Blow-up Equations with e^u Nonlinearlity (Thermal Runaway)

Xufeng Cai
F1607103
516021910727

2019-06-26

Abstract

In this report, I describe the topic about blow up with e^u nonlinearity in detail, and discuss the uniqueness of solutions and their asymptotic approximation near the blow-up point. Furthermore, I introduce two numerical methods: fining the fixed uniform mesh with finite difference scheme and the adaptive moving mesh method to reproduce the blow-up phenomenon, and discuss their advantages and disadvantages.
Keywords: blow-up equation, uniqueness of solutions, asymptotic approximation, adaptive moving mesh method

1 Topic Description

From mathematical models of heat transfer in reacting media, we can derive the equation

$$u_t = \nabla^2 u + \lambda e^u, \tag{1}$$

where the variable u asymptotically represents the temperature in a large-activation-energy, small-reactant-depletion model, and the diffusion term $\nabla^2 u$ is counteracted by the source term λe^u, which is an approximation to the Aarhenius function $exp(-E/RT)$. The number $\lambda > 0$ is a fixed parameter, which represents the effect on the reaction rate of the mixture of chemical species, and also can be taken as a possibly uniform, regular function of space and time.

We say that the solution of the equation **blows up**, if it ceases to exist for some finite time. One reason for the phenomenon of blow-up in finite time is that the positive feedback introduced by the e^u term allows for an explosive increase of u. We can estimate the rate of the explosion by noting that on an infinite interval, the equation

has the x-independent solution $u = -ln[\lambda(t^* - t)]$ for any number t^*. As $t \to t^*$, the solution u increases to infinity. If we include the diffusion term, the same phenomenon still occurs, except that in general the blow-up occurs at a single point. Specifically, we consider the $1D$ equation

$$u_t = u_{xx} + \lambda e^u \tag{2}$$

for $x \in [-1, 1]$ with boundary conditions $u(\pm 1) = 0$ and initial data zero. Figure 1 shows the blowup process for $\lambda = 1$ as $t \to t^* = 3.54466$. Each time when we move forward one more digit towards t^*, the height of the curve increases by a fixed amount, which illustrates the logarithmic nature of blowup. Moreover, the only infinite point of the curve is $x = 0$, thus the solution remains bounded at each fixed point other than the origin, which is different from the form of some blowups in the blow-up equations with u^p nonlinearity.

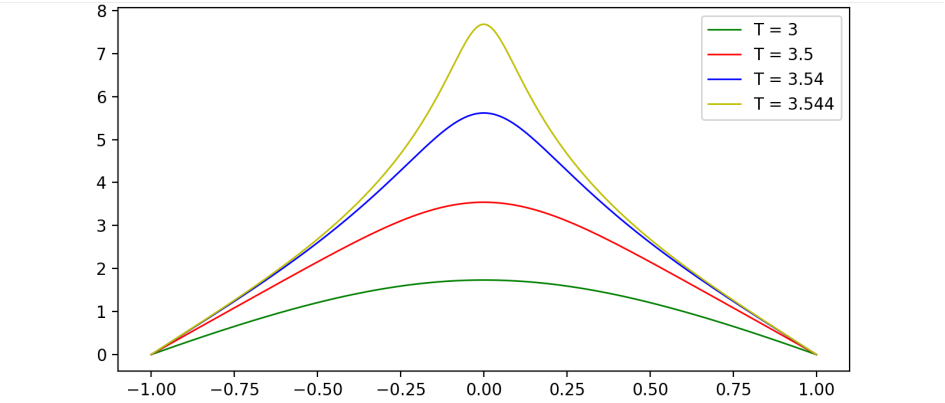

Figure 1: blowup at t ≈ 3.54466 with $\lambda = 1$ and zero initial data

The phenomenon of blow-up in finite time, also called **thermal runaway**, is associated with the non-existence of solutions of the equation for values of λ greater than a critical value λ_c, and I will explain it in detail in the next section.

2 Theoretical Analysis

2.1 Uniqueness of the solution

It has been addressed by Andrew Fowler that blowup in finite time is associated with the non-existence of solutions of the steady problem

$$u_{xx} + \lambda e^u = 0 \tag{3}$$

for values of λ greater than a critical value λ_c.

For the $1D$ equation (1), $\lambda_c \approx 0.878$, and figure 2 illustrates the result for different λ above and below this critical value. Symmetrically equivalent problems in two and three dimensions also have similar behaviors, having $\lambda_c = 2$ and $\lambda_c \approx 3.322$ respectively. Below these critical values, multiple solutions of the steady problem may exist, two in $1D$ and $2D$, and infinitely many in $3D$. As is shown in figure 2, in the simple $1D$ case, the upper branch is unstable. Also, a sufficiently large initial condition may cause thermal runaway here.

