SoftwareX 12 (2020) 100541

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

PyModPDE: A python software for modified equation analysis

Mokbel Karam, James C. Sutherland, Tony Saad *

Department of Chemical Engineering, University of Utah, Salt Lake City, Utah, USA

ARTICLE INFO

Article history:
Received 5 February 2020
Received in revised form 11 May 2020
Accepted 9 June 2020

Keywords:
Modified equation analysis
Finite difference schemes
Stability analysis
Numerical methods
Error analysis

ABSTRACT

The modified equation is a useful tool in the analysis of numerical methods for partial differential equations (PDEs). It gives insight into the stability, diffusion, and dispersion properties of a given numerical scheme. Its derivation, however, is rather tedious and error-prone due to the enormous amount of algebra involved. PyModPDE is a python software that uses a novel approach to generate the modified equation. It takes a discrete PDE as its input and outputs the modified equation in LaTeX format. We discuss the novel approach on which PyModPDE is based and then validate the software using one and two-dimensional PDEs. PyModPDE serves as an essential tool for computational scientists and engineers for both educational and research purposes.

Code metadata

Current code version	v1.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2020_22
Code Ocean compute capsule	NA
Legal Code License	MIT License
Code versioning system used	git
Software code languages, tools, and services used	Python
Compilation requirements, operating environments & dependencies	Standard Python 3 installation with sympy
If available Link to developer documentation/manual	https://github.com/saadgroup/pymodpde
Support email for questions	tony.saad@utah.edu

1. Motivation and significance

The modified equation is a valuable tool in the analysis of finite difference approximations to time-dependent partial differential equations (PDEs) because of the insight it provides into the effects of discretization. Although applicable to both linear and nonlinear PDEs [1–4], its use on linear differential equations has permeated both introductory and advanced numerical analysis. This is no surprise since the modified equation provides qualitative and quantitative information about the behavior and stability properties of a given finite difference scheme. The idea of the modified equation was first introduced by Warming and Hyett [5] in a landmark paper in 1974. Their argument was simple: because of the truncation error, a finite difference scheme for a given differential equation does not solve that equation; rather, it solves an entirely different PDE. Their approach consists of replacing each term in the finite difference formula by its Taylor series expansion and then expressing the high order time derivatives in terms of spatial derivatives, as dictated by the governing equation. The terms in the modified equation can then be used to explain the dissipative and dispersive behavior of the finite difference scheme used (although this is not very clear in the case of nonlinear PDEs).

Despite its mechanistic nature, deriving the modified equation is rather tedious and lends itself to user and programming errors due to the enormous amount of algebra involved. In 1990, Chang [6] presented the first[1] alternative to obtaining the modified equation in a simple and more systematic way that is amenable to easy implementation in a symbolic code. His method applies to linear time-dependent PDEs and builds

* Corresponding author.
E-mail addresses: mokbel.karam@chemeng.utah.edu (M. Karam), james.sutherland@utah.edu (J.C. Sutherland), tony.saad@utah.edu (T. Saad).

[1] To the best of the authors' knowledge.

https://doi.org/10.1016/j.softx.2020.100541

on the fact that the amplification factor derived from a Von-Neumann stability analysis is directly related to the coefficients of the modified equation. It is not clear why Chang's formulation has gone unnoticed. In a similar fashion, Ramshaw [7] independently used the amplification factor to estimate numerical diffusion. His method consists of taking the first and second derivative of the amplification factor with respect to the wave number k, evaluated at $k = 0$. Another approach to obtaining the coefficients of the modified equation using the amplification factor was proposed by Carpentier [8] but can be shown to be generally similar to that proposed by Chang [6].

In this article, we argue that Chang's approach should be the method of choice for conducting a modified equation analysis and then discuss how it is amenable to straightforward implementation in a software through `PyModPDE`. We first present Chang's work in a concise form and then generalize it to two and three spatial dimensions. We then discuss our software implementation of the method and then give illustrative examples to verify the software.

1.1. Proposed framework

For a given linear PDE of the form $u_t + \mathcal{L}(u_x, u_{xx}, \ldots) = 0$ with constant coefficients and a corresponding difference scheme

$$\delta_t u + \gamma_1 \delta_x u + \gamma_2 \delta_{xx} u + \cdots = 0, \tag{1}$$

the modified equation takes the form

$$u_t = a_1 u_x + a_2 u_{xx} + a_3 u_{xxx} + \cdots = \sum_{n=1}^{\infty} a_n \frac{\partial^n u}{\partial x^n} \tag{2}$$

where a_i are not, in general, equal to γ_i. In essence, the modified equation accounts for the truncation error introduced by a numerical scheme. This equation represents the physics that the difference scheme actually represents. Odd derivatives represent advection and dispersion while even derivatives correspond to diffusion. Based on the coefficients a_i, one can then make rational arguments about the stability of the finite difference method and explain the presence of numerical diffusion and dispersion in a particular numerical scheme.

