American Journal of Computational and Applied Mathematics 2021, 11(3): 65-70
DOI: 10.5923/j.ajcam.20211103.03

Comparison of Neural Network and Finite Difference Solutions

Jay P. Narain

Retired, Worked at Lockheed Martin Corporation, Sunnyvale, CA, USA

Abstract The neural network solution method is applied to solve coupled non-linear differential equations for a free convection problem on a stationary wall. The results show good comparison with results from other published methods. Next a finite difference equation solver is extended to three and four dimensions. The code is tested for zero Dirichlet boundary condition and mixed non-zero Dirichlet and Neumann boundary conditions cases. The results are compared with neural network solutions with satisfactory outcome. The finite difference equation has also been extended to polar coordinates to solve potential flow problem over a two-dimensional infinite cylinder. The results are excellent.

Keywords Differential equations: non-linear ordinary and partial, Neural netwok, Finite difference equations in polar, Three and four diemensions

1. Introduction

The ordinary and partial differential equations have been solved anaytically and numerically for many years. Various numerical schemes were developed with the advent of computers. From such schemes, the finite difference scheme has been a popular numerical method. The boundary condition based neural network method for non linear Blasius equation was investigated by Kitchin [1]. This work was extended to more complicated Falkar-Skan equations by Narain [2]. The neural network method is capable of solving coupled non-linear differential equations as well as ordinary to partial differential equations. There are numerous applications of neural network method to solve fractional differential equations in literature. Most of the finite-difference solvers are limited to two-dimensional domain. Extension of finite difference method to higher dimension will be desirable for analysis purposes. This will help in improving the neural network prediction capability. The finite difference runtime efficiency is extremely high in comparison to neural network solution. However, neural network solutions maintain high accuracy in any dimension irrespective of grid dimensions.

2. Discussions

Next, we will discuss a new non-shooting method which works well for most linear and nonlinear differential equations [1,2]. We will solve a set of coupled thermal boundary layer equations for free convection thermal boundary layer over a stationary flat plate. Derivation of the governing equations can be found in Deen’s book [3]. In the Boussinesq approximation for an incompressible fluid and a steady-state regime, the equations of momentum, mass and energy conservation for laminar free convection in a plane boundary layer are as follows

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = g\beta(T - T_\infty) + v\frac{\partial^2 u}{\partial y^2},$$
$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0,$$
$$u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y} = a\frac{\partial^2 T}{\partial y^2}. \quad (1)$$

We shall reduce the equations to the dimensionless form. It is convenient to use the quantities entering into the unambiguity conditions (boundary conditions) as the reduction scales. As a linear scale we shall take some characteristic dimension of a body L, for a temperature it is convenient to use, for instance, the relation $\theta = (T - T_\infty)/(T_w - T_\infty)$, where T_w is the body surface temperature, T_∞, is the surrounding temperature, T is the local temperature. The characteristic velocity may be obtained from the comparison of volumetric and viscosity forces $u_0 = \beta g\Delta TL^2/\nu$ or from the estimates of the type $u_0 = L/\tau_0$, where τ_0 is the time scale. The dimensionalization yields:

$$U\frac{\partial U}{\partial X} + V\frac{\partial U}{\partial Y} = \theta + Gr^{-1/2}\frac{\partial^2 U}{\partial Y^2},$$
$$U\frac{\partial \theta}{\partial X} + V\frac{\partial \theta}{\partial Y} = Gr^{-1/2}\,Pr^{-1}\frac{\partial^2 \theta}{\partial Y^2}. \quad (2)$$

* Corresponding author:
narain2@yahoo.com (Jay P. Narain)
Received: Sep. 20, 2021; Accepted: Oct. 13, 2021; Published: Oct. 30, 2021
Published online at http://journal.sapub.org/ajcam

The Grashof number $Gr = \beta g \Delta T L^3/\nu^2$ is the main governing criterion and the most important characteristic of free convection heat transfer. It is the measure of the relation between the buoyancy forces in a non-isothermal flow and the forces of molecular viscosity. It also determines the mode of medium motion along the heat transfer surface. In its physical meaning, it is similar to the Reynolds number for a forced flow.

