The Journal of Open Source Software

DistributedSparseGrids.jl: A Julia library implementing an Adaptive Sparse Grid collocation method

Maximilian Bittens [1] **and Robert L. Gates**[2]

1 Federal Institute for Geosciences and Natural Resources (BGR), Germany **2** Independent Researcher, Germany

DOI: 10.21105/joss.05003

Software

- Review
- Repository
- Archive

Editor: Vincent Knight

Reviewers:

- @ericneiva
- @matt-graham

Submitted: 22 November 2022
Published: 07 March 2023

Abstract

Numerical integration or interpolation of high-dimensional functions is subject to the curse of dimensionality on full tensor grids. One remedy to this problem is sparse grid approximations. The additional construction effort is often worth spending, especially for underlying functions whose evaluation is time-consuming. In the following, a Julia implementation of a local Lagrangian adaptive hierarchical sparse grid collocation method is presented, which is suitable for memory-heavy objects generated on distributed workers.

Statement of need

DistributedSparseGrids.jl is a Julia package for integrating and interpolating functions with generic return types. There are other approaches to sparse grid approximation written in the Julia language, such as SparseGrids.jl, AdaptiveSparseGrids.jl, GalerkinSparseGrids.jl, and Tasmanian.jl. However, there is no Julia package available at the moment that is suitable if the solution of the underlying (discretized) physical problem is time and resource-consuming, requiring it to be solved on either a server or cluster environment, or the solution is memory-heavy, like a Vector, Matrix, or, for example, a complete finite element solution.

Introduction

Sparse tensor product quadrature rules, mitigating the curse of dimensionality occurring in full tensor grid constructions, were provided first by Smolyak (1963). In the last two decades, collocation methods have been prominent in solving stochastic partial differential equations, as shown in Babuška et al. (2007) and Nobile et al. (2008). Ma & Zabaras (2009) were able to once again increase the efficiency of the collocation approach by introducing an error-adaptive formulation of the method, which will serve as a basis for the collocation method described in this project. For more information about the theory of the method implemented, see, e.g., Gates & Bittens (2015).

Features

In the following, some key features of the implemented approach are listed.

Arbitrary return types

DistributedSparseGrids.jl defines a `HierarchicalCollocationPoint{N,CP,RT}`, where `N` is the number of dimensions, `CP <: AbstractCollocationPoint{N,CT<:Real}`, and `RT` is a generic return type. `RT` can be conveniently defined as the type most suitable for studying the problem

at hand, such as a `Float64`, a `Vector{Float64}`, or a `Matrix{Float64}`, for example. Suppose the underlying physical problem stores its data in the VTU file format (Schroeder et al., 2000). In that case, the Julia project VTUFileHandler.jl (Bittens, 2022) can be used; it implements all operators needed to use complete result files with the sparse grid.

In-place operations

Computing the weights for the hierarchical basis as well as performing interpolation and integration relies heavily on the use of *arithmetic operators*, which allocate memory. This can be a problem, especially if the result type is memory heavy. Therefore, DistributedSparseGrids.jl defines in-place variants to all of these actions given in-place variants for the arithmetic operators are defined. For further information, see the documentation.

Distributed computing

If the function's runtime to be evaluated is long, it may be necessary to distribute the load to several workers. Julia provides this functionality *out-of-the-box* via the `Distributed` interface. Due to the hierarchical construction of the basis and the level-wise adaptive refinement indicator, it seems necessary to include this interface in the sparse grid for a performant application of distributed computing. DistributedSparseGrids.jl uses all workers included by the `Distributed.addprocs` command if the `distributed_init_weights!` function is used to determine the hierarchical weights.

Additional features

- Nested one-dimensional Clenshaw-Curtis rule
- Smolyak's sparse grid construction
- Local hierarchical Lagrangian basis
- Different pointsets (open, closed, halfopen)
- Adaptive refinement
- Multi-threaded calculation of basis coefficients with `Threads.@threads`
- Integration
- Experimental: integration over $X_{\sim(i)}$ (the $X_{\sim(i)}$ notation indicates the set of all variables except X_i).