The coefficients of the modified equation have been traditionally obtained by substituting the Taylor series of each term in the finite difference formula. High-order time derivatives are then replaced with spatial derivatives via substitution of the original PDE. An alternative approach [6,7] is based on the fact that the amplification factor is directly related to the coefficients of the modified equation. The amplification factor is obtained from a von-Neumann stability analysis and designates the growth rate of an error introduced into the finite difference representation of a PDE as given by Eq. (1). The von-Neumann stability method proceeds by analyzing what happens to a solution of the form $u = \sum_k e^{\alpha t} e^{\mathrm{i}kx}$, subject to Eq. (1). If the solution grows unbounded, then the scheme is unstable, otherwise, it is stable. In this formulation, the amplification factor is then defined as the relative growth of the solution over a single timestep, $G \equiv \frac{u(t+\Delta t)}{u(t)} = e^{\alpha \Delta t}$, and dictates the amplification/decay of the solution over time. Obviously, given that $\Delta t > 0$, the amplification exponent, α, determines whether the solution grows ($\alpha > 0$, unstable) or decays ($\alpha < 0$, stable) over time. Finally, given that the PDE is linear, it is sufficient to study the amplification of a single frequency $u_k = e^{\alpha t} e^{\mathrm{i}kx}$ thanks to superposition. According to [5], upon substitution of u_k into Eq. (2), one can relate the amplification exponent, α, to the coefficients of the modified equation via

$$\alpha = \sum_{n=1}^{\infty} \mathrm{i}^n a_n k^n. \tag{3}$$

The coefficients (a_n) of the modified equation can be obtained by differentiating Eq. (3) with respect to k^n at $k = 0$

$$a_n = \frac{1}{\mathrm{i}^n n!} \left. \frac{\partial^n \alpha}{\partial k^n} \right|_{k=0}. \tag{4}$$

This suggests that, if α is a known quantity, then the coefficients a_n of the modified equation can be computed by direct differentiation of α.

We argue that, rather than deriving the modified equation by using Taylor series and repetitive differentiation and substitutions, one can instead do the following:

1. substitute $u = e^{\alpha t} e^{\mathrm{i}kx}$ into the difference scheme in Eq. (1),
2. solve for α from Eq. (1),
3. find the coefficients a_n of the modified equation(s) using Eq. (4).

This approach is straightforward and is amenable to simple symbolic manipulation in a symbolic package such as `Sympy`.

1.2. Generalization to higher dimensions

A similar analysis can be applied in higher dimensions. In that case we propose the following form of the modified equation

$$\begin{aligned} u_t &= a_{100} u_x + a_{010} u_y + a_{001} u_z + a_{110} u_{xy} + a_{101} u_{xz} + a_{011} u_{yz} + \cdots \\ &= \sum_{\substack{j,l,m \geq 0 \\ j+l+m \geq 1}}^{\infty} a_{jlm} \frac{\partial^{j+l+m} u}{\partial x^j \partial y^l \partial z^m}. \end{aligned} \tag{5}$$

Note the presence of cross derivative terms, *e.g.* u_{xy}. These terms are generally *anisotropic* in nature since they result in different physical behavior in different directions (an example on this is given in Section 3.2). Assuming an elementary solution of the form

$$u_{\mathbf{k}} = e^{\alpha t} e^{\mathrm{i}(k_x x + k_y y + k_z z)} = e^{\alpha t} e^{\mathrm{i}\mathbf{k}\cdot\mathbf{x}}, \tag{6}$$

we substitute into Eq. (5) to recover the following relation between the amplification exponent and the modified equation

$$\alpha = \sum_{\substack{j,l,m \geq 0 \\ j+l+m \geq 1}}^{\infty} a_{jlm} \mathrm{i}^{j+l+m} k_x^j k_y^l k_z^m. \tag{7}$$

The coefficients a_{jlm} are obtained by via

$$a_{jlm} = \frac{1}{\mathrm{i}^n j!l!m!} \left. \frac{\partial^n \alpha}{\partial k_x^j \partial k_y^l \partial k_z^m} \right|_{\mathbf{k}=\mathbf{0}} \tag{8}$$

where $n = j + l + m$.

2. Software description

`PyModPDE` is written in python due to (1) its popularity amongst researchers and students, (2) its ease of use and quick turnaround development time, and (3) the availability of robust symbolic libraries, specifically `Sympy`. The `PyModPDE` software offers a single interface to a first order (in time) time dependent PDE class that allows users to construct its right-hand-side (RHS), and to generate the modified equation representation in LaTeX form.