A specific feature of laminar free convection on a vertical plate at a constant wall temperature is the fact that it allows a self-similar solution if a new variable is introduced into above equations in the form

$$\eta_s \equiv \frac{y}{x}\left(\frac{Gr_x}{4}\right)^{1/4}.$$

The boundary conditions are

$$Y = 0: \; u = v = 0, \; T = T_w$$

$$Y \Rightarrow \infty : u \to 0, \quad T \to T_\infty .$$

Having represented the stream function and the dimensionless temperature as

$$\Psi(x, y) \equiv f(\eta_s)\left[4\nu\left(\frac{Gr_x}{4}\right)^{1/4}\right]; \; \theta = \frac{T - T_\infty}{T_w - T_\infty},$$

we obtain Eqs. (1) in the form

$$f''' + 3ff'' - 2(f')**2 + \theta = 0 \tag{3}$$

$$\theta'' + 3Prf\theta' = 0 \tag{4}$$

with the boundary conditions:

$$\text{at} \quad \eta = 0, \; f = f' = 0, \theta = 1 \tag{5}$$

$$\text{at} \quad \eta = \infty, \; f' = 0, \theta = 0 \tag{6}$$

where f', f'', and f''' are first, second and third derivatives, * and ** denote multiplication and exponential mathematical operation.

This problem was solved originally by Ostrach [4]. Sparrow and Gregg [5] also solved this problem and found their results in agreement with experimental data.

The present numerical scheme consists of two neural networks, one for stream function f and another for modified temperature function θ.

```
def swish(x):
" Activation function"
    return x / (1.0 + np.exp(-x))

def f(params, inputs):
    "Neural network functions"
    for W, b in params:
        outputs = np.dot(inputs, W) + b
        inputs = swish(outputs)
    return outputs

def th(params2, inputs):
    "Neural network functions"
    for W1, b1 in params2:
        outputs = np.dot(inputs, W1) + b1
        inputs = swish(outputs)
    return outputs
```

The network differentiation is carried out with "autograd" differentiation scheme [6].

```
fp = elementwise_grad(f, 1)
fpp = elementwise_grad(fp, 1)
fppp = elementwise_grad(fpp, 1)

thp = elementwise_grad(th, 1)
thpp = elementwise_grad(thp, 1)
```

The loss functions are created by taking mean of the equations (3) thru (6).

Stream function loss function setup:

```
def objective(params, step):
        # Good for free heat transfer on a vertical plate
        zeq = fppp(params, eta) + 3.* f(params, eta) * fpp(params, eta) -2.*fp(params, eta)**2 + th(params2, eta)
        bc0 = f(params, 0.0)   # equal to zero at solution
        bc1 = fp(params, 0.0)   # equal to zero at solution
        ## the bc2 cahnges wit problem. The next is for free conv near vert plate
        bc2 = fp(params, ab) # this is the one at "infinity"
        return np.mean(zeq**2) + np.mean(bc0**2 + bc1**2 + bc2**2 )
```

Tempearture function loss function setup:

```
def objective2( params2, step):
        # Good for free heat transfer on a vertical plate
        zeq2 = thpp(params2, eta) + 3.* Pr*f(params,eta)*thp(params2, eta)
        bc3 = th(params2, 0.0) - 1.0
        bc4 = th(params2, ab)
        return np.mean(zeq2**2) + np.mean(bc3**2 + bc4**2)
```

Next these functions are minimized simultaneously using Adam [7] optimization.

```
for iterations   in range(400):
    params = adam(grad(objective), params,
                step_size=0.005, num_iters=5, callback=callback)

    params2 = adam(grad(objective2), params2,
                step_size=0.005, num_iters=5, callback=callback2)
```

The results are carried out for Pr=0.1, 1.0, and 10. It took about 620 s on an Intel i7-6700T, 2.8 Ghz, 8 GB ram home personal computer.

We will compare the results of our neural network with a well-known solver [8,9] "fsolve" which is used to find root of an equation. Only a few iterations are required with this method, to iteratively solve the coupled equations and get a good solution. This is a non-shooting, non-machine learning method scheme which uses finite difference form of equations and takes lot less run time. It took only 47.7 s to solve. The results are compared for both the schemes and are presented in the following figures.

