5) * np.sin(0.5)

def f_3(x):
    return torch.sum(torch.sin(x), dim=1)

def f_3_np(*x):
    return np.sum(np.sin(x))
```

Now let's evaluate the integral using the scipy function `nquad`.

```
start = time.time()
opts = {"limit": 10, "epsabs": 1, "epsrel": 1}
result, _, details = nquad(f_3_np, integration_domain, opts=opts, full_output=True)
end = time.time()
print("Results:", result)
print("Abs. Error:", np.abs(result - ground_truth))
print(details)
print(f"Took {(end - start) * 1000.0:.3f} ms")
```

```
Output: Results: 2.2984884706593016
        Abs. Error: 0.0
        {'neval': 4084101}
        Took 33067.629 ms
```

Using scipy, we get the result in about 33 seconds on the authors' machine (this might take shorter or longer on your machine). The integral was computed with `nquad`, which on the inside uses the highly adaptive QUADPACK algorithm.

In any event, *torchquad* can reach the same accuracy much, much quicker by utilizing the GPU.

```
N = 37 ** dimension
simp = Simpson()  # Initialize Simpson solver
start = time.time()
result = simp.integrate(f_3, dim=dimension, N=N, integration_domain=integration_domain)
end = time.time()
print_error(result, ground_truth)
print("neval=", N)
print(f"Took {(end - start) * 1000.0:.3f} ms")
```

If you tried this yourself and ran out of CUDA memory, simply decrease N (this will, however, lead to a loss of accuracy).

Note that we use more evaluation points ($37^5 = 69,343,957$ for *torchquad* vs. $4,084,101$ for scipy), given the comparatively simple algorithm. Anyway, the decisive factor for this specific problem is runtime. A comparison with regard to function evaluations is difficult, as `nquad` provides no support for a fixed number of evaluations. This may follow in the future.

The results from using Simpson's rule in *torchquad* is:

```
Output: Results: 2.2984883785247803
        Abs. Error: 0.00000000e+00
        Rel. Error: 0.00000000e+00
        neval= 69343957
        Took 162.147 ms
```

In our case, *torchquad* with Simpson's rule was more than 300 times faster than `scipy.integrate.nquad`. We will add more elaborate integration methods over time; however, this tutorial should already showcase the advantages of numerical integration on the GPU.

Reasonably, one might prefer Monte Carlo integration methods for a 5-D problem. We might add this comparison to the tutorial in the future.

2.8 Using different backends with torchquad

This section shows how to select a different numerical backend for the quadrature. Let's change the minimal working example so that it uses Tensorflow instead of PyTorch:

```
import tensorflow as tf
from torchquad import MonteCarlo, set_up_backend

# Enable Tensorflow's NumPy behaviour and set the floating point precision
set_up_backend("tensorflow", data_type="float32")

# The integrand function rewritten for Tensorflow instead of PyTorch
def some_function(x):
    return tf.sin(x[:, 0]) + tf.exp(x[:, 1])

mc = MonteCarlo()
# Set the backend argument to "tensorflow" instead of "torch"
integral_value = mc.integrate(
    some_function,
    dim=2,
    N=10000,
    integration_domain=[[0, 1], [-1, 1]],
```

(continues on next page)

(continued from previous page)

```
    backend="tensorflow",
)
```

As the name suggests, the `set_up_backend` function configures a numerical backend so that it works with torchquad and it optionally sets the floating point precision. For Tensorflow this means in our code it enables NumPy behaviour and configures torchquad so that it uses float32 precision when initialising Tensors for Tensorflow. More details about *torchquad.set_up_backend()* can be found in its documentation.

To calculate an integral with Tensorflow we changed the `backend` argument of the `integrate` method. An alternative way to select Tensorflow as backend is to set the `integration_domain` argument to a `tf.Tensor` instead of a list.

The other code changes we did, for example rewriting the integrand, are not directly related to torchquad. To use NumPy or JAX we would analogously need to change the two backend arguments to `"numpy"` resp. `"jax"` and rewrite the integrand function.

2.9 Computing gradients with respect to the integration domain

torchquad allows fully automatic differentiation. In this tutorial, we will show how to extract the gradients with respect to the integration domain with the PyTorch backend. We selected the composite Trapezoid rule and the Monte Carlo method to showcase that getting gradients is possible for both deterministic and stochastic methods.

```
import torch
from torchquad import MonteCarlo, Trapezoid, set_up_backend

def test_function(x):
    """V shaped test function."""
    return 2 * torch.abs(x)

set_up_backend("torch", data_type="float64")
# Number of function evaluations
N = 10000

# Calculate a gradient with the MonteCarlo integrator
# Define the integrator
integrator_mc = MonteCarlo()
# Integration domain
domain = torch.tensor([[-1.0, 1.0]])
# Enable the creation of a computational graph for gradient calculation.
domain.requires_grad = True
# Calculate the 1-D integral by using the previously defined test_function
# with MonteCarlo; set a RNG seed to get reproducible results
result_mc = integrator_mc.integrate(
    test_function, dim=1, N=N, integration_domain=domain, seed=0
)
# Compute the gradient with a backward pass
result_mc.backward()
gradient_mc = domain.grad

# Calculate a gradient analogously with the composite Trapezoid integrator
integrator_tp = Trapezoid()
domain = torch.tensor([[-1.0, 1.0]])
domain.requires_grad = True
result_tp = integrator_tp.integrate(
```

(continues on next page)

(continued from previous page)

```
    test_function, dim=1, N=N, integration_domain=domain
)
result_tp.backward()
gradient_tp = domain.grad

# Show the results
print(f"Gradient result for MonteCarlo: {gradient_mc}")
print(f"Gradient result for Trapezoid: {gradient_tp}")
```

The code above calculates the integral for a 1-D test-function `test_function()` in the [-1,1] domain and prints the gradients with respect to the integration domain. The command `domain.requires_grad = True` enables the creation of a computational graph, and it shall be called before calling the `integrate(...)` method. Gradients computation is, then, performed calling `result.backward()`. The output of the print statements is as follows:

```
Gradient result for MonteCarlo: tensor([[-1.9828,  2.0196]])
Gradient result for Trapezoid: tensor([[-2.0000,  2.0000]])
```

2.10 Speedups for repeated quadrature

2.10.1 Compiling the integrate method

To speed up the quadrature in situations where it is executed often with the same number of points `N`, dimensionality `dim`, and shape of the `integrand` (see *the next section* for more information on integrands), we can JIT-compile the performance-relevant parts of the integrate method:

```
import time
import torch
from torchquad import Boole, set_up_backend


def example_integrand(x):
    return torch.sum(torch.sin(x), dim=1)


set_up_backend("torch", data_type="float32")
N = 912673
dim = 3
integrator = Boole()
domains = [torch.tensor([[-1.0, y]] * dim) for y in range(5)]

# Integrate without compilation
times_uncompiled = []
for integration_domain in domains:
    t0 = time.perf_counter()
    integrator.integrate(example_integrand, dim, N, integration_domain)
    times_uncompiled.append(time.perf_counter() - t0)

# Integrate with partial compilation
integrate_jit_compiled_parts = integrator.get_jit_compiled_integrate(
    dim, N, backend="torch"
)
```

(continues on next page)

(continued from previous page)

```
times_compiled_parts = []
for integration_domain in domains:
    t0 = time.perf_counter()
    integrate_jit_compiled_parts(example_integrand, integration_domain)
    times_compiled_parts.append(time.perf_counter() - t0)