2.1. Software architecture

The abstraction followed by PyModPDE is based on interpreting a PDE as consisting of a left-hand-side (LHS) and a right-hand-side (RHS). The LHS always consists of a first-order time derivative of the dependent variable, i.e. ϕ_t. On the other hand, the RHS consists of an arbitrary number of spatial terms with arbitrary order such as $a\phi_x$ or $\mu\phi_{xx}$. To construct a discrete equivalent of a PDE one must then provide a discretization scheme for each term on the RHS. The discretization of the LHS is "baked" into the software as $\frac{1}{\Delta t}(\phi^{n+1} - \phi^n)$. Higher order discretizations are absolutely possible and will be part of future releases of the code.

Based on this, PyModPDE defines a class for first order linear PDEs where the LHS is fixed to being a first-order time derivative. This class allow users to instantiate a differential equation with user-specified names for the dependent variable and independent variable(s). The user then proceeds by creating terms to add to the RHS. When creating a term, the user provides the order of the spatial derivative, the direction (x, y, z), the time level at which the discretization is supposed to take place (n for explicit or $n+1$ for implicit) and the type of discretization (discussed later). Finally, a simple call to generate the modified equation is made and a LaTeX formatted equation becomes available. A graphical description of PyModPDE functionalities is shown in Fig. 1.

2.2. Software functionalities and sample code snippets

Using PyModPDE is straightforward and requires four steps: (1) Instantiate a `DifferentialEquation` class, (2) build RHS terms, (3) set the rhs of the PDE using expressions generated in step (2), and finally (4) generate the modified equation or the amplification factor. To discuss the API, it is best to start with a concrete example, where all the source code will be detailed after. Consider obtaining the modified equation for the advection equation, $u_t = -au_x$ in one-dimensional space with a forward Euler in time, UPWIND in space (FTUS scheme) discretization. The discrete equation at an arbitrary spatial grid point with index i - not to be confused with the imaginary unit i - is given by the formula

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = -a\frac{u_i^n - u_{i-1}^n}{\Delta x}. \tag{9}$$

In PyModPDE this can be implemented as follows:

```
from src.pymodpde import DifferentialEquation
from sympy import symbols

# declare the advection velocity
a = symbols("a")
# declare the spatial and temporal indices
i, n = symbols("i n")

# construct a time dependent differential equation
DE = DifferentialEquation(dependentVarName="u",
    independentVarsNames = ["x"], indices=[i],
    timeIndex=n)

# method I of constructing the rhs (method II will be
    discussed later in this section):
advectionTerm = -a * DE.expr(order=1, directionName="x
    ", time=n,stencil=[-1, 0])

# setting the rhs of the differential equation
DE.set_rhs( advectionTerm )

# generate the modified equation up to two terms
DE.generate_modified_equation(nterms=2)
```

This example illustrates the four steps we referred to earlier. By design, all coefficients of the PDE and spatial and time indices will be defined as `sympy` symbols while other properties of the PDE such as the name of dependent and independent variables are passed as strings. After declaring all symbols that will be used (a, i, and n), one instantiates a `DifferentialEquation` object which represents a first order time dependent differential equation. This class has the following constructor signature

```
DifferentialEquation(dependentVarName,
    independentVarsNames,indices=[i, j, k], timeIndex
    =n)
```

where

- `dependentVarName` is a required string for the name of the dependent variable.
- `independentVarsNames` is a required list of strings for the names of the independent variables.
- `indicies` is a list of `sympy` symbols (e.g. `[i, j ,k]`) which correspond to the indexing used to refer to different grid locations, e.g. $u_{i,j}$. These symbols are matched to the independent variables in the order they are specified.
- `timeIndex` is a `sympy` symbol for the time index, e.g. `n`. Note that the symbols used in the `indices` and `timeIndex` arguments must be declared beforehand.

As an example of constructing a `DifferentialEquation` for a two-dimensional quantity $u = u(t, x, y)$. We will use i and j as spatial indices that correspond to x and y, respectively, and n as the time index.

```
i, j , n = symbols('i j n') # must declare all symbols
     first
DE = DifferentialEquation(dependentVarName='u',
    independentVarsNames=['x','y'], indices=[i, j],
    timeIndex=n)
```

Once a `DifferentialEquation` object has been instantiated, the next step is to start building the terms in the RHS for this PDE. Two methods are available to achieve this.