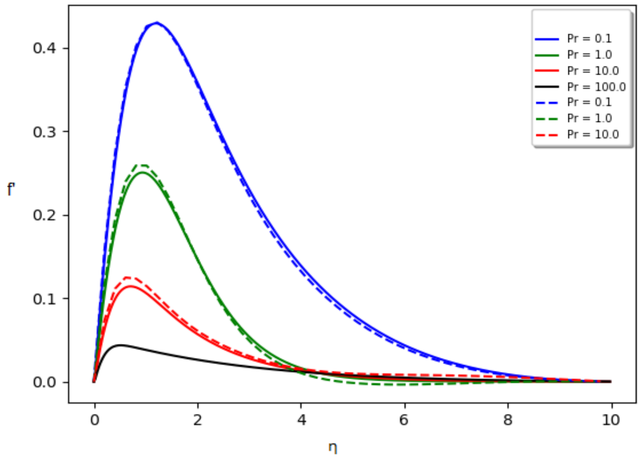

Figure (1). Comparison of velocity(f') profile from neural network (-- lines) and "fsolve" (solid lines) solvers

The case for Pr = 100 did not run with the neural network.

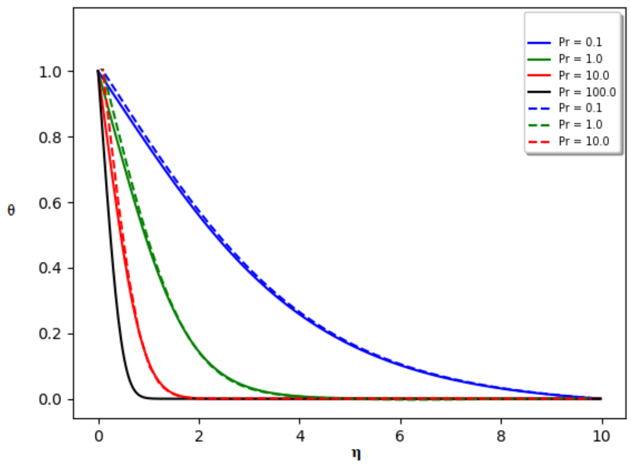

Figure (2). Comparison of modified temperature profile with the two schemes

In conclusion, the finite difference solvers are computationally more efficient and stable. On the other hand, the neural network method does not require knowledge of any numerical methods, such as finite difference, and are easy to set up. Overall, the neural network method is quite slow and needs considerable improvement in computational efficiency in future. The code has been hosted in narain42/Machine-learning-methods-in-Fluid-Dynamics on GitHub.

Recently the solution of Poisson's equation was presented in three and four dimensions using neural network scheme [10]. The computational time was staggering and of some concern for their practicality. Presently we will develop iterative finite difference solver up to four dimensions using known solution methods such as Jacobi, Gauss-Seidel, Gradient descent and Conjugate gradient. A base two-dimensional code [11] was extended to higher dimensions. The basic two-dimensional Jacobi Poisson's equation iterator was for similar grid in two dimensions(dx=dy). Here the grid spacing in two dimensions are kept distinct.;

```
Two dimension: grid (x,y, in range 0,1)
denom = 2*(dx**2 + dy**2)
pxx[1:-1,1:-1] = ((p[1:-1,2:] + p[1:-1,:-2])*dy**2 )/ denom
pyy[1:-1,1:-1] = ( (p[2:,1:-1] + p[:-2,1:-1])*dx**2)/ denom
pnew[1:-1, 1:-1] = pxx[1:-1,1:-1] + pyy[1:-1,1:-1] -
(b[1:-1,1:-1]*dx**2 * dy**2) / (denom)
```
(7)

where dx and dy are the grid spacing increment, p is the solution function, pxx and pyy are the second derivative of p with respect to x and y, and b is the source function for a typical Poisson's equation. pnew is the iterative update in Python subject to the following boundary conditions:

```
pnew[:,0] = a given function of x at y=0
pnew[:, ny-1] = a given function of x at y=1
pnew[0, :] = a given function of y at x=0
pnew[nx-1, :] = a given function of y at x=1
```
(8)

For a contrived test case with equation f(x,y), the source function b(x,y) is Laplacian (pxx+pyy) of the function f(x,y). The boundary conditions can be easily derived by setting x,y values in the f(x,y). In reality, the exact function f(x,y) will not be known in advance, but the boundary conditions and the source function will be known for a given scenario.