Example

Below, an example of an adaptive sampling of a function with a curved singularity in 2D is provided. Figure 1 shows an illustration of the sparse grid approximation.

```
using DistributedSparseGrids
using Distributed
using StaticArrays
import PlotlyJS

function sparse_grid(N::Int,pointprobs,nlevel=6,RT=Float64,CT=Float64)
  # define collocation point
  CPType = CollocationPoint{N,CT}
  # define hierarchical collocation point
  HCPType = HierarchicalCollocationPoint{N,CPType,RT}
  # init grid
  asg = init(AHSG{N,HCPType},pointprobs)
  # set of all collocation points
  cpts = Set{HierarchicalCollocationPoint{N,CPType,RT}}(collect(asg))
  # fully refine grid nlevel-1 times
```

```
  for i = 1:nlevel-1
    union!(cpts,generate_next_level!(asg))
  end
  return asg
end

# Sparse Grid with 4 initial levels
pp = @SVector [1,1]
asg = sparse_grid(2, pp, 4)

# add 2 worker
ar_worker = addprocs(2)

@everywhere begin
  using StaticArrays
  # Function with curved singularity
  fun1(x::SVector{2,Float64},ID::String) =
    (1.0-exp(-1.0*(abs(2.0 - (x[1]-1.0)^2.0 -
      (x[2]-1.0)^2.0) +0.01)))/(abs(2-(x[1]-1.0)^2.0-(x[2]-1.0)^2.0)+0.01)
end

# calculate weights on master
init_weights!(asg, fun1)

# adaptive refine
for i = 1:20
  # call generate_next_level! with tol=1e-5 and maxlevels=20
  cpts = generate_next_level!(asg, 1e-5, 20)
  # calculate weights on all worker
  distributed_init_weights!(asg, collect(cpts), fun1, ar_worker)
end

# plot
surfplot = PlotlyJS.surface(asg, 100)
gridplot = PlotlyJS.scatter3d(asg)
PlotlyJS.plot([surfplot, gridplot])
```

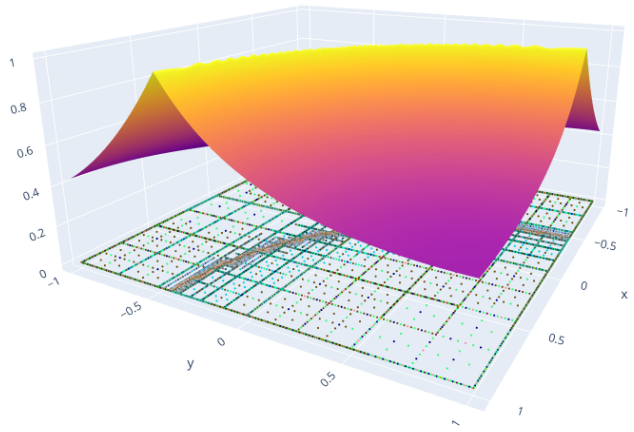

Figure 1: Refined sparse grid.

References

Babuška, I., Nobile, F., & Tempone, R. (2007). A stochastic collocation method for elliptic partial differential equations with random input data. *SIAM Journal on Numerical Analysis*, *45*(3), 1005–1034. https://doi.org/10.1137/100786356

Bittens, M. (2022). VTUFileHandler: A VTU library in the Julia language that implements an algebra for basic mathematical operations on VTU data. *Journal of Open Source Software*, *7*(73), 4300. https://doi.org/10.21105/joss.04300

Gates, R. L., & Bittens, M. R. (2015). A multilevel adaptive sparse grid stochastic collocation approach to the non-smooth forward propagation of uncertainty in discretized problems. *arXiv Preprint arXiv:1509.01462*. https://doi.org/10.48550/arXiv.1509.01462

Ma, X., & Zabaras, N. (2009). An adaptive hierarchical sparse grid collocation algorithm for the solution of stochastic differential equations. *Journal of Computational Physics*, *228*(8), 3084–3113. https://doi.org/10.1016/j.jcp.2009.01.006

Nobile, F., Tempone, R., & Webster, C. G. (2008). A sparse grid stochastic collocation method for partial differential equations with random input data. *SIAM Journal on Numerical Analysis*, *46*(5), 2309–2345. https://doi.org/10.1137/060663660

Schroeder, W. J., Avila, L. S., & Hoffman, W. (2000). Visualizing with VTK: A tutorial. *IEEE Computer Graphics and Applications*, *20*(5), 20–27. https://doi.org/10.1109/38.865875

Smolyak, S. A. (1963). Quadrature and interpolation formulas for tensor products of certain classes of functions. *Doklady Akademii Nauk*, *148*, 1042–1045.