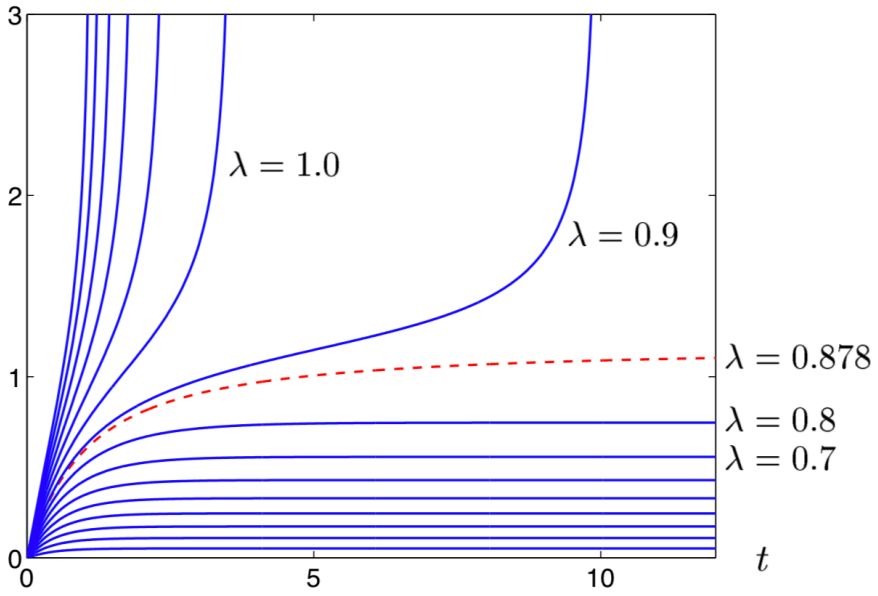

Figure 2: $max_x u(x,t)$ as a function of t for different λ

For further analysis, we consider the boundary problem BVP for the equation

$$\Delta u + e^u = 0 \qquad (x \in \Omega) \tag{4}$$

with zero boundary value. This equation naturally has close relation with the initial value problem (1) with $\lambda = 1$ and initial condition $v|_{t=0} = a(x)$ we want to look into. We assume that Ω is a bounded domain in R^m whose boundary $\partial\Omega$ is sufficiently smooth, and $a = a(x)$ is continuous in $\overline{\Omega}$. As I. M. Gel'fand showed, in the special case where Ω is an m-dimensional ball of radius r, that for $m = 1, 2$ there exists a critical radius r_c such that BVP has two solutions, one solution or no solution, according as $0 < r < r_c$, $r = r_c$ or $r > r_c$. If $m = 3$, then the number of solutions of BVP can be $0, 1, 2, \cdots, \infty$, depending on r.

Moreover, H. Fujita proved a certain relation among solutions of BVP when they exist and study the asymptotic stability of these solutions, i.e. the convergence of solutions

of IVP to solutions of BVP as $t \to \infty$. In detail, let S be the totality of solutions of BVP, then a function $u \in S$ is called the minimum solution if $u \leq v$ for any $v \in S$. Fujita stated that the minimum solution is unique if it exists.

2.2 Asymptotic Analysis

Asymptotic approximation can be applied to look into the spatial structure of the blow-up point. For a problem with blow-up at $x = 0$ and $t = t^*$, define new time and space variable by

$$\tau = -\ln(t^* - t), \qquad \xi = x/\sqrt{4\tau(t^* - t)} \tag{5}$$

Then it can be shown that $u(x,t)$ has an asymptotic expansion close to the blow-up point as $t \to t^*$ that begins

$$u(x,t) \sim \tau - \ln\lambda - \ln\left(1+\xi^2\right) - \frac{5}{2}\frac{\ln\tau}{\tau}\frac{\xi^2}{1+\xi^2} + \frac{1}{\tau}\left[\frac{1}{2} + \frac{\xi^2}{1+\xi^2}\left\{\alpha - \ln\left(1+\xi^2\right)\right\}\right] + \cdots \tag{6}$$

for some constant α. Figure 3 represents the approximation near the blow-up point through the term $-\ln(1+\xi^2)$ for different $t \to t^*$.

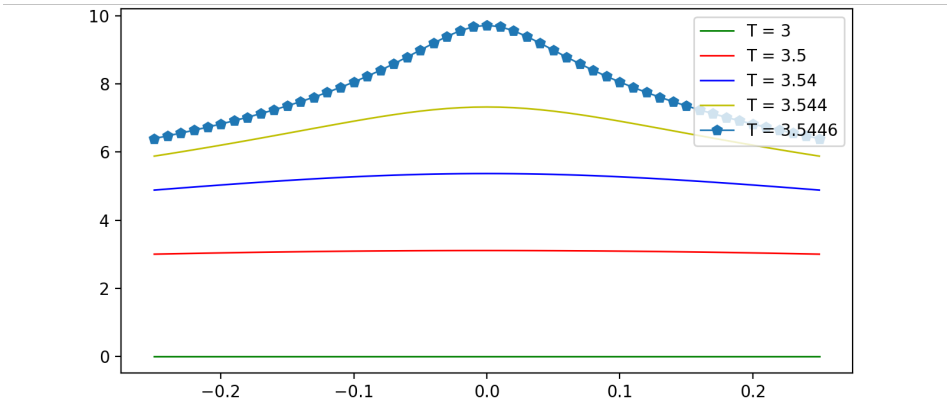

Figure 3: Asymptotic approximation through the term $-\ln(1+\xi^2)$

Due to the existence of such natural spatial coordinate, the so-called ignition kernel, we try to solve for the similarity solution for the problem (1). The above asymptotic expansion gives us an approximate self-similar solution. A similarity solution of the equation is any solution which is invariant under such natural scaling. In fact, We can recast the equation in terms of similarity variables to give a new PDE supplemented with the condition to match the original boundary condition. Then a similarity solution of the original problem is a steady state solution of the new PDE with satisfying the new condition. The natural relationship between the various scaling involves in the

solution of the blow-up problem. Those scaling will be helpful for deciding the choice of an appropriate numerical method.

3 Numerical Methods

It is tricky to solve such blow-up problems. When a singularity forms, changes occur on increasingly smaller length scales and, as the time t^* is approached, on increasingly smaller timescales. If a numerical method with a fixed mesh is used to reproduce such behavior, then its accuracy will diminish significantly when the length scale of the singularity approaches the spacing between mesh points. Here we consider two numerical methods to solve the blow-up problem (2) for $\lambda = 1$ with zero initial condition.

3.1 Fixed mesh with finer mesh points

In order to deal with the problem that the accuracy diminishes as the length scale of the singularity approaches the spacing between mesh points, we can simply fine the mesh according the expected accuracy.