The extension to three dimension and four dimensions is as follows:

```
Three dimension: grid (x,y,z in range 0,1)
denom = 2.*(dx**2*dy**2 + dy**2*dz**2+dz**2*dx**2)
pzz[1:-1,1:-1,1:-1] = ( (p[2:,1:-1,1:-1] +
p[:-2,1:-1,1:-1])*dx**2*(dy**2))/ denom
pyy[1:-1,1:-1,1:-1] = ((p[1:-1,2:,1:-1] +
p[1:-1,:-2,1:-1])*dx**2*(dz**2) )/ denom
pxx[1:-1,1:-1,1:-1] = ( (p[1:-1:,1:-1,2:] +
p[1:-1,1:-1,:-2])*dy**2*(dz**2))/ denom
pnew[1:-1,1:-1, 1:-1] = pxx[1:-1,1:-1,1:-1] +
pyy[1:-1,1:-1,1:-1] + pzz[1:-1,1:-1,1:-1] \
- (b[1:-1,1:-1,1:-1]*(dx**2 )* (dy**2)* (dz**2))/ denom (9)
#Boundary Conditions
pnew[0,:,:] = a given function of x and y at z=0
pnew[nz-1,:,:]= a given function of x and y at z=1
pnew[:,0,:] = a given function of x and z at y=0
pnew[:,ny-1,:] = a given function of x and z at y=1
pnew[:,:,0] = a given function of y and z at x=0
pnew[:,:,nx-1] = a given function of y and z at x=1 (10)
```

where dx, dy, and dz are the grid spacing increment, p is the solution function, pxx ,pyy and pzz are the second derivative of p with respect to x, y, and z, and b is the source function in a typical Poisson's equation. pnew is the iterative update for p. The equations are written in Python:

The extension to four dimensions is routinely intuitive and will be provided in GitHub repository. Presently three-point central differencing is used which is second order accurate. Higher order differencing 14 to 27 order have been published for three-dimension case to solve Poisson's equations using matrix inversion. [12]

The Gauss-Seidel scheme follows same equations, where some of the explicit terms are replaced with implicit terms. With the Pythonic vectorization., the results and run times are similar to Jacobi iterative scheme.

The Gradient descent and Conjugate gradient are very powerful and run time efficient schemes. They are limited to any dimensional case with zero Dirichlet boundary conditions. It only requires the Poisson's operator A(v), and source function b. The A(v) is extended as follows:

```
if dimension == 2:
Av = -((v[:-2, 1:-1]-2.0*v[1:-1, 1:-1]+v[2:, 1:-1])/dx**2
      + (v[1:-1, :-2]-2.0*v[1:-1,1:-1]+v[1:-1, 2:])/dy**2)
elif dimension == 3:
Av = -((v[:-2,1:-1, 1:-1]-2.0*v[1:-1,1:-1, 1:-1]+v[2:,1:-1, 1:-1])/dx**2 \
      + (v[1:-1, :-2, 1:-1]-2.0*v[1:-1,1:-1,1:-1]+v[1:-1, 2:,1:-1])/dy**2 \
      + (v[1:-1, 1:-1, :-2 ]-2.0*v[1:-1,1:-1,1:-1]+v[1:-1, 1:-1, 2:])/dz**2 )          (11)
```

The solution procedure is very short and is defined in the code.

The present computational methods are compared with the neural network for a zero Dirichlet case and a case with non-zero boundary conditions.

Case 1:

Zero Dirichlet Boundary conditions: The analytical function will be sin(pi*x)*sin(pi*y)*sin(pi*z)*sin(pi*t). This function can be used for any dimension by deleting extra dimension in the function, like for two dimension use sin(pi*x)*sin(pi*y). The grid size is (31x31x31x31) with order reduction for lower dimension analysis. Accuracy is determined as follows consistent with that used in neural network solutions [10].

```
ex_norm = np.linalg.norm(p_exact)
p_norm = np.linalg.norm(pnew)
norm_diff = ex_norm-p_norm
Accuracy = 100.0 *(1. - (abs(norm_diff)/ex_norm))   (12)
```

where p_exact is the contrived analytical function, pnew is the derived solution, linalg.norm is a numpy function which returns the norm of a vector or matrix.