# Integrate with everything compiled
times_compiled_all = []
integrate_compiled = None
for integration_domain in domains:
    t0 = time.perf_counter()
    if integrate_compiled is None:
        integrate_compiled = torch.jit.trace(
            lambda dom: integrator.integrate(example_integrand, dim, N, dom),
            (integration_domain,),
        )
    integrate_compiled(integration_domain)
    times_compiled_all.append(time.perf_counter() - t0)

print(f"Uncompiled times: {times_uncompiled}")
print(f"Partly compiled times: {times_compiled_parts}")
print(f"All compiled times: {times_compiled_all}")
speedups = [
    (1.0, tu / tcp, tu / tca)
    for tu, tcp, tca in zip(times_uncompiled, times_compiled_parts, times_compiled_all)
]
print(f"Speedup factors: {speedups}")
```

This code shows two ways of compiling the integration. In the first case, we use `integrator.get_jit_compiled_integrate`, which internally uses `torch.jit.trace` to compile performance-relevant code parts except the integrand evaluation. In the second case we directly compile `integrator.integrate`. The function created in the first case may be a bit slower, but it works even if the integrand cannot be compiled and we can re-use it with other integrand functions. The compilations happen in the first iteration of the for loops and in the following iterations the previously compiled functions are re-used.

With JAX and Tensorflow it is also possible to compile the integration. In comparison to compilation with PyTorch, we would need to use `jax.jit` or `tf.function` instead of `torch.jit.trace` to compile the whole integrate method. `get_jit_compiled_integrate` automatically uses the compilation function which fits to the numerical backend. There is a special case with JAX and MonteCarlo: If a function which executes the integrate method is compiled with `jax.jit`, the random number generator's current PRNGKey value needs to be an input and output of this function so that MonteCarlo generates different random numbers in each integration. torchquad's RNG class has methods to set and get this PRNGKey value.

The disadvantage of compilation is the additional time required to compile or re-compile the code, so if the integrate method is executed only a few times or certain arguments, e.g. `N`, change often, the program may be slower overall.

2.10.2 Reusing sample points

With the MonteCarlo and composite Newton Cotes integrators it is possible to execute the methods for sample point calculation, integrand evaluation and result calculation separately. This can be helpful to obtain a speedup in situations where integration happens very often with the same `integration_domain` and `N` arguments. However, separate sample point calculation has some disadvantages:

- The code is more complex.

- The memory required for the grid points is not released after each integration.
- With MonteCarlo the same sample points would be used for each integration, which corresponds to a fixed seed.

Here is an example where we integrate two functions with Boole and use the same sample points for both functions:

```
import torch
from torchquad import Boole

def integrand1(x):
    return torch.sin(x[:, 0]) + torch.exp(x[:, 1])

def integrand2(x):
    return torch.prod(torch.cos(x), dim=1)

# The integration domain, dimensionality and number of evaluations
# For the calculate_grid method we need a Tensor and not a list.
integration_domain = torch.Tensor([[0.0, 1.0], [-1.0, 1.0]])
dim = 2
N = 9409

# Initialize the integrator
integrator = Boole()
# Calculate sample points and grid information for the result calculation
grid_points, hs, n_per_dim = integrator.calculate_grid(N, integration_domain)

# Integrate the first integrand with the sample points
function_values, _ = integrator.evaluate_integrand(integrand1, grid_points)
integral1 = integrator.calculate_result(function_values, dim, n_per_dim, hs, integration_
↪domain)

# Integrate the second integrand with the same sample points
function_values, _ = integrator.evaluate_integrand(integrand2, grid_points)
integral2 = integrator.calculate_result(function_values, dim, n_per_dim, hs, integration_
↪domain)

print(f"Quadrature results: {integral1}, {integral2}")
```

2.11 Multidimensional/Vectorized Integrands

If you wish to evaluate many different integrands over the same domain, it may be faster to pass in a vectorized formulation if possible. Our inspiration for this came from scipy's own vectorization capabilities e.g., from its `fixed_quad` method.

As an example, here we evaluate a similar integrand many times for different values of `a` and `b`. This is an example that could be sped up by a vectorized evaluation of all integrals:

```
def parametrized_integrand(x, a, b):
    return torch.sqrt(torch.cos(torch.sin((a + b) * x)))

a_params = torch.arange(40)
b_params = torch.arange(10, 20)
integration_domain = torch.Tensor([[0, 1]])
simp = Simpson()
```

(continues on next page)

(continued from previous page)

```
result = torch.stack([torch.Tensor([simp.integrate(lambda x: parametrized_integrand(x, a,
↪ b), dim=1, N=101, integration_domain=integration_domain) for a in a_params]) for b in␣
↪b_params])
```

Now let's see how to do this a bit more simply, and in a way that provides signficant speedup as the size of the integrand's `grid` grows:

```
grid = torch.stack([torch.Tensor([a + b for a in a_params]) for b in b_params])

def integrand(x):
    return torch.sqrt(torch.cos(torch.sin(torch.einsum("i,jk->ijk", x.flatten(), grid))))

result_vectorized = simp.integrate(integrand, dim=1, N=101, integration_
↪domain=integration_domain)

torch.all(torch.isclose(result_vectorized, result)) # True!
```

Note

VEGAS does not support multi-dimensional integrands. If you would like this, please consider opening an issue or PR.

2.12 Parametric Integration with Variable Domains

Sometimes you need to perform multiple integrations where both the integrand and the integration domain depend on parameters. This is particularly useful in applications where you need to compute integrals for many different parameter values simultaneously.

For example, you might want to compute:

$$I(a, b) =$$
$$int_a^b f(x, a, b)dx$$

for multiple values of a and b simultaneously.

Currently, torchquad doesn't have built-in support for parametric domains, but you can extend the existing integrators to handle this case. Below is an example of how to create a custom integrator that supports batch 1D integration with variable domains:

```
import torch
from loguru import logger
from autoray import numpy as anp
from autoray import infer_backend
from torchquad import Gaussian


class Batch1DIntegrator(Gaussian):
    """Custom integrator for batch 1D integration with variable domains.

    This integrator can compute multiple integrals with different domains
    in a single call, providing significant speedup over sequential computation.
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.disable_integration_domain_check = True

    def _resize_roots(self, integration_domain, roots):
        """Resize roots for batched integration domains.

        Args:
            integration_domain: Shape [batch_size, 2] for multiple domains
            roots: Shape [N] - the Gaussian quadrature nodes

        Returns:
            Resized roots with shape [batch_size, N]
        """
        if integration_domain.ndim == 1:
            # Single domain case - use parent implementation
            return super()._resize_roots(integration_domain, roots)

        # Batch case
        assert roots.ndim == 1
        assert integration_domain.ndim == 2
        assert integration_domain.shape[-1] == 2

        roots = roots.to(integration_domain.device)

        # Extract bounds for all domains
        a = integration_domain[:, 0:1]  # Shape [batch_size, 1]
        b = integration_domain[:, 1:2]  # Shape [batch_size, 1]

        # Broadcast and transform roots for each domain
        roots_expanded = roots.unsqueeze(0)  # [1, N]

        # Transform from [-1, 1] to [a, b] for each domain
        out = ((b - a) / 2) * roots_expanded + ((a + b) / 2)  # [batch_size, N]

        return out

    def integrate(self, fn, dim, N, integration_domain=None, backend="torch"):
        """Integrate function over multiple domains in a single call.