Here we choose the finite difference scheme on a fixed mesh as an example. We apply the standard method of lines to solve the PDE

$$u_t = u_{xx} + e^u, \quad x \in [-1, 1] \tag{7}$$

with zero initial and boundary conditions. We consider its finite difference solution on a uniform spatial mesh. Given a positive integer N, define the mesh

$$\mathscr{T}_h : \quad x_j = -1 + (j-1)h, \quad j = 1, \ldots, N \tag{8}$$

where $h = 2/(N-1)$. A semi-discretization of the equation (7) using central differences in space is given by

$$\frac{du_j}{dt} = \frac{\varepsilon}{h^2}\left(u_{j+1} - 2u_j + u_{j-1}\right) + e^{u_j}, \qquad j = 2, \ldots, N-1 \tag{9}$$

where $u_j(t)$ is an approximation to the solution $u = u(x,t)$ at $x = x_j$, i.e., $u_j(t) \approx u(x_j, t)$. The discrete boundary and initial conditions become

$$u_1(t) = 0, \quad u_N(t) = 0, \quad t > 0 \tag{10}$$

$$u_j(0) = 0, \quad j = 1, \ldots, N. \tag{11}$$

The boundary conditions (10) are replaced by the ODE form for implementation:

$$\frac{du_1}{dt} = 0, \quad \frac{du_N}{dt} = 0. \tag{12}$$

The equations (9) and (12) with the initial condition (11) constitute an initial value problem which can be conveniently solved by an ODE solver, and here we choose the backward method to solve the ODE.

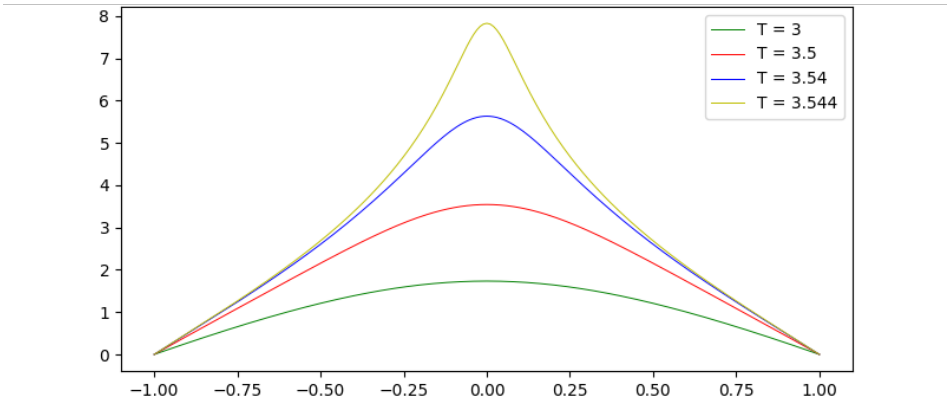

Figure 4: Finite difference scheme on the fixed mesh with $dx = 0.01$

Figure 4 illustrates the results for the finite difference scheme on the fixed mesh with $dx = 0.01$. It seems that such coarse mesh works well for reproducing the blow-up point of the problem (7). However, if we try to move forward one more digit toward t^*, we can see that the numerical solution near the blow-up point differs a lot with the asymptotic approximation, as is shown in figure 5, which means that the spacing between mesh points is not small enough to reproduce the blow-up accurately.

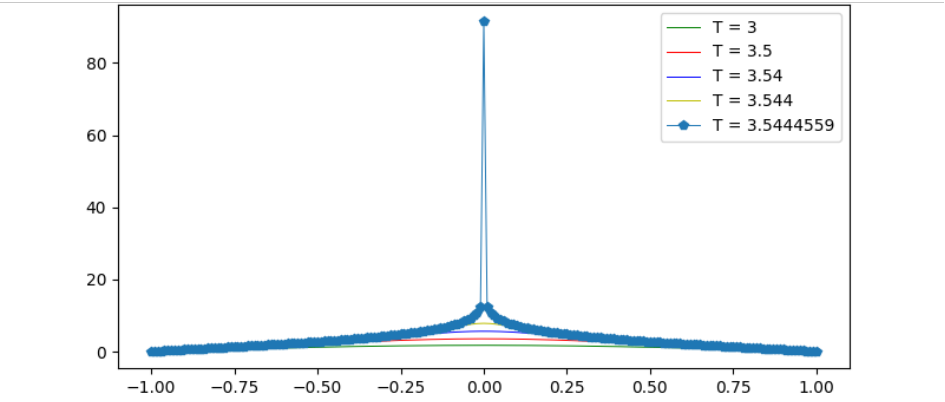

Figure 5: Finite difference scheme on the fixed mesh with $dx = 0.01$ forward to the blow-up point

Instead, we fine the mesh to make the spacing between mesh points to match the length scale of the singularity. Here we choose $dx = 0.005$, and the result is shown in figure 6, where we can see we can move forward one more digit towards t^* compared to situation where $dx = 0.01$.

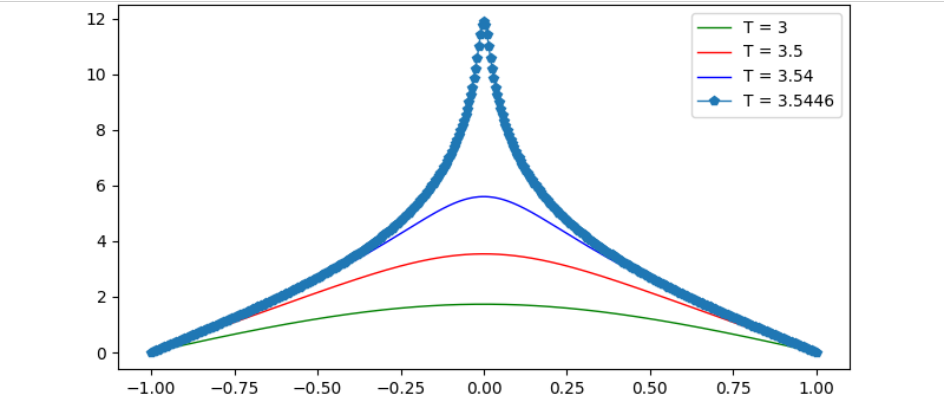

Figure 6: Finite difference scheme on the fixed mesh with $dx = 0.005$

3.2 Adaptive moving mesh

Using very fine mesh is quite expensive in terms of computer time and memory, and much more for two- and three- dimensional problems. Instead, to compute such singular behavior accurately, it is essential and more convenient to use a numerical method which adapts the spatial mesh as the singularity develops. Ideally, we expect that the numerical method will reproduce the singularity sufficiently accurately as $t \to t^*$ to mimic the asymptotic behavior of the solution, which requires that the mesh points be concentrated around the steep structure of the singularity. Such a dynamically adjusting mesh is referred to as an **adaptive moving mesh**.