Starting with the first method, one can use the member function `expr`. This convenient function automatically generates a finite difference approximation based on Taylor series expansions for the nth order spatial derivative in a particular direction and returns its discretized symbolic expression. The function `expr` has the following signature

```
expr(order, directionName, time, stencil)
```

where

- `order` is a required integer denoting the spatial order of the derivative.
- `directionName` is a required string for the name of the independent variable with respect to which we are differentiating.
- `time` is a required Sympy symbol for the time level at which we are evaluating this expression, e.g. `n+1, n, \ldots`.
- `stencil` is a required list of integers for the N points used in the finite difference representation with a reference index 0 at grid point i.

Note that `order` has to be less than N for the finite difference representation to exist. For example, assume you want to add an advection term in the x-direction for $u(t, x, y)$ using the backward difference formula

$$\left.\frac{\partial u}{\partial x}\right|_{t=t_n} = \frac{u_{i,j}^n - u_{i-1,j}^n}{\Delta x}. \tag{10}$$

To implement this using `expr`, one sets `order = 1` (first derivative), `directionName = 'x'` (x-derivative), `time = n` (explicit integration), and `stencil = [-1, 0]` indicating a backward difference formula

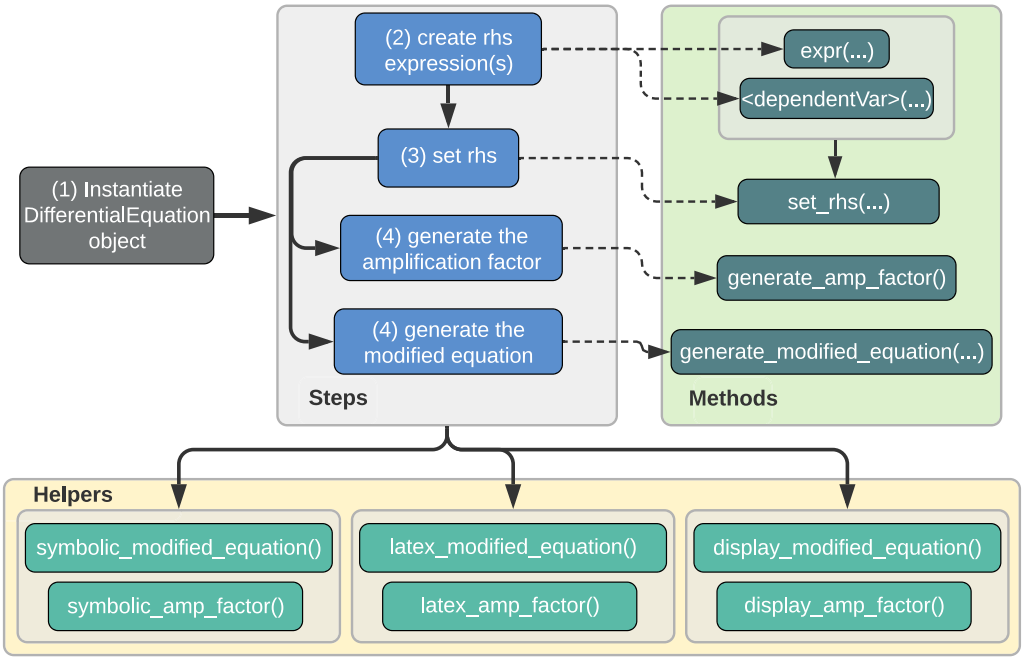

Fig. 1. Steps to generate the modified equation using PyModPDE.

```
term1 = expr(order=1, directionName='x', time=n,
    stencil = [-1,0])
```

Recall that `directionName` only takes one string value as an argument, which means that cross derivative terms such as $\frac{\partial^2 u}{\partial x \partial y}$ cannot be represented using this member function. Also, time levels for a given term cannot be mixed, meaning that all the elements in the discrete representation of the expression **must** be evaluated at the same `timeIndex`. This means that one cannot use something like $(u_i^{n+1} - u_{i-1}^n)$ with this approach.

The second method to build terms for the RHS provides more user control over the discretization and enables one to construct arbitrarily complex finite difference schemes with arbitrary stencils in time and space. It effectively mimics the way we write a finite difference formula by hand. This is done by using the member function `<dependentVarName>(...)` which has the following signature

```
<dependentVarName>(time, **kwargs)
```

where `time` is the discrete time level at which the dependent variable is evaluated e.g. `n+1`, `n`, `\ldots` and `kwargs` are indices of the spatial points at which this expression is evaluated, e.g. `x=i+1, y=j, \ldots` .