        Args:
            fn: Function to integrate
            dim: Must be 1 for this implementation
            N: Number of quadrature points
            integration_domain: Shape [batch_size, 2] for batch integration
            backend: Must be "torch"

        Returns:
            Tensor of shape [batch_size] with integral results
        """
        assert dim == 1
```

(continues on next page)

(continued from previous page)

```
        assert backend == "torch"

        if integration_domain.ndim == 1:
            integration_domain = integration_domain.reshape(1, 2)

        batch_size = integration_domain.shape[0]

        # Get Gaussian quadrature points and weights
        N = self._adjust_N(dim=1, N=N)
        roots = self._roots(N, backend, integration_domain.requires_grad)
        weights = self._weights(N, dim, backend)

        # Resize roots for all domains at once
        grid_points = self._resize_roots(integration_domain, roots)  # [batch_size, N]

        # Evaluate integrand at all points
        # Flatten for function evaluation: [batch_size * N, 1]
        points_flat = grid_points.reshape(-1, 1)
        function_values = fn(points_flat)  # [batch_size * N]

        # Reshape back to [batch_size, N]
        function_values = function_values.reshape(batch_size, N)

        # Apply weights and sum for each domain
        weighted_values = function_values * weights.unsqueeze(0)

        # Scale by domain width and sum
        domain_widths = (integration_domain[:, 1] - integration_domain[:, 0]) / 2
        results = domain_widths * weighted_values.sum(dim=1)

        return results
```

Now let's see a concrete example of using this for parametric integration:

```
# Example 1: Compute multiple integrals in ONE call
# I(a) = integral from 0 to a of x^2 dx = a^3/3
# for a = 1, 2, 3, 4, 5

def integrand(x):
    # x has shape [batch_size * N, 1] where N is the number of quadrature points
    return x[:, 0] ** 2

# Create multiple integration domains
upper_bounds = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0])
domains = torch.stack([torch.zeros_like(upper_bounds), upper_bounds], dim=1)
print(f"Integration domains shape: {domains.shape}")
print(f"Domains:\n{domains}")

# Initialize the batch integrator
batch_integrator = Batch1DIntegrator()

# Compute ALL integrals in ONE call - this is the key difference!
```

(continues on next page)

(continued from previous page)

```
results = batch_integrator.integrate(integrand, dim=1, N=50, integration_domain=domains)

# Analytical solution: a^3/3
analytical = upper_bounds ** 3 / 3

print(f"\nResults shape: {results.shape}")
print(f"Numerical results: {results}")
print(f"Analytical results: {analytical}")
print(f"Absolute errors: {torch.abs(results - analytical)}")
```

Output:

```
Integration domains shape: torch.Size([5, 2])
Domains:
tensor([[0., 1.],
        [0., 2.],
        [0., 3.],
        [0., 4.],
        [0., 5.]])

Results shape: torch.Size([5])
Numerical results: tensor([ 0.3333,  2.6667,  9.0000, 21.3333, 41.6667])
Analytical results: tensor([ 0.3333,  2.6667,  9.0000, 21.3333, 41.6667])
Absolute errors: tensor([9.9341e-09, 7.9473e-08, 1.7764e-14, 6.3578e-07, 1.2716e-06])
```

The key advantage of this approach is that all integrals are computed in a single vectorized operation, which can provide significant speedups:

```
# Performance comparison - batch vs sequential
import time
from torchquad import GaussLegendre

# Many domains
n_domains = 500
many_upper_bounds = torch.linspace(0.1, 5.0, n_domains)
many_domains = torch.stack([torch.zeros(n_domains), many_upper_bounds], dim=1)

# Batch computation
start = time.time()
batch_results = batch_integrator.integrate(integrand, dim=1, N=50, integration_
↪domain=many_domains)
batch_time = time.time() - start

# Sequential computation for comparison
standard_integrator = GaussLegendre()
start = time.time()
sequential_results = []
for i in range(n_domains):
    result = standard_integrator.integrate(
        integrand, dim=1, N=50,
        integration_domain=[[0.0, many_upper_bounds[i].item()]]
    )
```

(continues on next page)

(continued from previous page)

```
    sequential_results.append(result)
sequential_time = time.time() - start

print(f"Computed {n_domains} integrals:")
print(f"Batch time: {batch_time:.4f} seconds")
print(f"Sequential time: {sequential_time:.4f} seconds")
print(f"Speedup: {sequential_time/batch_time:.2f}x")
```

Output:

```
Computed 500 integrals:
Batch time: 0.0010 seconds
Sequential time: 0.2289 seconds
Speedup: 228.90x
```

This approach can be extended to more complex scenarios where both the integrand and the domain depend on parameters. The key insight is that by properly vectorizing the computation, we can achieve significant performance improvements over sequential integration.

Note

This implementation is specifically for 1D integrals. Extending it to higher dimensions would require more careful handling of the grid generation and result calculation.

2.13 Multi-GPU Usage

While torchquad doesn't have a built-in device parameter for selecting specific GPUs, you can effectively use multiple GPUs using standard PyTorch practices and environment variables.

2.13.1 Using CUDA_VISIBLE_DEVICES

The recommended way to control which GPU torchquad uses is through the `CUDA_VISIBLE_DEVICES` environment variable:

```
# Use only GPU 0
export CUDA_VISIBLE_DEVICES=0
python your_integration_script.py

# Use only GPU 1
export CUDA_VISIBLE_DEVICES=1
python your_integration_script.py

# Use GPUs 0 and 2
export CUDA_VISIBLE_DEVICES=0,2
python your_integration_script.py
```

This approach has several advantages:

- **Clean separation**: Each process sees only the specified GPU(s)
- **No code changes**: Your torchquad code remains unchanged
- **Standard practice**: This is the recommended approach in the PyTorch community

- **Process isolation**: Different processes can use different GPUs without interference

2.13.2 Parallel Processing with Multiple GPUs

For compute-intensive workloads that can be parallelized, you can spawn multiple processes, each using a different GPU:

```
import multiprocessing as mp
import os
import torch
from torchquad import MonteCarlo, set_up_backend

def run_integration_on_gpu(gpu_id, integration_params, result_queue):
    """Run integration on a specific GPU"""
    # Set the GPU for this process
    os.environ['CUDA_VISIBLE_DEVICES'] = str(gpu_id)

    # Initialize torchquad
    set_up_backend("torch", data_type="float32")

    # Your integration code here
    mc = MonteCarlo()
    result = mc.integrate(
        integration_params['fn'],
        dim=integration_params['dim'],
        N=integration_params['N'],
        integration_domain=integration_params['domain'],
        backend="torch"
    )

    result_queue.put((gpu_id, result.item()))

def parallel_integration_example():
    """Example of parallel integration across multiple GPUs"""
    # Define your integration parameters
    def integrand(x):
        return torch.sin(x[:, 0]) + torch.exp(x[:, 1])

    integration_params = {
        'fn': integrand,
        'dim': 2,
        'N': 100000,
        'domain': [[0, 1], [-1, 1]]
    }

    # Check available GPUs
    available_gpus = list(range(torch.cuda.device_count()))
    if not available_gpus:
        print("No CUDA GPUs available")
        return

    print(f"Using GPUs: {available_gpus}")

    # Create processes for each GPU
```

(continues on next page)

(continued from previous page)

```
    processes = []
    result_queue = mp.Queue()

    for gpu_id in available_gpus:
        p = mp.Process(
            target=run_integration_on_gpu,
            args=(gpu_id, integration_params, result_queue)
        )
        p.start()
        processes.append(p)

    # Collect results
    results = {}
    for _ in available_gpus:
        gpu_id, result = result_queue.get()
        results[gpu_id] = result

    # Wait for all processes to complete
    for p in processes:
        p.join()

    print("Results from each GPU:")
    for gpu_id, result in sorted(results.items()):
        print(f"  GPU {gpu_id}: {result:.6f}")

    return results
```

2.13.3 Use Cases for Multi-GPU Integration

1. **Parameter Sweeps**: Run the same integration with different parameters on different GPUs
2. **Different Integration Methods**: Compare multiple integration methods simultaneously
3. **Monte Carlo with Different Seeds**: Run multiple Monte Carlo integrations with different random seeds for error estimation
4. **Batch Processing**: Process multiple independent integration problems in parallel

2.13.4 Example: Monte Carlo Error Estimation

```
import subprocess
import numpy as np

def monte_carlo_error_estimation():
    """Estimate integration error using multiple independent Monte Carlo runs"""