Adaptive moving mesh is often understood by interpreting the problem in terms of a suitable coordinate transformation. Assume a time-dependent coordinate transformation $x = x(\xi, t) : \Omega_c \equiv [-1, 1] \to \Omega \equiv [-1, 1]$ is given, where Ω_c and Ω are the computational and physical domains respectively. Generally, this transformation is chosen such that the solution in the transformed spatial variable,

$$\hat{u}(\xi, t) = u(x(\xi, t), t) \tag{13}$$

is smooth and economical to approximate using a uniform mesh. A corresponding moving mesh can be described as

$$\mathscr{T}_h(t): \qquad x_j(t) = x\left(\xi_j, t\right), \quad j = 1, \ldots, N \tag{14}$$

for the fixed, uniform mesh on Ω_c,

$$\mathscr{T}_h^c: \quad \xi_j = -1 + \frac{j-1}{N-1}, \quad j = 1, \ldots, N. \tag{15}$$

By the central difference in space and the chain rule, we obtain the semi-discretization on the uniform computational mesh $\mathscr{T}_h^c$,

$$\frac{du_j}{dt} - \frac{u_{j+1} - u_{j-1}}{x_{j+1} - x_{j-1}} \frac{dx_j}{dt} = \frac{2}{x_{j+1} - x_{j-1}} \left[\frac{u_{j+1} - u_j}{x_{j+1} - x_j} - \frac{u_j - u_{j-1}}{x_j - x_{j-1}} \right] + e^{u_j}, \quad j = 2, \ldots, N-1 \tag{16}$$

where $u_j(t) \approx \hat{u}(\xi_j, t) = u(x_j(t), t)$.

The remaining problem is how to determine the coordinate transformation. There are multiple ways to determine it, and one of them is to solve the following so-called moving mesh PDE (MMPDE):

$$x_t = \frac{1}{\rho\tau} (\rho x_\xi)_\xi \tag{17}$$

with the boundary conditions

$$x(-1, t) = 0, \quad x(1, t) = 0. \tag{18}$$

Here, $\rho = \rho(x, t)$ is a mesh density function to choose to control the concentration or density of the mesh, and $\tau > 0$ is a tunable parameter for adjusting the responding time of mesh movement to changes in $\rho(x, t)$. Here we choose $\rho(x, t)$ as e^u. Then the semi-discretization of (17) on the uniform computational mesh $\mathscr{T}_h^c$ gives

$$\frac{dx_j}{dt} = \frac{1}{\rho_j \tau \Delta\xi^2} \left[\frac{\rho_{j+1} + \rho_j}{2} (x_{j+1} - x_j) - \frac{\rho_j + \rho_{j-1}}{2} (x_j - x_{j-1}) \right], \quad j = 2, \ldots, N-1 \tag{19}$$

with the boundary condition $\frac{dx_1}{dt} = 0, \frac{dx_N}{dt} = 0$. Here, $\Delta\xi = 2/(N-1)$, and we need to smooth the mesh density function such as weighted averaging if u is not smooth.

Coupling the above two ODE systems supplemented with boundary conditions, let

$$y = [u_1(t), \ldots, u_N(t), x_1(t), \ldots, x_N(t)]^T, \quad y' = \frac{dy}{dt}. \tag{20}$$

Then the ODE system can be written in the implicit form

$$f(t, y, y') = 0. \tag{21}$$

The underlying PDE and MMPDE are solved simultaneously to determine the solution and the mesh, or can be carried out through iterations.

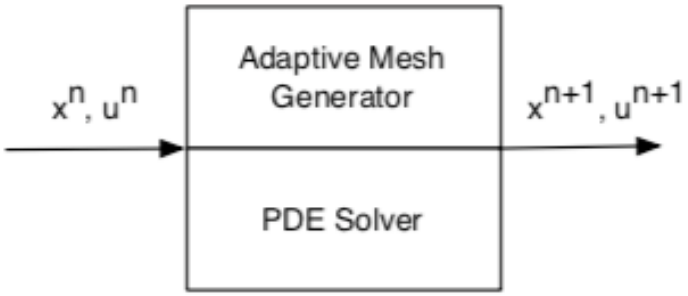

Figure 7: Adaptive moving mesh PDE solver - 1

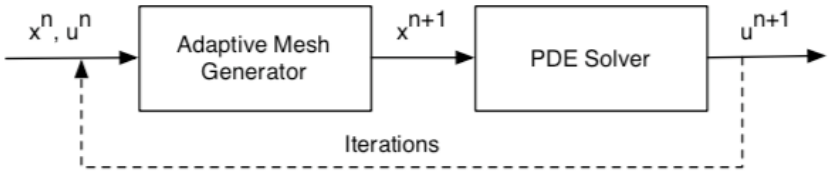

Figure 8: Adaptive moving mesh PDE solver - 2

Figure 9 gives the results of solving the MMPDE (17) to determine the moving mesh with $dx = 0.01$ on the uniform computational mesh.