As an example of how to use this method consider the discretization given in Eq. (10). Then the finite difference representation will translate into the following

```
(DE.u(time=n, x=i, y=j) - DE.u(time=n, x=i-1, y=j))/DE
    .dx
```

Unlike the first approach that uses `expr`, this method can be used to represent cross derivative terms. As an example consider the following second order cross derivative term evaluated at time $t = t_n$

$$\left.\frac{\partial^2 u}{\partial x \partial y}\right|_{t=t_n} = \frac{u_{i,j}^n - u_{i-1,j}^n - u_{i,j-1}^n + u_{i-1,j-1}^n}{\Delta x \Delta y}. \tag{11}$$

We can code this as follows

```
term2 = (DE.u(time=n, x=i, y=j) - DE.u(time=n, x=i-1,
    y=j) - DE.u(time=n, x=i, y=j-1) + DE.u(time=n, x=
    i-1, y=j-1))/(DE.dx * DE.dy)
```

After building terms for the RHS expression using either one of the two previous methods or a combination of both, the user can set the RHS for the differential equation by calling the member function `set_rhs`

```
set_rhs(expression)
```

where `expression` is a linear combination of terms just constructed, i.e.

```
set_rhs(term1 + term2)
```

At this point, the setup is complete and a modified equation can be generated by calling the member function `generate_modified_equation`

```
generate_modified_equation(nterms)
```

where `nterms` is a positive integer that indicates the total number of terms to return in the modified equation.

The call for the function `generate_modified_equation(...)` changes the state of the `DifferentialEquation` object and allows for the use of helper functions to access various forms of the modified equation and the amplification factor. For example, one can call `display_modified_equation()` to display a rendered LaTeX representation of the modified equation. For this example

```
DE.display_modified_equation()
```

will display the following

$$\frac{\partial u}{\partial t} = -a\frac{\partial u}{\partial x} + \frac{a}{2}(\Delta x - \Delta t a)\frac{\partial^2 u}{\partial x^2}. \tag{12}$$

Also, one can acquire other usable forms of the modified equation by calling other helper functions of the `DifferentialEquation` object. For example, calling the following function,

```
DE.latex_modified_equation()
```

will return a LaTeX string representation of the modified equation which can be printed

```
\frac{\partial u }{\partial t}  =  - a \frac{\partial
    u }{\partial x}  + \frac{a}{2} \left(\Delta x - \
    Delta{t} a\right) \frac{\partial^{2} u }{\partial
     x^{2}}
```

In addition, the symbolic form of the modified equation is accessed using `DE.symbolic_modified_equation()`.

In addition, once the RHS of the differential equation is set, one can make a call to the member function `generate_amp_factor()` to generate the amplification factor, $e^{\alpha \Delta t}$. Then by calling `DE.display_amp_factor()`, the user can display the rendered LaTeX form of the amplification factor. For instance, the amplification factor of the PDE discretized in Eq. (9) is rendered and displayed as follows

$$e^{\alpha \Delta t} = 1 - a\frac{\Delta t}{\Delta x}\left(1 - e^{-\mathrm{i}k_x \Delta x}\right), \tag{13}$$

which is the expected amplification factor for this scheme. On a different note, the helper functions for the amplification factor can be used after the call for `DE.generate_modified_equation(...)` and without calling `DE.generate_amp_factor()` first. Similar to the modified equation, other representations of the amplification factor, $e^{\alpha \Delta t}$, are available. For example, one can call the function

```
DE.latex_amp_factor()
```

that will return a LaTeX string representation of the amplification factor that can be printed

```
e^{\alpha\Delta t} = 1 - a \frac{\Delta t}{\Delta x}\
    left(1-e^{-\i k_{x}\Delta x}\right)
```

Also, the symbolic form for the amplification factor is available, and can be accessed by calling `DE.symbolic_amp_factor()`.

3. Illustrative examples

To test PyModPDE, we examine the generation of the modified equation for time dependent PDEs in one and two dimensions.

3.1. Example using BTCS on 1D advection

As an example of the framework proposed above, consider the advection equation

$$u_t = -cu_x; \quad c > 0, \tag{14}$$

to be solved using backward Euler discretization in time and central discretization in space (BTCS). The difference equation is

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = -c\left(\frac{u_{i+1}^{n+1} - u_{i-1}^{n+1}}{2\Delta x}\right). \tag{15}$$

Let $u_i^n = e^{\alpha t}e^{\mathrm{i}kx}$. Upon substitution into the difference equation, we solve for α

$$\alpha = \frac{1}{\Delta t}\log\left(\frac{2\Delta x e^{\mathrm{i}\Delta x k}}{2\Delta x e^{\mathrm{i}\Delta x k} + \Delta t c e^{2\mathrm{i}\Delta x k} - \Delta t c}\right) \tag{16}$$