    # Script content for each GPU process
    script_template = '''
```

import os import torch from torchquad import MonteCarlo, set_up_backend

os.environ[‘CUDA_VISIBLE_DEVICES’] = ‘{gpu_id}’ set_up_backend(“torch”, data_type=”float32”)

def integrand(x):
    return torch.sin(x[:, 0]) + torch.exp(x[:, 1])

mc = MonteCarlo() result = mc.integrate(

integrand, dim=2, N=50000, integration_domain=[[0, 1], [-1, 1]], seed={seed}, backend="torch"

)

print(result.item()) '"

num_gpus = torch.cuda.device_count() runs_per_gpu = 5

results = [] processes = []

for gpu_id in range(num_gpus):

for run in range(runs_per_gpu):
seed = gpu_id * runs_per_gpu + run + 1000 script = script_template.format(gpu_id=gpu_id, seed=seed)

Launch subprocess process = subprocess.Popen(

['python', '-c', script], stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True

) processes.append(process)

Collect results for process in processes:

stdout, stderr = process.communicate() if process.returncode == 0:

results.append(float(stdout.strip()))

else:
print(f"Error in subprocess: {stderr}")

Calculate statistics results = np.array(results) mean_result = np.mean(results) std_error = np.std(results) / np.sqrt(len(results))

print(f"Monte Carlo Results from {len(results)} runs:") print(f" Mean: {mean_result:.6f}") print(f" Standard Error: {std_error:.6f}") print(f" 95% Confidence Interval: [{mean_result - 1.96*std_error:.6f}, {mean_result + 1.96*std_error:.6f}]")

return mean_result, std_error

2.13.5 Best Practices for Multi-GPU Usage

1. **Use CUDA_VISIBLE_DEVICES**: This is the cleanest way to control GPU selection
2. **Process-based parallelism**: Use `multiprocessing` rather than threading for true parallelism
3. **Memory management**: Each GPU process will have its own memory space
4. **Load balancing**: Distribute work evenly across available GPUs
5. **Error handling**: Handle cases where specific GPUs might be unavailable or busy

Warning

Avoid using `torch.cuda.set_device()` within torchquad applications, as this can interfere with torchquad's internal device management. Always use `CUDA_VISIBLE_DEVICES` instead.

2.14 Custom Integrators

It is of course possible to extend our provided Integrators, perhaps for a special class of functions or for a new algorithm.

```
import scipy
from torchquad import Gaussian
from autoray import numpy as anp

class GaussHermite(Gaussian):
    """
    Gauss Hermite quadrature rule in torch, for integrals of the form :math:`\\int_{-\\
↪infty}^{+\\infty} e^{-x^{2}} f(x) dx`. It will correctly integrate
    polynomials of degree :math:`2n - 1` or less over the interval
    :math:`[-\\infty, \\infty]` with weight function :math:`f(x) = e^{-x^2}`. See https://
↪en.wikipedia.org/wiki/Gauss%E2%80%93Hermite_quadrature
    """

    def __init__(self):
        super().__init__()
        self.name = "Gauss-Hermite"
        self._root_fn = scipy.special.roots_hermite

    @staticmethod
    def _apply_composite_rule(cur_dim_areas, dim, hs, domain):
        """Apply "composite" rule for gaussian integrals
        cur_dim_areas will contain the areas per dimension
        """
        # We collapse dimension by dimension
        for _ in range(dim):
            cur_dim_areas = anp.sum(cur_dim_areas, axis=len(cur_dim_areas.shape) - 1)
        return cur_dim_areas

gh=GaussHermite()
integral=gh.integrate(lambda x: 1-x,dim=1,N=200) #integral from -inf to inf of np.exp(-
↪(x**2))*(1-x)
# Computed integral was 1.7724538509055168.
# analytic result = sqrt(pi)
```

CHAPTER

THREE

INTEGRATION METHODS

This is the list of all available integration methods in *torchquad*.

We are continuously implementing new methods in our library. For the code behind the integration methods, please see the code page or check out our full code and latest news at https://github.com/esa/torchquad.

Contents

3.1 Stochastic Methods

3.1.1 Monte Carlo Integrator

class torchquad.**MonteCarlo**

Monte Carlo integration

integrate(*fn*, *dim*, *N=1000*, *integration_domain=None*, *seed=None*, *rng=None*, *backend=None*)

Integrates the passed function on the passed domain using vanilla Monte Carlo Integration.

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the function's domain over which to integrate.
- **N** (*int, optional*) – Number of sample points to use for the integration. Defaults to 1000.

- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It can also determine the numerical backend.
- **seed** (*int, optional*) – Random number generation seed to the sampling point creation, only set if provided. Defaults to None.
- **rng** (RNG, *optional*) – An initialised RNG; this can be used when compiling the function for Tensorflow
- **backend** (*string, optional*) – Numerical backend. Defaults to integration_domain's backend if it is a tensor and otherwise to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Raises
ValueError – If len(integration_domain) != dim

Returns
Integral value

Return type
backend-specific number

3.1.2 VEGAS Enhanced

class torchquad.**VEGAS**

VEGAS Enhanced. Refer to https://arxiv.org/abs/2009.05112 . Implementation inspired by https://github.com/ycwu1030/CIGAR/ . EQ <n> refers to equation <n> in the above paper. JAX and Tensorflow are unsupported. For Tensorflow there exists a VEGAS+ implementation called VegasFlow: https://github.com/N3PDF/vegasflow

integrate(*fn*, *dim*, *N=10000*, *integration_domain=None*, *seed=None*, *rng=None*, *use_grid_improve=True*, *eps_rel=0*, *eps_abs=0*, *max_iterations=20*, *use_warmup=True*, *backend=None*)

Integrates the passed function on the passed domain using VEGAS.

If the integrand output is far away from zero, i.e. lies within [b, b+c] for a constant b with large absolute value and small constant c, VEGAS does not adapt well to the integrand. Shifting the integrand so that it is close to zero may improve the accuracy of the calculated integral in this case. This method does not support multi-dimensional/vectorized integrands (i.e., integrating an integrand repeatedly over a grid of points).

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the function's domain over which to integrate.
- **N** (*int, optional*) – Approximate maximum number of function evaluations to use for the integration. This value can be exceeded if the vegas stratification distributes evaluations per hypercube very unevenly. Defaults to 10000.
- **integration_domain** (*list, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim.
- **seed** (*int, optional*) – Random number generation seed for the sampling point creation; only set if provided. Defaults to None.
- **rng** (RNG, *optional*) – An initialised RNG; this can be used as alternative to the seed argument and to avoid problems with integrand functions which reset PyTorch's RNG seed.
- **use_grid_improve** (*bool, optional*) – If True, improve the vegas map after each iteration. Defaults to True.
- **eps_rel** (*float, optional*) – Relative error to abort at. Defaults to 0.

- **eps_abs** (*float, optional*) – Absolute error to abort at. Defaults to 0.
- **max_iterations** (*int, optional*) – Maximum number of vegas iterations to perform. The number of performed iterations is usually lower than this value because the number of sample points per iteration increases every fifth iteration. Defaults to 20.
- **use_warmup** (*bool, optional*) – If True, execute a warmup to initialize the vegas map. Defaults to True.
- **backend** (*string, optional*) – Numerical backend. "jax" and "tensorflow" are unsupported. Defaults to integration_domain's backend if it is a tensor and otherwise to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Raises
ValueError – If the integration_domain or backend argument is invalid

Returns
Integral value

Return type
backend-specific float

3.2 Deterministic Methods

3.2.1 Boole's Rule

class torchquad.**Boole**

Boole's rule. See https://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas#Closed_Newton%E2%80%93Cotes_formulas .

integrate(*fn*, *dim*, *N=None*, *integration_domain=None*, *backend=None*)

Integrates the passed function on the passed domain using Boole's rule.

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the integration domain.
- **N** (*int, optional*) – Total number of sample points to use for the integration. N has to be such that N^(1/dim) - 1 % 4 == 0. Defaults to 5 points per dimension if None is given.
- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It can also determine the numerical backend.
- **backend** (*string, optional*) – Numerical backend. Defaults to integration_domain's backend if it is a tensor and otherwise to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Returns
Integral value

Return type
backend-specific number

3.2.2 Simpson's Rule

class torchquad.**Simpson**

Simpson's rule. See https://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas#Closed_Newton%E2%80%93Cotes_formulas .

integrate(*fn*, *dim*, *N=None*, *integration_domain=None*, *backend=None*)

Integrates the passed function on the passed domain using Simpson's rule.

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the integration domain.
- **N** (*int, optional*) – Total number of sample points to use for the integration. Should be odd. Defaults to 3 points per dimension if None is given.
- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It can also determine the numerical backend.
- **backend** (*string, optional*) – Numerical backend. Defaults to integration_domain's backend if it is a tensor and otherwise to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Returns
Integral value

Return type
backend-specific number

3.2.3 Trapezoid Rule

class torchquad.**Trapezoid**

Trapezoidal rule. See https://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas#Closed_Newton%E2%80%93Cotes_formulas .

integrate(*fn*, *dim*, *N=1000*, *integration_domain=None*, *backend=None*)

Integrates the passed function on the passed domain using the trapezoid rule.

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the function to integrate.
- **N** (*int, optional*) – Total number of sample points to use for the integration. Defaults to 1000.
- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It can also determine the numerical backend.
- **backend** (*string, optional*) – Numerical backend. Defaults to integration_domain's backend if it is a tensor and otherwise to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Returns
Integral value

Return type
backend-specific number

3.2.4 Gaussian Quadrature

class torchquad.**Gaussian**

Base method for Gaussian Quadrature. Different Gaussian methods should inherit from this class, and override as necessary methods. Default behaviour is Gauss-Legendre quadrature on [-1,1] (i.e., this "parent" class should __not__ be used directly with other integration domains, and for this parent class *integration_domain* as an argument to *integrate* is ignored internally).

For an example of how to properly override the behavior to acheive different Gaussian Integration methods, please see the *Custom Integrators* section of the Tutorial or the implementation of *GaussLegendre*.

The primary methods/attributes of interest to override are *_root_fn* (for different polynomials, like *numpy.polynomial.legendre.leggauss*), *_apply_composite_rule* (as in other integration methods), and *_resize_roots* (for handling different integration domains).

name

A human-readable name for the integral.

Type
str

_root_fn

A function that returns roots and weights like *numpy.polynomial.legendre.leggauss*.

Type
function

_root_args

a way of adding information to be passed into *_root_fn* as needed. This is then used when caching roots/weights to potentially distinguish different calls to *_root_fn* based on arguments.

Type
tuple

_cache

a cache for roots and weights, used internally.

Type
dict

_resize_roots(*integration_domain*, *roots*)

Resize the roots based on domain of [a,b]. Default behavior is to simply return the roots, unsized by *integraton_domain*.

Parameters

- **integration_domain** (*backend tensor*) – domain
- **roots** (*backend tensor*) – polynomial nodes

Returns
rescaled roots

Return type
backend tensor

integrate(*fn*, *dim*, *N=8*, *integration_domain=None*, *backend=None*)

Integrates the passed function on the passed domain using a Gaussian rule (Gauss-Legendre on [-1,1] as a default).

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the integration domain.
- **N** (*int, optional*) – Total number of sample points to use for the integration. Should be odd. Defaults to 3 points per dimension if None is given.
- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It also determines the numerical backend if possible.
- **backend** (*string, optional*) – Numerical backend. This argument is ignored if the backend can be inferred from integration_domain. Defaults to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Returns

Integral value

Return type

backend-specific number

class torchquad.**GaussLegendre**

Gauss Legendre quadrature rule in torch for any domain [a,b]. See https://en.wikipedia.org/wiki/Gaussian_quadrature#Gauss%E2%80%93Legendre_quadrature.

Examples