Dimension	Method	Run Time in seconds	% Accuracy
Two Dimension	Matrix Inversion	0.0468	99.90
	Jacobian	0.90625	99.90
	Gauss-Seidel	0.9218	99.90
	Gradient descent	0.005	99.90
	Conjugate Gradient	0.002	99.81
	Neural Network	400.0	99.45
Three Dimension			
	Jacobian	15.2968	87.32
	Gauss-Seidel	17.125	87.32
	Gradient descent	8.201	87.32
	Conjugate Gradient	0.1875	87.32
	Neural Network	17809.32	96.27
Four Dimension			
	Jacobian	1011.265	76.67
	Gauss-Seidel	1200.276	76.67
	Gradient descent	64.187	76.67
	Conjugate Gradient	13.484	76.67
	Neural Network	18000.00	80.4864 (4x11 grid)

The accuracy seems to decrease with higher dimensionality. Obviously higher order difference schemes will improve the results. The neural network solutions show much better capability and are insensitive to grid size used. Above table shows that in four-dimensional case, a coarse (4(11)) grid gives better solution than (4(31)) grid used in finite difference solution.

Case 2:

Dimension	Method	Run Time in seconds	%Accuracy
Two Dimension			
	Jacobian	0.84375	99.999
	Gauss-Seidel	0.953125	99.999
	Neural Network	400.0	99.45
Three Dimension			
	Jacobian	22.671875	89.397
	Gauss-Seidel	22.640262	89.397
	Neural Network	20935.45	85.775
Four Dimension			
	Jacobian	1535.2968	89.867
	Gauss-Seidel	1469.9531	89.867
	Neural Network	42155.00	98.34 (4x11 grid)

Non-zero Mixed Dirichlet and Neumann Boundary conditions: The analytical function will be exp(-x)*(x + y **3 + z**3 + t**3) This function can be used for any dimension by deleting extra dimension in the function, like for two dimension, use exp(-x)*(x + y **3) The grid size is (31x31x31x31) with order reduction for lower dimension analysis. Accuracy is determined by using equation (12). Only Jacobian and Gauss Seidel results will be compared with neural network since gradient descent and conjugate gradient capability are limited to zero Dirichlet boundary conditions. Matrix inversion method was not attempted for higher dimensions.

The accuracy seems to decrease with higher dimensionality. Obviously higher order difference schemes will improve these results. The neural network solutions show much better capability and are insensitive to grid size used. Above table shows that in four-dimensional case, a coarse (4(11)) grid gives better solution than (4(31)) grid used in finite difference solution.

Case 3:

Most of the neural network analysis is done in cartesian coordinate. Even the regression theory uses cartesian concept. A solution of potential fluid flow over a cylinder has a Laplacian in polar coordinate with source term equal to zero. An attempt to analyze this problem with neural network was partially successful [10]. Even the finite difference formulation runs into problem because of the nature of the equation below:

$$\Delta\phi = \delta^2\phi/ \delta r\text{^}2 + (1/r)\ \delta\phi/ \delta r + (1/r\text{^}2)\ \delta^2\phi/ \delta\theta\text{^}2 = 0 \quad (13)$$

where polar coordinates are radius r and angle θ.

The coding of second and first derivative is straight forward in finite difference scheme. The coefficients (1/r) and (1/r^2) create problem in Pythonic vector notation. Since r is always positive, we can assume these as coefficients and use them as variable scalar quantities. With the coding notation of x as r and y as theta, the following Python code gives the finite difference operator in polar coordinate:

```
rx2 = np.dot(x[1:-1],x[1:-1])
rx1 = math.sqrt(rx2)
denom = 2.*(dx**2 + dy**2*rx2)
pyy[1:-1,1:-1] = ((p[1:-1,2:] + p[1:-1,:-2])*dx**2 )
pxx[1:-1,1:-1] = ( (p[2:,1:-1] + p[:-2,1:-1])*dy**2*rx2 )
px[1:-1,1:-1] = ( (p[2:,1:-1] - p[:-2,
1:-1])*dx**2*dy**2*rx2 )/ (2.*rx1*dx)
pnew[1:-1,1:-1] = (pxx[1:-1,1:-1] + pyy[1:-1,1:-1] +
px[1:-1,1:-1]) / denom                                  (14)
```