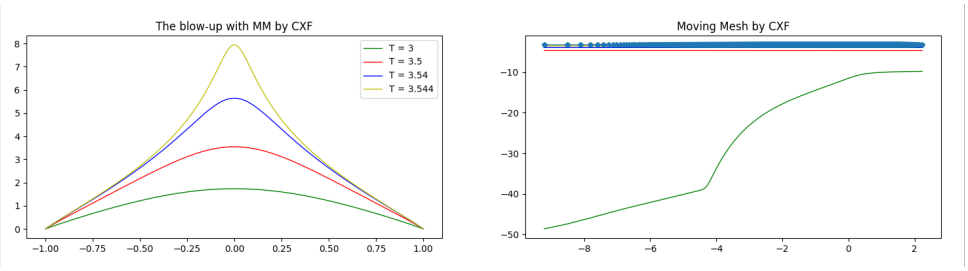

Figure 9: Results for the above MMPDE

If we regard the asymptotic approximation as the exact solution, we can compute the numerical error of the adaptive moving mesh method. Here we use the $L2$ norm, and since asymptotic approximation only approximates the solution accurately near the blow-up point, we take a quarter of mesh points around the blow-up points for computing. Moreover, we can compute the numerical error for the finite difference method on a fixed mesh. Figure 10 shows the numerical error for finite difference methods on a fixed mesh and on a moving mesh against the mesh points.

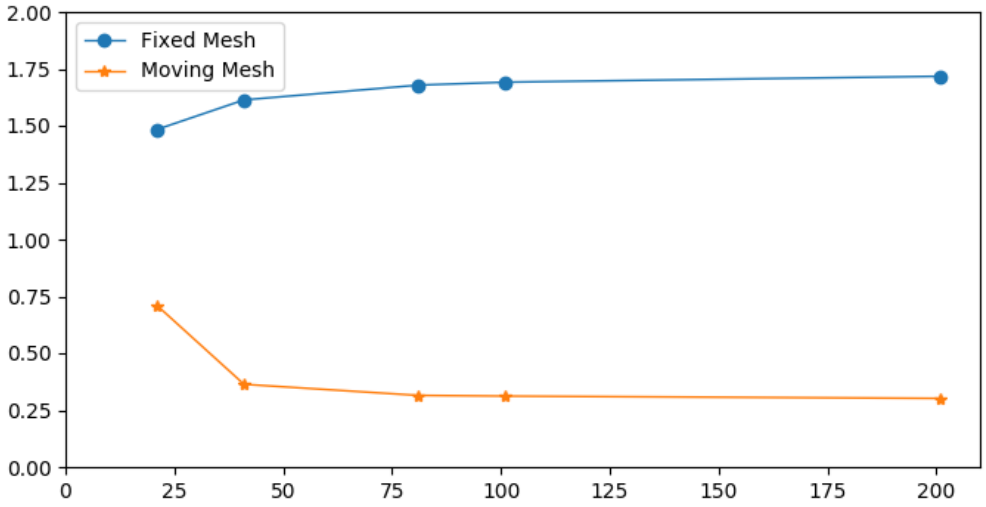

Figure 10: Numerical error for the moving mesh and fixed uniform mesh against number of mesh points at $t = 3.5$

According to the relevant materials, adaptive moving mesh methods shall have advantage both in numerical error and computation time. As we can see in figure 10, the adaptive moving mesh method reproduces the blow-up phenomenon better than finite difference on the fixed uniform mesh. However, such advantage is not significant enough, since we can not move forward one more digit towards $t = t^*$ using adaptive moving mesh methods for the same number of mesh points with fixed uniform mesh. There may be several reasons leading to this consequence.

Firstly, here I use the method of iteration to computing MMPDE and the original PDE, thus there may be a loss of correlation between these two PDEs during the computing process. However, due to the limit of computational resources, I do not run the experiment of simultaneous computing.

Secondly, the choice of the mesh density function , the parameter τ, and the proper difference scheme may effect the result a lot. Here we choose first-order finite difference method, which may influence the computational speed and accuracy.

Last but not least, here we choose to solve the equation (17), which is called modified MMPDE5, to determine the moving mesh (coordinate transformation). In fact, different positive-definite differential operator chosen to satisfy the equidistribution re-

lation will lead to different MMPDEs. For example, we can have

$$\text{(MMPDE4)}: \quad \frac{\partial}{\partial \xi}\left(\rho \frac{\partial x_t}{\partial \xi}\right) = -\frac{1}{\tau}\frac{\partial}{\partial \xi}\left(\rho \frac{\partial x}{\partial \xi}\right), \tag{22}$$

$$\text{(MMPDE6)}: \quad \frac{\partial^2 x_t}{\partial \xi^2} = -\frac{1}{\tau}\frac{\partial}{\partial \xi}\left(\rho \frac{\partial x}{\partial \xi}\right), \tag{23}$$

and the corresponding semi-discretization form can be derived similarly with the modified MMPDE5 (17). Moreover, the difference between MMPDE4 and MMPDE6 lies in that MMPDE4 ceases to evolve the mesh when the timescale of the blow-up is less than τ, while MMPDE6 gives an accurate resolution of the blow-up peak. Hence, choosing an appropriate MMPDE is quite significant when we try to reproduce a blow-up phenomenon accurately.

4 Conclusion

In this report, I introduce the blow-up phenomenon in detail, and discuss the uniqueness of the solution of blow-up equations with e^u nonlinearity. Also, to better understand the structure near the blow-up point, I give the asymptotic expansion of the solution as an approximation. To deal with the problem that the accuracy diminishes as the length scale of the singularity approaches the spacing between mesh points, I introduce two numerical methods respectively. One is fining the fixed uniform mesh and applying the finite difference scheme on it, but such method is expensive both in memory and memory. The other one is the adaptive moving mesh methods, which can dynamically adjust the mesh to match the steep structure near the blow-up point. Such method is better in reproducing the blow-up phenomenon, but is harder to compute (the computing procedure is more complex), and tune the parameter. To sum up, the methods in this report are only preliminary trials, and blow-up phenomenon in many equations concerning physics models calls for more research in the future.