Finally, according to Eq. (4), we have

$$\begin{aligned} a_1 &= \frac{1}{\mathrm{i}}\alpha'(0) = -c; \quad a_2 = \frac{1}{2\mathrm{i}^2}\alpha''(0) = \frac{\Delta t c^2}{2}; \\ a_3 &= \frac{1}{3!\mathrm{i}^3}\alpha'''(0) = -c\left(\frac{\Delta x^2}{6} + \frac{\Delta t^2 c^2}{3}\right), \end{aligned} \tag{17}$$

and the modified equation is

$$u_t = -cu_x + \frac{\Delta t c^2}{2}u_{xx} - c\left(\frac{\Delta x^2}{6} + \frac{\Delta t^2 c^2}{3}\right)u_{xxx} + \cdots. \tag{18}$$

as expected for this scheme. The coefficient of the diffusion term is positive which indicates that this is a diffusive scheme, and the odd derivatives in the modified equation indicates the existence of dispersion in the numerical solution. We tested this numerically by advecting a Gaussian function u, $u(x, t = 0) = e^{-x^2/0.1}$, using BTCS scheme on a periodic domain from -2 to 2 with

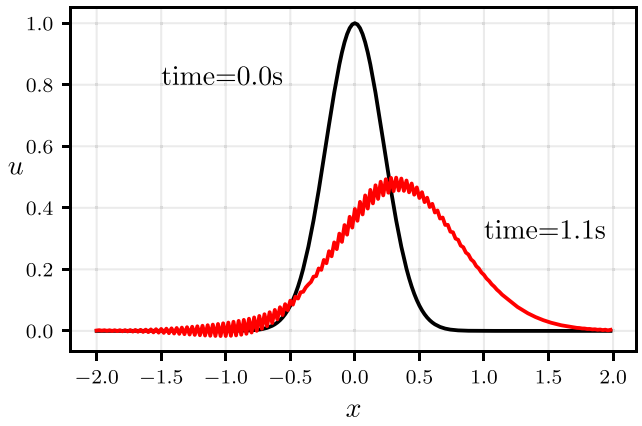

Fig. 2. Advection of a Gaussian function using BTCS on a periodic domain, CFL= 4.

$\Delta x = 0.02$, $\Delta t = 0.01$, and running at CFL= 4. From the second term in the modified equation Eq. (18) we know that the scheme is diffusive no matter what value we pick for Δt and c which is reflected in the numerical results as a decrease in the amplitude of the Gaussian function as the solution is advanced in time. The third term in Eq. (18) is also nonzero regardless of the values of Δx and Δt, which indicates that the scheme is dispersive. This behavior is represented in Fig. 2 as 'wiggles' due to each wavenumber traveling at different speed compared to the original Gaussian function.

To obtain the modified equation for this advection equation using PyModPDE we simply set the following

```
from src.pymodpde import DifferentialEquation
from sympy import symbols

# declare symbols for the velocity, spatial and time
    indices
c, i, n = symbols('c i n')

# construct a time dependent differential equation
DE = DifferentialEquation(dependentVarName='u',
    independentVarsNames=['x'], indices=[i],
    timeIndex=n)

# method I of constructing the rhs
advectionTerm = DE.expr(order=1, directionName='x',
    time=n+1, stencil=[-1, 1])

# set the rhs of the differential equation
DE.set_rhs(- c * advectionTerm )

# generate and display the modified equation up to two
    terms
DE.generate_modified_equation(nterms=3)

# display the modified equation
DE.display_modified_equation()
```

The rendered latex representation of the modified equation is displayed as

$$\frac{\partial u}{\partial t} = -c\frac{\partial u}{\partial x} + \frac{\Delta t c^2}{2}\frac{\partial^2 u}{\partial x^2} - c\left(\frac{\Delta x^2}{6} + \frac{\Delta t^2 c^2}{3}\right)\frac{\partial^3 u}{\partial x^3}$$

3.2. Example using FTUS on 2D advection

As a multidimensional example, consider the advection equation in two dimensions

$$u_t = -bu_x - cu_y; \quad (b, c) \in \mathbb{R}^+ \tag{19}$$

to be discretized using a forward Euler method in time and UPWIND in space (FTUS) as follows

$$\frac{u_{i,j}^{n+1}-u_{i,j}^{n}}{\Delta t}=-b\frac{u_{i,j}^{n}-u_{i-1,j}^{n}}{\Delta x}-c\frac{u_{i,j}^{n}-u_{i,j-1}^{n}}{\Delta y}, \tag{20}$$

where (i,j) refers to the location on a two dimensional grid (not to be confused with the imaginary unit and other indexing in this manuscript). The amplification factor exponent, α, for this case is

$$\alpha=\frac{1}{\Delta t}\ln\left[1-\frac{b\Delta t}{\Delta x}\left(1-e^{-\mathrm{i}k_x\Delta x}\right)-\frac{c\Delta t}{\Delta y}\left(1-e^{-\mathrm{i}k_y\Delta y}\right)\right]. \tag{21}$$