```
>>> gl=torchquad.GaussLegendre()
>>> integral = gl.integrate(lambda x:np.sin(x), dim=1, N=101, integration_
↪domain=[[0,5]]) #integral from 0 to 5 of np.sin(x)
|TQ-INFO| Computed integral was 0.7163378000259399 #analytic result = 1-np.cos(5)
```

integrate(*fn*, *dim*, *N=8*, *integration_domain=None*, *backend=None*)

Integrates the passed function on the passed domain using a Gaussian rule (Gauss-Legendre on [-1,1] as a default).

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the integration domain.
- **N** (*int, optional*) – Total number of sample points to use for the integration. Should be odd. Defaults to 3 points per dimension if None is given.
- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It also determines the numerical backend if possible.
- **backend** (*string, optional*) – Numerical backend. This argument is ignored if the backend can be inferred from integration_domain. Defaults to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Returns

Integral value

Return type
backend-specific number

CHAPTER

FOUR

ALL CONTENT

This is the list of all content in *torchquad*. The type *backend tensor* in the documentation is a placeholder for the tensor type of the current numerical backend, for example `numpy.array` or `torch.Tensor`.

We are continuously implementing new content in our library. For the code, please see the code page or check out our full code and latest news at https://github.com/esa/torchquad.

class torchquad.**BaseIntegrator**

Bases: `object`

The (abstract) integrator that all other integrators inherit from. Provides no explicit definitions for methods.

static evaluate_integrand(*fn*, *points*, *weights=None*, *args=None*)

Evaluate the integrand function at the passed points

Parameters

- **fn** (*function*) – Integrand function
- **points** (*backend tensor*) – Integration points
- **weights** (*backend tensor, optional*) – Integration weights. Defaults to None.
- **args** (*list or tuple, optional*) – Any arguments required by the function. Defaults to None.

Returns
Integrand function output int: Number of evaluated points

Return type
backend tensor

integrate()

class torchquad.**Boole**

Bases: *NewtonCotes*

Boole's rule. See https://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas#Closed_Newton%E2%80%93Cotes_formulas .

integrate(*fn*, *dim*, *N=None*, *integration_domain=None*, *backend=None*)

Integrates the passed function on the passed domain using Boole's rule.

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the integration domain.

- **N** (*int, optional*) – Total number of sample points to use for the integration. N has to be such that N^(1/dim) - 1 % 4 == 0. Defaults to 5 points per dimension if None is given.
- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It can also determine the numerical backend.
- **backend** (*string, optional*) – Numerical backend. Defaults to integration_domain's backend if it is a tensor and otherwise to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Returns
Integral value

Return type
backend-specific number

class torchquad.**GaussLegendre**

Bases: *Gaussian*

Gauss Legendre quadrature rule in torch for any domain [a,b]. See https://en.wikipedia.org/wiki/Gaussian_quadrature#Gauss%E2%80%93Legendre_quadrature.

Examples