Reference

[1] A. Fowler. (2001). Blow-up equation with e^u nonlinearity.

[2] H. Fujita, On the nonlinear equations $\Delta u + e^u = 0$ and $\partial v/\partial t = \Delta v + e^v$, Bull. Amer. Math. Soc., 75 (1969), 132-135.

[3] Chris J. Budd, Weizhang Huang, and Robert D. Russell, Moving Mesh Methods for Problems with Blow-Up, SIAM J. Sci. Comput., 17(2), 305–327.

[4] W. Huang, R. D. Russel, Adaptive moving mesh methods, Springer, 2011

Appendix: Python code for adaptive moving mesh methods (simultaneous one)

```
from sympy import *
import numpy as np
import cmath
import math
import matplotlib.pyplot as plt
import matplotlib.axes as ax
from mpl_toolkits.mplot3d import Axes3D
from scipy.integrate import ode


global dx
global tau
tau = 1
dx = 0.01
num = int(2 / dx) + 1

def MMPDE(u, x):
    num = int(u.shape[0])
    A = np.zeros((num, num))
    for i in range(num - 1):
        A[i][i + 1] = - 0.5 * (math.exp(u[i + 1]) + math.exp(u[i]))
        A[i + 1][i] = - 0.5 * (math.exp(u[i + 1]) + math.exp(u[i]))
    for i in range(1, num - 1):
        A[i][i] = - A[i][i + 1] - A[i][i - 1]
    A[0][0] = 1
    A[num - 1][num - 1] = 1
    B = np.zeros((num, num))
    for i in range(num - 1):
        B[i][i + 1] = 0.5 * (math.exp(u[i + 1]) + math.exp(u[i])) * tau
        B[i + 1][i] = 0.5 * (math.exp(u[i + 1]) + math.exp(u[i])) * tau
    for i in range(1, num - 1):
        B[i][i] = - B[i][i + 1] - B[i][i - 1]
    B[0][0] = 1
    B[num - 1][num - 1] = 1
    A = np.mat(A)
    B = np.mat(B)
    C = B.I * A
    x = np.mat(x)
    U = C * x.T
    U = np.array(U)
    soln = np.zeros(num)
    soln[0] = 0
    soln[num - 1] = 0
    for i in range(1, num - 1):
        soln[i] = U[i][0]
    return soln
```

```
def f_mm(t, y):
    n = int(y.shape[0])
    U = np.zeros(n)
    num = int(n / 2)
    X = np.zeros(num)
    Y = np.zeros(num)
    for i in range(num):
        X[i] = y[i + num]
    for i in range(num):
        Y[i] = y[i]
    for i in range(num):
        U[i] = amm_f(t, Y, X)[i]
    for i in range(num, n):
        U[i] = MMPDE(Y, X)[i - num]
    return U

def f_mesh(t, y, u):
    n = int(y.shape[0])
    U = np.zeros(n)
    U[0] = 0
    U[n - 1] = 0
    for i in range(1, n - 1):
        U[i] = ( 0.5 * (math.exp(u[i + 1]) + math.exp(u[i])) * (y[i + 1]
                                                - y[i]) - 0.5 * (math.exp(u[
                                                i]) + math.exp(u[i - 1])) *
                                                (y[i] - y[i - 1])) / (math.
                                                exp(u[i]) * tau * dx**2)

def f(t, y):
    n = int(y.shape[0])
    U = np.zeros(n)
    U[0] = 0
    U[n - 1] = 0
    for i in range(1, n - 1):
        U[i] = (y[i + 1] - 2 * y[i] + y[i - 1]) / dx**2 + math.exp(y[i])
    return U

def amm_f(t, y, x):
    n = int(y.shape[0])
    U = np.zeros(n)
    U[0] = 0
    U[n - 1] = 0
    for i in range(1, n - 1):
        U[i] = MMPDE(y, x)[i] * (y[i + 1] - y[i - 1]) / (x[i + 1] - x[i -
                                                  1]) + 2 * ((y[i + 1] - y[i]
                                                ) / (x[i + 1] - x[i]) - (y[i
                                                ] - y[i - 1]) / (x[i] - x[i
                                                - 1])) / (x[i + 1] - x[i - 1
                                                ]) + math.exp(y[i])
    return U
```

```
X_1 = []
X_2 = []
X_3 = []
X_4 = []
X_5 = []
U_max = []

#amm
y_0 = np.zeros(num)
t_0 = 0
t_1 = 3.5446
dt = 0.0001
X = np.linspace(-1, 1, num)
mesh = X
Z = np.zeros(2 * num)
for i in range(num):
    Z[i] = y_0[i]
    Z[i + num] = X[i]
r = ode(f_mm).set_integrator('zvode', method='bdf')
r.set_initial_value(Z, t_0)

plt.figure(figsize = (18, 4))
plt.subplot(121)
while r.successful() and r.t <= 0.03:
    r.integrate(r.t + dt)
    for i in range(num):
        mesh[i] = r.y[i + num]
    print(r.t)
    soln_t = np.zeros(num)
    for i in range(num):
        soln_t[i] = [i]
    if abs(r.t - 3) < 1e-6:
        plt.plot(X, soln_t, "g-", linewidth=1.0, label="T = 3")
    elif abs(r.t - 3.5) < 1e-6:
        plt.plot(X, soln_t, "r-", linewidth=1.0, label="T = 3.5")
    elif abs(r.t - 3.54) < 1e-6:
        plt.plot(X, soln_t, "b-", linewidth=1.0, label="T = 3.54")
    elif abs(r.t - 3.544) < 1e-6:
        plt.plot(X, soln_t, "y-", linewidth=1.0, label="T = 3.544")
    elif abs(r.t - 3.5446) < 1e-6:
        plt.plot(X, soln_t, "p-", linewidth=1.0, label="T = 3.5446")
    elif abs(r.t - 0.025) < 1e-6:
        plt.plot(X, soln_t, "p-", linewidth=1.0, label="T = 3.5446")
    x = mesh[int((num - 1) / 2)]
    x = math.log(abs(x))
    X_1.append(x)
    x = mesh[int((num - 1) / 2) + 1]
    x = math.log(abs(x))
    X_2.append(x)
    x = mesh[int((num - 1) / 2) + 2]
    x = math.log(abs(x))
    X_3.append(x)
```

```
    x = mesh[int((num - 1) / 2) + 3]
    x = math.log(abs(x))
    X_4.append(x)
    x = mesh[int((num - 1) / 2) + 4]
    x = math.log(abs(x))
    X_5.append(x)
    u = math.log(r.y[int((num - 1) / 2)])
    U_max.append(u)
plt.legend()
plt.title("The blow-up with MM by CXF")
plt.subplot(122)
plt.plot(U_max, X_1, "g-", linewidth=1.0)
plt.plot(U_max, X_2, "r-", linewidth=1.0)
plt.plot(U_max, X_3, "b-", linewidth=1.0)
plt.plot(U_max, X_4, "y-", linewidth=1.0)
plt.plot(U_max, X_5, "p-", linewidth=1.0)
plt.title("Moving Mesh by CXF")
plt.savefig("./blow_up.png")
plt.show()
```