In two dimensions, the modified equation takes the general form

$$\begin{aligned}u_t&=a_{10}u_x+a_{01}u_y+a_{11}u_{xy}+a_{20}u_{xx}+a_{02}u_{yy}\\&\quad+a_{12}u_{xyy}+a_{21}u_{xxy}+\cdots\\&=\sum_{\substack{j,l\geq 0\\ j+l\geq 1}}^{\infty}a_{jl}\frac{\partial^{j+l}u}{\partial x^j\partial y^l}\end{aligned} \tag{22}$$

with the coefficients given by

$$a_{jl}=\frac{1}{\mathrm{i}^n j!l!}\left.\frac{\partial^n\alpha}{\partial k_x^j\partial k_y^l}\right|_{\mathbf{k}=\mathbf{0}}. \tag{23}$$

For the case given by Eq. (21)

$$\begin{aligned}a_{10}&=\frac{1}{\mathrm{i}}\left.\frac{\partial\alpha}{\partial k_x}\right|_{\mathbf{k}=\mathbf{0}}=-b,\quad a_{01}=\frac{1}{\mathrm{i}}\left.\frac{\partial\alpha}{\partial k_y}\right|_{\mathbf{k}=\mathbf{0}}=-c,\\a_{11}&=\frac{1}{\mathrm{i}}\left.\frac{\partial^2\alpha}{\partial k_x\partial k_y}\right|_{\mathbf{k}=\mathbf{0}}=-bc\Delta t,\end{aligned} \tag{24}$$

with a_{11} indicating the presence of anisotropic diffusion in the $x-y$ direction. The numerical diffusion terms are also non-zero with coefficients given by

$$\begin{aligned}a_{20}&=\frac{1}{2!\mathrm{i}^2}\left.\frac{\partial^2\alpha}{\partial k_x^2}\right|_{\mathbf{k}=\mathbf{0}}=\frac{b}{2}\left(\Delta x-\Delta tb\right);\\a_{02}&=\frac{1}{2!\mathrm{i}^2}\left.\frac{\partial^2\alpha}{\partial k_y^2}\right|_{\mathbf{k}=\mathbf{0}}=\frac{c}{2}\left(\Delta y-\Delta tc\right).\end{aligned} \tag{25}$$

The first few terms of the modified equation in this case are

$$\begin{aligned}u_t&=-bu_x-cu_y-bc\Delta tu_{xy}+\frac{b}{2}\left(\Delta x-\Delta tb\right)u_{xx}\\&\quad+\frac{c}{2}\left(\Delta y-\Delta tc\right)u_{yy}+\cdots\end{aligned} \tag{26}$$

The third term on the right-hand-side of Eq. (26) is an anisotropic term that results in contraction in the direction of advection and diffusion in the orthogonal direction of the motion, see Fig. 3.

Using PyModPDE, a few lines of code can generate the modified equation for this numerical scheme, as follows:

```
from src.pymodpde import DifferentialEquation
from sympy import symbols

# declare symbols
c, b, i, j, n = symbols('c b i j n')

# construct a time dependent PDE
DE = DifferentialEquation(dependentVarName='u',
    independentVarsNames=['x','y'], indices=[i,j],
    timeIndex=n)

# method I of constructing the rhs:
advectionTerm1 = -b * DE.expr(order=1, directionName='
    x', time=n, stencil=[-1, 0])
advectionTerm2 = -c * DE.expr(order=1, directionName='
    y', time=n, stencil=[-1, 0])

# set the rhs of the differential equation
DE.set_rhs( advectionTerm1 + advectionTerm2 )
```

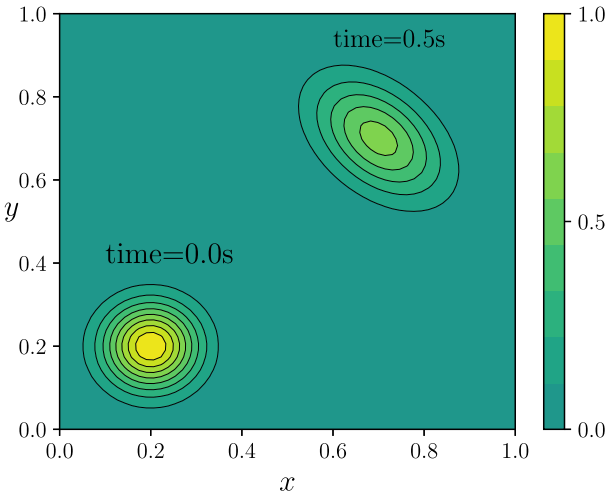

Fig. 3. Advection of Gaussian function in 2D on a periodic domain.

```
# generate and display  the modified equation up to
    two terms
DE.generate_modified_equation(nterms=2)

# display the modified equation
DE.display_modified_equation()
```