```
>>> gl=torchquad.GaussLegendre()
>>> integral = gl.integrate(lambda x:np.sin(x), dim=1, N=101, integration_
↪domain=[[0,5]]) #integral from 0 to 5 of np.sin(x)
|TQ-INFO| Computed integral was 0.7163378000259399 #analytic result = 1-np.cos(5)
```

class torchquad.**Gaussian**

Bases: *GridIntegrator*

Base method for Gaussian Quadrature. Different Gaussian methods should inherit from this class, and override as necessary methods. Default behaviour is Gauss-Legendre quadrature on [-1,1] (i.e., this "parent" class should __not__ be used directly with other integration domains, and for this parent class *integration_domain* as an argument to *integrate* is ignored internally).

For an example of how to properly override the behavior to acheive different Gaussian Integration methods, please see the *Custom Integrators* section of the Tutorial or the implementation of *GaussLegendre*.

The primary methods/attributes of interest to override are *_root_fn* (for different polynomials, like *numpy.polynomial.legendre.leggauss*), *_apply_composite_rule* (as in other integration methods), and *_resize_roots* (for handling different integration domains).

name

A human-readable name for the integral.

Type
str

_root_fn

A function that returns roots and weights like *numpy.polynomial.legendre.leggauss*.

Type
function

_root_args

a way of adding information to be passed into *_root_fn* as needed. This is then used when caching roots/weights to potentially distinguish different calls to *_root_fn* based on arguments.

Type
tuple

_cache

a cache for roots and weights, used internally.

Type
dict

integrate(*fn*, *dim*, *N=8*, *integration_domain=None*, *backend=None*)

Integrates the passed function on the passed domain using a Gaussian rule (Gauss-Legendre on [-1,1] as a default).

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the integration domain.
- **N** (*int, optional*) – Total number of sample points to use for the integration. Should be odd. Defaults to 3 points per dimension if None is given.
- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It also determines the numerical backend if possible.
- **backend** (*string, optional*) – Numerical backend. This argument is ignored if the backend can be inferred from integration_domain. Defaults to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Returns
Integral value

Return type
backend-specific number

class torchquad.**GridIntegrator**

Bases: *BaseIntegrator*

The abstract integrator that grid-like integrators (Newton-Cotes and Gaussian) integrators inherit from

calculate_grid(*N*, *integration_domain*, *disable_integration_domain_check=False*)

Calculate grid points, widths and N per dim

Parameters

- **N** (*int*) – Number of points
- **integration_domain** (*backend tensor*) – Integration domain
- **disable_integration_domain_check** (*bool*) – Disbaling integration domain checks (default False)

Returns
Grid points backend tensor: Grid widths int: Number of grid slices per dimension

Return type
backend tensor

calculate_result(***kwargs*)

get_jit_compiled_integrate(*dim*, *N=None*, *integration_domain=None*, *backend=None*)

Create an integrate function where the performance-relevant steps except the integrand evaluation are JIT compiled. Use this method only if the integrand cannot be compiled. The compilation happens when the function is executed the first time. With PyTorch, return values of different integrands passed to the compiled function must all have the same format, e.g. precision.

Parameters

- **dim** (*int*) – Dimensionality of the integration domain.
- **N** (*int, optional*) – Total number of sample points to use for the integration. See the integrate method documentation for more details.
- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It can also determine the numerical backend.
- **backend** (*string, optional*) – Numerical backend. Defaults to integration_domain's backend if it is a tensor and otherwise to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Returns

JIT compiled integrate function where all parameters except the integrand and domain are fixed

Return type

function(fn, integration_domain)

integrate(*fn*, *dim*, *N*, *integration_domain*, *backend*)

Integrate the passed function on the passed domain using a Composite Newton Cotes rule. The argument meanings are explained in the sub-classes.

Returns

integral value

Return type

float

class torchquad.**IntegrationGrid**(*N*, *integration_domain*, *grid_func=<function grid_func>*, *disable_integration_domain_check=False*)

Bases: `object`

This class is used to store the integration grid for methods like Trapezoid or Simpsons, which require a grid.

h = None

points = None

class torchquad.**MonteCarlo**

Bases: *BaseIntegrator*

Monte Carlo integration

calculate_result(***kwargs*)

calculate_sample_points(*N*, *integration_domain*, *seed=None*, *rng=None*)

Calculate random points for the integrand evaluation

Parameters

- **N** (*int*) – Number of points
- **integration_domain** (*backend tensor*) – Integration domain

- **seed** (*int, optional*) – Random number generation seed for the sampling point creation, only set if provided. Defaults to None.
- **rng** (*RNG, optional*) – An initialised RNG; this can be used when compiling the function for Tensorflow

Returns
Sample points

Return type
backend tensor

get_jit_compiled_integrate(*dim*, *N=1000*, *integration_domain=None*, *seed=None*, *backend=None*)

Create an integrate function where the performance-relevant steps except the integrand evaluation are JIT compiled. Use this method only if the integrand cannot be compiled. The compilation happens when the function is executed the first time. With PyTorch, return values of different integrands passed to the compiled function must all have the same format, e.g. precision.

Parameters

- **dim** (*int*) – Dimensionality of the integration domain.
- **N** (*int, optional*) – Number of sample points to use for the integration. Defaults to 1000.
- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It can also determine the numerical backend.
- **seed** (*int, optional*) – Random number generation seed for the sequence of sampling point calculations, only set if provided. The returned integrate function calculates different points in each invocation with and without specified seed. Defaults to None.
- **backend** (*string, optional*) – Numerical backend. Defaults to integration_domain's backend if it is a tensor and otherwise to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Returns
JIT compiled integrate function where all parameters except the integrand and domain are fixed

Return type
function(fn, integration_domain)

integrate(*fn*, *dim*, *N=1000*, *integration_domain=None*, *seed=None*, *rng=None*, *backend=None*)

Integrates the passed function on the passed domain using vanilla Monte Carlo Integration.

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the function's domain over which to integrate.
- **N** (*int, optional*) – Number of sample points to use for the integration. Defaults to 1000.
- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It can also determine the numerical backend.
- **seed** (*int, optional*) – Random number generation seed to the sampling point creation, only set if provided. Defaults to None.

- **rng** (RNG, *optional*) – An initialised RNG; this can be used when compiling the function for Tensorflow
- **backend** (*string, optional*) – Numerical backend. Defaults to integration_domain's backend if it is a tensor and otherwise to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Raises
ValueError – If len(integration_domain) != dim

Returns
Integral value

Return type
backend-specific number

class torchquad.**RNG**(*backend*, *seed=None*, *torch_save_state=False*)

Bases: object

A random number generator helper class for multiple numerical backends

Notes

- The seed argument may behave differently in different versions of a numerical backend and when using GPU instead of CPU
  - https://pytorch.org/docs/stable/notes/randomness.html
  - https://numpy.org/doc/stable/reference/random/generator.html#numpy.random.Generator
  - https://www.tensorflow.org/api_docs/python/tf/random/Generator Only the Philox RNG guarantees consistent behaviour in Tensorflow.
- Often uniform random numbers are generated in [0, 1) instead of [0, 1].
  - numpy: random() is in [0, 1) and uniform() in [0, 1]
  - JAX: uniform() is in [0, 1)
  - torch: rand() is in [0, 1)
  - tensorflow: uniform() is in [0, 1)

jax_get_key()

Get the current PRNGKey. This function is needed for non-determinism when JIT-compiling with JAX.

jax_set_key(*key*)

Set the PRNGKey. This function is needed for non-determinism when JIT-compiling with JAX.

uniform(*size*, *dtype*)

Generate uniform random numbers in [0, 1) for the given numerical backend. This function is backend-specific; its definitions are in the constructor.

Parameters
- **size** (*list*) – The shape of the generated numbers tensor
- **dtype** (*backend dtype*) – The dtype for the numbers, e.g. torch.float32

Returns
A tensor with random values for the given numerical backend

Return type
backend tensor

class torchquad.**Simpson**

Bases: *NewtonCotes*

Simpson's rule. See https://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas#Closed_Newton%E2%80%93Cotes_formulas .

integrate(*fn*, *dim*, *N=None*, *integration_domain=None*, *backend=None*)

Integrates the passed function on the passed domain using Simpson's rule.

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the integration domain.
- **N** (*int, optional*) – Total number of sample points to use for the integration. Should be odd. Defaults to 3 points per dimension if None is given.
- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It can also determine the numerical backend.