Appendix: Python code for adaptive moving mesh methods (iterative one)

```
from sympy import *
import numpy as np
import cmath
import math
import matplotlib.pyplot as plt
import matplotlib.axes as ax
from mpl_toolkits.mplot3d import Axes3D
from scipy.integrate import ode


global dx
global tau
tau = 1000
dx = 0.01
num = int(2 / dx) + 1

def MMPDE(u, x, dt):
    num = int(u.shape[0])
    A = np.zeros((num, num))
    for i in range(num - 1):
        A[i][i + 1] = -dt * 0.5 * (math.exp(u[i + 1]) + math.exp(u[i])) /
                                                    (math.exp(u[i]) * tau * dx
                                                   **2)
        A[i + 1][i] = -dt * 0.5 * (math.exp(u[i + 1]) + math.exp(u[i])) /
                                                    (math.exp(u[i]) * tau * dx
```

```
                                                **2)
    for i in range(1, num - 1):
        A[i][i] = 1 - A[i][i + 1] - A[i][i - 1]
    A[0][0] = 1
    A[num - 1][num - 1] = 1
    A = np.mat(A)
    x = np.mat(x)
    U = A.I * x.T
    U = np.array(U)
    soln = np.zeros(num)
    soln[0] = -1
    soln[num - 1] = 1
    for i in range(1, num - 1):
        soln[i] = U[i][0]
    return soln

def f_mesh(t, y, u):
    n = int(y.shape[0])
    U = np.zeros(n)
    U[0] = 0
    U[n - 1] = 0
    for i in range(1, n - 1):
        U[i] = ( 0.5 * (math.exp(u[i + 1]) + math.exp(u[i])) * (y[i + 1]
                                                - y[i]) - 0.5 * (math.exp(u[
                                                i]) + math.exp(u[i - 1])) *
                                                (y[i] - y[i - 1])) / (math.
                                                exp(u[i]) * tau * dx**2)

def f(t, y):
    n = int(y.shape[0])
    U = np.zeros(n)
    U[0] = 0
    U[n - 1] = 0
    for i in range(1, n - 1):
        U[i] = (y[i + 1] - 2 * y[i] + y[i - 1]) / dx**2 + math.exp(y[i])
    return U

def amm_f(t, y, x):
    n = int(y.shape[0])
    U = np.zeros(n)
    U[0] = 0
    U[n - 1] = 0
    for i in range(1, n - 1):
        U[i] = ( 0.5 * (math.exp(y[i + 1]) + math.exp(y[i])) * (x[i + 1]
                                                - x[i]) - 0.5 * (math.exp(y[
                                                i]) + math.exp(y[i - 1])) *
                                                (x[i] - x[i - 1])) / (math.
                                                exp(y[i]) * tau * dx**2) * (
                                                y[i + 1] - y[i - 1]) / (x[i
                                                + 1] - x[i - 1]) + 2 * ((y[i
                                                 + 1] - y[i]) / (x[i + 1] -
                                                x[i]) - (y[i] - y[i - 1]) /
```

```
                                                (x[i] - x[i - 1])) / (x[i +
                                                1] - x[i - 1]) + math.exp(y[
                                                i])
    return U

X_1 = []
X_2 = []
X_3 = []
X_4 = []
X_5 = []
U_max = []

#amm
y_0 = np.zeros(num)
t_0 = 0
t_1 = 3.5444
dt = 0.0001
X = np.linspace(-1, 1, num)
mesh = X
r = ode(f).set_integrator('zvode', method='bdf')
r.set_initial_value(y_0, t_0)
plt.figure(figsize = (18, 4))
plt.subplot(121)
while r.successful() and r.t <= t_1:
    r.integrate(r.t + dt)
    mesh = MMPDE(r.y, mesh, dt)
    print(r.t)
    if abs(r.t - 3) < 1e-6:
        plt.plot(X, r.y, "g-", linewidth=1.0, label="T = 3")
    elif abs(r.t - 3.5) < 1e-6:
        plt.plot(X, r.y, "r-", linewidth=1.0, label="T = 3.5")
    elif abs(r.t - 3.54) < 1e-6:
        plt.plot(X, r.y, "b-", linewidth=1.0, label="T = 3.54")
    elif abs(r.t - 3.544) < 1e-6:
        plt.plot(X, r.y, "y-", linewidth=1.0, label="T = 3.544")
    elif abs(r.t - 3.5446) < 1e-6:
        plt.plot(X, r.y, "p-", linewidth=1.0, label="T = 3.5446")
    elif abs(r.t - 0.025) < 1e-6:
        plt.plot(X, r.y, "p-", linewidth=1.0, label="T = 3.5446")
    x = mesh[int((num - 1) / 2)]
    x = math.log(abs(x))
    X_1.append(x)
    x = mesh[int((num - 1) / 2) + 1]
    x = math.log(abs(x))
    X_2.append(x)
    x = mesh[int((num - 1) / 2) + 2]
    x = math.log(abs(x))
    X_3.append(x)
    x = mesh[int((num - 1) / 2) + 3]
    x = math.log(abs(x))
    X_4.append(x)
    x = mesh[int((num - 1) / 2) + 4]
```

```
    x = math.log(abs(x))
    X_5.append(x)
    u = math.log(r.y[int((num - 1) / 2)])
    U_max.append(u)
    Y = r.y
    T = r.t
    r = ode(amm_f).set_integrator('zvode', method='bdf')
    r.set_initial_value(Y, T).set_f_params(mesh)
plt.legend()
plt.title("The blow-up with MM by CXF")
plt.subplot(122)
plt.plot(U_max, X_1, "g-", linewidth=1.0)
plt.plot(U_max, X_2, "r-", linewidth=1.0)
plt.plot(U_max, X_3, "b-", linewidth=1.0)
plt.plot(U_max, X_4, "y-", linewidth=1.0)
plt.plot(U_max, X_5, "p-", linewidth=1.0)
plt.title("Moving Mesh by CXF")
plt.savefig("./blow_up.png")
plt.show()
```