The rendered latex form of the modified equation is displayed as

$$\begin{aligned}\frac{\partial u}{\partial t}&=-c\frac{\partial u}{\partial y}-b\frac{\partial u}{\partial x}+\frac{c}{2}\left(\Delta y-\Delta tc\right)\frac{\partial^2 u}{\partial y^2}-\Delta tbc\frac{\partial^2 u}{\partial x\partial y}\\&\quad+\frac{b}{2}\left(\Delta x-\Delta tb\right)\frac{\partial^2 u}{\partial x^2}.\end{aligned}$$

4. Impact

PyModPDE provides a high level of abstraction for the user to describe the finite difference representation of a time dependent differential equation along with functions that generate and return a latex representation of the modified equation.

This software is used as an assisting tool in Numerical methods class at the University of Utah to introduce the concept of the modified equation. Where, students use this tool to help them understand the numerical art effects and the instability that occurs in a certain numerical solution for different combinations of domain resolution and time-step size. By using PyModPDE, students will spend more time understanding the numerical method they are using and less time debugging. Also, students will be able to pick up the conditions for which certain schemes are considered stable.

We are unaware of publicly available software that is similar to PyModPDE. We believe that making PyModPDE available to the community will benefit those involved in numerical analysis and computational physics — both in education and research.

5. Conclusions

In this article, we argue that there is a much easier technique for deriving a modified equation for a given finite difference approximation. First presented by Chang [6], and independently arrived at by the authors, the technique is based on first finding the amplification factor exponent, α, and then taking its derivatives with respect to the wave number. Those derivatives correspond to the coefficients of the modified equation. We also extended the method to higher dimensions where the effect of

cross derivatives is often overlooked when conducting higher-dimensional analyses. We showed how the multidimensional modified equation results in cross-derivative terms that introduce anisotropic effects to advection. Our proposed approach was then implemented in a `Python` software, `PyModPDE`, which was shown to generate the modified equation in latex form for linear time dependent PDEs. It was verified using examples with different time integrators in multiple spatial dimensions. Future work will focus on support for high-order time integrators which result in multiple modified equations.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

The authors gratefully acknowledge support from the Department of Chemical Engineering at the University of Utah.

References

[1] Hirt CW. Heuristic stability theory for finite-difference equations. J Comput Phys 1968;2(4):339–55. http://dx.doi.org/10.1016/0021-9991(68)90041-7, URL http://www.sciencedirect.com/science/article/pii/0021999168900417.

[2] Klopfer G, McRae DS. The nonlinear modified equation approach to analyzing finite difference schemes. In: 5th computational fluid dynamics conference; 1981. p. 1029.

[3] Villatoro FR, Ramos JI. On the method of modified equations. I: Asymptotic analysis of the Euler forward difference method. Appl Math Comput 1999;103(23):111–39. http://dx.doi.org/10.1016/S0096-3003(98)10031-0, URL http://www.sciencedirect.com/science/article/pii/S0096300398100310.

[4] Momoniat E. A modified equation approach to selecting a nonstandard finite difference scheme applied to the regularized long wave equation. Abstr Appl Anal 2014;2014:e754543. http://dx.doi.org/10.1155/2014/754543, URL https://www.hindawi.com/journals/aaa/2014/754543/abs/.

[5] Warming RF, Hyett BJ. The modified equation approach to the stability and accuracy analysis of finite-difference methods. J Comput Phys 1974;14(2):159–79, URL http://www.sciencedirect.com/science/article/pii/0021999174900114.

[6] Chang S-C. A critical analysis of the modified equation technique of warming and hyett. J Comput Phys 1990;86(1):107–26.

[7] Ramshaw JD. Numerical viscosities of difference schemes. Commun Numer Methods Eng 1994;10(11):927–31. http://dx.doi.org/10.1002/cnm.1640101108, URL http://onlinelibrary.wiley.com/doi/10.1002/cnm.1640101108/abstract.

[8] Carpentier R, de La Bourdonnaye A, Larrouturou B. On the derivation of the modified equation for the analysis of linear numerical methods. RAIRO Model Math Anal Numer 1997;31:459–70.