- **backend** (*string, optional*) – Numerical backend. Defaults to integration_domain's backend if it is a tensor and otherwise to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Returns
Integral value

Return type
backend-specific number

class torchquad.**Trapezoid**

Bases: *NewtonCotes*

Trapezoidal rule. See https://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas#Closed_Newton%E2%80%93Cotes_formulas .

integrate(*fn*, *dim*, *N=1000*, *integration_domain=None*, *backend=None*)

Integrates the passed function on the passed domain using the trapezoid rule.

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the function to integrate.
- **N** (*int, optional*) – Total number of sample points to use for the integration. Defaults to 1000.
- **integration_domain** (*list or backend tensor, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim. It can also determine the numerical backend.
- **backend** (*string, optional*) – Numerical backend. Defaults to integration_domain's backend if it is a tensor and otherwise to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Returns
Integral value

Return type
backend-specific number

class torchquad.**VEGAS**

Bases: *BaseIntegrator*

VEGAS Enhanced. Refer to https://arxiv.org/abs/2009.05112 . Implementation inspired by https://github.com/ycwu1030/CIGAR/ . EQ <n> refers to equation <n> in the above paper. JAX and Tensorflow are unsupported. For Tensorflow there exists a VEGAS+ implementation called VegasFlow: https://github.com/N3PDF/vegasflow

integrate(*fn*, *dim*, *N=10000*, *integration_domain=None*, *seed=None*, *rng=None*, *use_grid_improve=True*, *eps_rel=0*, *eps_abs=0*, *max_iterations=20*, *use_warmup=True*, *backend=None*)

Integrates the passed function on the passed domain using VEGAS.

If the integrand output is far away from zero, i.e. lies within [b, b+c] for a constant b with large absolute value and small constant c, VEGAS does not adapt well to the integrand. Shifting the integrand so that it is close to zero may improve the accuracy of the calculated integral in this case. This method does not support multi-dimensional/vectorized integrands (i.e., integrating an integrand repeatedly over a grid of points).

Parameters

- **fn** (*func*) – The function to integrate over.
- **dim** (*int*) – Dimensionality of the function's domain over which to integrate.
- **N** (*int, optional*) – Approximate maximum number of function evaluations to use for the integration. This value can be exceeded if the vegas stratification distributes evaluations per hypercube very unevenly. Defaults to 10000.
- **integration_domain** (*list, optional*) – Integration domain, e.g. [[-1,1],[0,1]]. Defaults to [-1,1]^dim.
- **seed** (*int, optional*) – Random number generation seed for the sampling point creation; only set if provided. Defaults to None.
- **rng** (*RNG, optional*) – An initialised RNG; this can be used as alternative to the seed argument and to avoid problems with integrand functions which reset PyTorch's RNG seed.
- **use_grid_improve** (*bool, optional*) – If True, improve the vegas map after each iteration. Defaults to True.
- **eps_rel** (*float, optional*) – Relative error to abort at. Defaults to 0.
- **eps_abs** (*float, optional*) – Absolute error to abort at. Defaults to 0.
- **max_iterations** (*int, optional*) – Maximum number of vegas iterations to perform. The number of performed iterations is usually lower than this value because the number of sample points per iteration increases every fifth iteration. Defaults to 20.
- **use_warmup** (*bool, optional*) – If True, execute a warmup to initialize the vegas map. Defaults to True.
- **backend** (*string, optional*) – Numerical backend. "jax" and "tensorflow" are unsupported. Defaults to integration_domain's backend if it is a tensor and otherwise to the backend from the latest call to set_up_backend or "torch" for backwards compatibility.

Raises

ValueError – If the integration_domain or backend argument is invalid

Returns

Integral value

Return type

backend-specific float

torchquad.**enable_cuda**(*data_type='float32'*)

This function sets torch's default device to CUDA if possible. Call before declaring any variables! The default precision can be set here initially, or using set_precision later.

Parameters

data_type (*"float32", "float64" or None, optional*) – Data type to use. If None, skip the call to set_precision. Defaults to "float32".

torchquad.**set_log_level**(*log_level: str*)

Set the log level for the logger. The preset log level when initialising Torchquad is the value of the TORCHQUAD_LOG_LEVEL environment variable, or 'WARNING' if the environment variable is unset.

Parameters

log_level (*str*) – The log level to set. Options are 'TRACE','DEBUG', 'INFO', 'SUCCESS', 'WARNING', 'ERROR', 'CRITICAL'

torchquad.**set_precision**(*data_type='float32'*, *backend='torch'*)

Set the default precision for floating-point numbers for the given numerical backend. Call before declaring your variables.

NumPy and doesn't have global dtypes: https://github.com/numpy/numpy/issues/6860

Therefore, torchquad sets the dtype argument for these it when initialising the integration domain.

Parameters

- **data_type** (*str, optional*) – Data type to use, either "float32" or "float64". Defaults to "float32".
- **backend** (*str, optional*) – Numerical backend for which the data type is changed. Defaults to "torch".

torchquad.**set_up_backend**(*backend*, *data_type=None*, *torch_enable_cuda=True*)

Configure a numerical backend for torchquad.

With the torch backend, this function calls torchquad.enable_cuda unless torch_enable_cuda is False. With the tensorflow backend, this function enables tensorflow's numpy behaviour, which is a requirement for torchquad. If a data type is passed, set the default floating point precision with torchquad.set_precision.

Parameters

- **backend** (*string*) – Numerical backend, e.g. "torch"
- **data_type** (*"float32", "float64" or None, optional*) – Data type which is passed to set_precision. If None, do not call set_precision except if CUDA is enabled for torch. Defaults to None.
- **torch_enable_cuda** (*Bool, optional*) – If True and backend is "torch", call enable_cuda. Defaults to True.

class torchquad.integration.newton_cotes.**NewtonCotes**

Bases: *GridIntegrator*

The abstract integrator that Composite Newton Cotes integrators inherit from

class torchquad.integration.base_integrator.**BaseIntegrator**

Bases: object

The (abstract) integrator that all other integrators inherit from. Provides no explicit definitions for methods.

static evaluate_integrand(*fn*, *points*, *weights=None*, *args=None*)

Evaluate the integrand function at the passed points

Parameters

- **fn** (*function*) – Integrand function
- **points** (*backend tensor*) – Integration points
- **weights** (*backend tensor, optional*) – Integration weights. Defaults to None.
- **args** (*list or tuple, optional*) – Any arguments required by the function. Defaults to None.

Returns

Integrand function output int: Number of evaluated points

Return type

backend tensor

integrate()

CHAPTER

FIVE

CONTINUOUS INTEGRATION AND DEPLOYMENT

This document describes the continuous integration (CI) and continuous deployment (CD) setup for torchquad, which ensures code quality, testing, and automated releases.

5.1 Overview

Torchquad uses GitHub Actions for CI/CD with the following key objectives:

- **Automated Testing**: Run comprehensive test suites on every code change
- **Code Quality**: Enforce consistent formatting and linting standards
- **Multi-Backend Support**: Test across PyTorch, JAX, TensorFlow, and NumPy
- **Automated Deployment**: Streamlined releases to PyPI and Test PyPI
- **Documentation**: Automated paper builds for JOSS submissions

5.2 GitHub Actions Workflows

The CI/CD pipeline consists of five main workflows:

1. **Test Suite** (`run_tests.yml`)

   Triggers: Push to main/develop branches, pull requests, manual dispatch

   This is the core testing workflow that runs on every code change:

   - **Linting Stage**: Uses flake8 to check code quality and style
   - **Testing Stage**: - Sets up Python 3.9 environment - Installs all backend dependencies via micromamba - Runs full pytest suite with coverage reporting - Posts coverage reports as PR comments

   Key Features:

   - Multi-backend testing (all numerical backends)
   - Coverage tracking with pytest-cov
   - JUnit XML output for CI integration
   - Automated PR comments with test results

2. **Code Formatting** (`autoblack.yml`)

   Triggers: Pull requests only

   Ensures consistent code formatting across the project:

   - Uses Black formatter with 100-character line length

- Python 3.11 environment
- Checks formatting without modifying files
- Fails if reformatting is needed

3. **PyPI Deployment** (`deploy_to_pypi.yml`)

   Triggers: Manual workflow dispatch only

   Production deployment to PyPI:

   - Python 3.10 environment
   - Builds source distribution and wheel packages
   - Uploads to PyPI using stored authentication token
   - Manual trigger ensures controlled releases

4. **Test PyPI Deployment** (`deploy_to_test_pypi.yml`)

   Triggers: Manual workflow dispatch, GitHub releases

   Test deployment for validation:

   - Same process as PyPI deployment
   - Targets Test PyPI for safe testing
   - Used to validate packages before production release

5. **Documentation** (`draft-pdf.yml`)

   Triggers: Changes to paper directory

   Builds academic paper PDF:

   - Uses OpenJournals GitHub Action
   - Compiles Markdown to PDF for JOSS submissions
   - Stores generated PDF as workflow artifact

5.3 Environment Setup

The CI system uses conda/micromamba for dependency management:

```yaml
# From run_tests.yml
- name: provision-with-micromamba
  uses: mamba-org/setup-micromamba@v1
  with:
    environment-file: environment_all_backends.yml
    environment-name: torchquad
    cache-downloads: true
```

5.3.1 Environment Files

- `environment.yml` - Basic PyTorch setup for development
- `environment_all_backends.yml` - Complete backend support for CI
- `rtd_environment.yml` - ReadTheDocs documentation builds

5.4 Test Execution

The test suite runs with comprehensive coverage:

```
cd tests/
pytest -ra --error-for-skips \\
       --junitxml=pytest.xml \\
       --cov-report=term-missing:skip-covered \\
       --cov=../torchquad . | tee pytest-coverage.txt
```

Test Parameters:

- `-ra` - Show summary for all test outcomes
- `--error-for-skips` - Treat skipped tests as errors (fail CI)
- `--junitxml` - Generate XML report for CI integration
- `--cov` - Generate coverage report for the torchquad package

5.5 Code Quality Standards

5.5.1 Linting with Flake8

Two-stage linting process:

1. **Critical Errors**: Check for syntax errors and undefined names

   ```
   flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
   ```

2. **Full Analysis**: Complete code quality check using project `.flake8` configuration

   ```
   flake8 . --count --show-source --statistics
   ```

5.5.2 Formatting with Black

Consistent code style enforcement:

```
black --check --line-length 100 .
```

Configuration:

- Line length: 100 characters
- Target: Python 3.11+
- Complies with project style guide

5.6 Coverage Reporting

The CI system provides detailed coverage analysis:

- **PR Comments**: Automated coverage reports on pull requests
- **Trend Tracking**: Coverage change detection
- **Missing Lines**: Identification of untested code
- **Badge Integration**: Coverage badges for README

Coverage Requirements:

- New features must include comprehensive tests
- Significant coverage decreases block PR merges
- Target: >90% coverage for new code

5.7 Local Development

Before pushing changes, run these checks locally:

```
# Format code
black . --line-length 100

# Check linting
flake8 . --count --show-source --statistics

# Run tests
cd tests/
pytest

# Run with coverage
pytest --cov=../torchquad
```

5.7.1 Environment Setup

For local development:

```
# Create environment
conda env create -f environment_all_backends.yml
conda activate torchquad

# Install in development mode
pip install -e .
```

5.8 Backend Testing

5.8.1 Multi-Backend Strategy

Tests run across all supported numerical backends:

- **NumPy**: Reference implementation and baseline testing
- **PyTorch**: GPU acceleration and automatic differentiation
- **JAX**: JIT compilation and XLA optimization
- **TensorFlow**: Graph execution and TPU support

Backend-Specific Considerations:

- Some tests are backend-specific and use appropriate skip decorators
- GPU tests run automatically when CUDA is available
- Complex number support varies by backend
- Performance characteristics differ between backends

5.9 Release Process

5.9.1 PyPI Deployment

Production releases follow this process:

1. **Code Review**: All changes go through PR review
2. **Testing**: Full test suite must pass
3. **Version Update**: Update version in `pyproject.toml`
4. **Test Deployment**: Deploy to Test PyPI first
5. **Validation**: Test installation from Test PyPI
6. **Production**: Manual trigger of PyPI deployment workflow

Required Secrets:

- `PYPI_TOKEN` - PyPI API token for package uploads
- `TEST_PYPI_TOKEN` - Test PyPI API token

5.10 Security Considerations

- **Token Management**: API tokens stored as GitHub secrets
- **Manual Triggers**: Production deployments require manual approval
- **Branch Protection**: Main branch protected with required status checks
- **Dependency Scanning**: Automated security updates via Dependabot

5.11 Troubleshooting

5.11.1 Common CI Failures

1. **Formatting Issues**:

```
# Fix locally
black . --line-length 100
git add . && git commit -m "Fix formatting"
```

2. **Import Errors**:
   - Check dependency versions in environment files
   - Verify relative imports after package structure changes
   - Ensure test files are properly isolated
3. **Backend-Specific Failures**:
   - Check if backend is properly installed in CI environment
   - Verify skip decorators for unavailable backends
   - Review backend-specific test logic
4. **Coverage Decreases**:
   - Add tests for new functionality

- Check test discovery (files must match `*_test.py` or `test_*.py`)
- Verify coverage configuration in `pyproject.toml`

5. **Environment Issues**:
   - Update `environment_all_backends.yml` for new dependencies
   - Check for version conflicts between backends
   - Verify micromamba cache invalidation

5.12 Building Documentation Locally

To build the Sphinx documentation locally:

```
# Navigate to docs directory
cd docs

# Build HTML documentation
make html

# On Windows, you can also use:
make.bat html

# Clean build directory
make clean

# View all available targets
make help
```

The built documentation will be available in `docs/_build/html/`. Open `docs/_build/html/index.html` in your browser to view the documentation.

Note: Make sure you have Sphinx and all documentation dependencies installed:

```
pip install sphinx sphinx-rtd-theme
```

5.13 Getting Help

For CI/CD issues:

1. Check the GitHub Actions page for detailed logs
2. Review similar successful runs for comparison
3. Check environment file consistency
4. Verify all required secrets are configured
5. Open an issue with CI logs if problems persist

The CI/CD system is designed to catch issues early and ensure high code quality. When in doubt, run the same commands locally that CI runs to debug issues quickly.

CHAPTER

SIX

CONTACT INFORMATION

Created by ESA's Advanced Concepts Team:

- Pablo Gómez - *pablo.gomez (at) esa.int*
- Gabriele Meoni - *gabriele.meoni (at) esa.int*
- Håvard Hem Toftevaag - *havard.hem.toftevaag (at) esa.int*

Project Link: https://github.com/esa/torchquad.

6.1 Feedback

If you want to get in touch with the creators of torchquad, please send an email to *pablo.gomez (at) esa.int*.

CHAPTER

SEVEN

CONTRIBUTING

The project is open to community contributions. Feel free to open an issue or write us an email if you would like to discuss a problem or idea first.

If you want to contribute, please

1. Fork the project on GitHub.
2. Get the most up-to-date code by following this quick guide for installing *torchquad* from source:

a. Get miniconda or similar

b. Clone the repo

```
git clone https://github.com/esa/torchquad.git
```

c. With the default configuration, all numerical backends with CUDA support are installed. If this should not happen, comment out unwanted packages in `environment.yml`.

d. Set up the environment. This creates a conda environment called `torchquad` and installs the required dependencies.

```
conda env create -f environment.yml
conda activate torchquad
```

Once the installation is done, then you are ready to contribute. Please note that PRs should be created from and into the `develop` branch. For each release the develop branch is merged into main.

3. Create your Feature Branch: `git checkout -b feature/AmazingFeature`
4. Commit your Changes: `git commit -m 'Add some AmazingFeature'`
5. Push to the Branch: `git push origin feature/AmazingFeature`
6. Open a Pull Request on the `develop` branch, *not* `main` (NB: We autoformat every PR with black. Our GitHub actions may create additional commits on your PR for that reason.)

and we will have a look at your contribution as soon as we can.

Furthermore, please make sure that your PR passes all automated tests. Review will only happen after that. Only PRs created on the `develop` branch with all tests passing will be considered. The only exception to this rule is if you want to update the documentation in relation to the current release on `conda` / `pip`. In that case you may ask to merge directly into `main`.

CHAPTER

EIGHT

ROADMAP

See the open issues for a list of proposed features (and known issues).

CHAPTER

NINE

LICENSE

Distributed under the GPL-3.0 License. See LICENSE for more information.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

B

C

E

G

H

I

J

M

N

P

R

S

T

U

V