Appendix: Python code for calculating the numerical error

```
from sympy import *
import numpy as np
import cmath
import math
import matplotlib.pyplot as plt
import matplotlib.axes as ax
from mpl_toolkits.mplot3d import Axes3D
from scipy.integrate import ode

global dx
global tau
tau = 100
dx = 0.1
num = int(2 / dx) + 1

def ta(t):
    return -math.log(3.54466 - t)

def xi(x, t):
    return x / math.sqrt(4 * ta(t) * (3.54466 - t))

def u_exact(x, t):
    return ta(t) - math.log(1 + xi(x, t)**2)
```

```
def MMPDE(u, x, dt):
    num = int(u.shape[0])
    A = np.zeros((num, num))
    for i in range(num - 1):
        A[i][i + 1] = -dt * 0.5 * (math.exp(u[i + 1]) + math.exp(u[i])) /
                                                  (math.exp(u[i]) * tau * dx
                                                 **2)
        A[i + 1][i] = -dt * 0.5 * (math.exp(u[i]) + math.exp(u[i - 1])) /
                                                  (math.exp(u[i]) * tau * dx
                                                 **2)
    for i in range(1, num - 1):
        A[i][i] = 1 - A[i][i + 1] - A[i][i - 1]
    A[0][0] = 1
    A[num - 1][num - 1] = 1
    A = np.mat(A)
    x = np.mat(x)
    U = A.I * x.T
    U = np.array(U)
    soln = np.zeros(num)
    soln[0] = -1
    soln[num - 1] = 1
    for i in range(1, num - 1):
        soln[i] = U[i][0]
    return soln

def f(t, y):
    n = int(y.shape[0])
    U = np.zeros(n)
    U[0] = 0
    U[n - 1] = 0
    for i in range(1, n - 1):
        U[i] = (y[i + 1] - 2 * y[i] + y[i - 1]) / dx**2 + math.exp(y[i])
    return U

def amm_f(t, y, x):
    n = int(y.shape[0])
    U = np.zeros(n)
    U[0] = 0
    U[n - 1] = 0
    for i in range(1, n - 1):
        U[i] = ( 0.5 * (math.exp(y[i + 1]) + math.exp(y[i])) * (x[i + 1]
                                                  - x[i]) - 0.5 * (math.exp(y[
                                                  i]) + math.exp(y[i - 1])) *
                                                  (x[i] - x[i - 1])) / (math.
                                                  exp(y[i]) * tau * dx**2) * (
                                                  y[i + 1] - y[i - 1]) / (x[i
                                                  + 1] - x[i - 1]) + 2 * ((y[i
                                                   + 1] - y[i]) / (x[i + 1] -
                                                  x[i]) - (y[i] - y[i - 1]) /
                                                  (x[i] - x[i - 1])) / (x[i +
                                                  1] - x[i - 1]) + math.exp(y[
                                                  i])
```

```
    return U

t_0 = 0
e1 = 0
e2 = 0
X = np.linspace(-1, 1, num)
mesh = X
y_0 = np.zeros(num)
y_1 = np.zeros(num)
y_2 = np.zeros(num)
r = ode(f).set_integrator('zvode', method='bdf')
r.set_initial_value(y_1, t_0)
#s = ode(f).set_integrator('zvode', method='bdf')
#s.set_initial_value(y_2, t_0)
t_1 = 3.54466
dt = 0.0001
while r.successful() and r.t <= t_1:
    r.integrate(r.t + dt)
    print(r.t)
    #s.integrate(s.t + dt)
    #mesh = MMPDE(s.y, mesh, dt)
    Y = r.y
    T = r.t
    r = ode(f).set_integrator('zvode', method='bdf')
    r.set_initial_value(Y, T).set_f_params(mesh)
    if abs(r.t - 3.5) < 1e-6:
        for i in range(1, num - 1):
            y_0[i] = u_exact(-1 + i * dx, r.t)
        for i in range(int((num - 1) / 2), int((num - 1) * 3 / 4)):
            e1 += (y_0[i] - r.y[i])**2
        e1 = e1 * 4 / int(num - 1)
        print(math.sqrt(e1))
        #for i in range(1, num - 1):
        #    y_0[i] = u_exact(mesh[i], s.t)
        #for i in range(int((num - 1) / 2), int((num - 1) * 3 / 4)):
        #    e2 += (y_0[i] - s.y[i])**2
        #e2 = e2 * 4 / int(num - 1)
        #print(math.sqrt(e2))
        break
```