The equation (13) has a simple closed form solution on a cylinder of radius one and is given by [13];

$$\phi = (x+1./x)\ \cos(y) \quad (15)$$

In equation (14) ϕ is represented by p and pnew in iteration process. With the known exact solution, the boundary conditions on a r or x (1,60) and θ or y (-pi,pi) grid are given by,

```
pnew[:,pi] =  -(x+1./x)
pnew[:, -pi] = -(x+1./x)
pnew[1, :] = 2.* cos(y)
pnew[60, :] = (x[-1] +1./x[-1])* cos(y)         (16)
```

where x[-1] refers to outer radial boundary, and the parameters in pnew have been altered for theoretical depiction.

The iterative solution took 0.9531 s and had the accuracy of 96.8175%. The neural network [10] run time was 400 s and accuracy of 91.6%. Presently extending the outer limit of r to higher values and increasing number of grid points in radial direction slightly improves the accuracy. The following figure shows the results for ϕ.

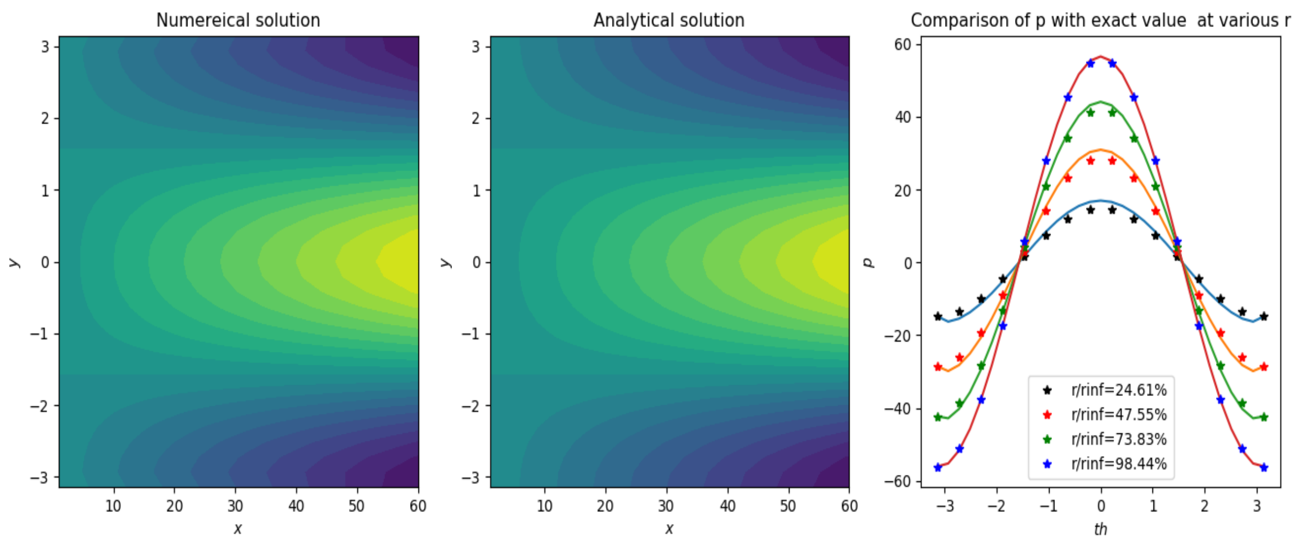

Figure (3). Solution of potential flow over a cylinder

The code for all the iterative analysis has been posted in narain42/Poission-s-Equation-Solver-Using-ML as new iterative_methods_rev.py on Github.

3. Conclusions

The neural network seems to have excellent predictive capability in coupled non-linear differential equations. The free convection from a vertical flat plate results from this method and a method using "fsolve" shows excellent correlation. The extension of finite difference solution to higher dimension shows promising capability. The comparison of two test cases showed that the accuracy in finite difference decreases with higher dimensions. The finite difference runtime efficiency is extremely high in comparison to neural network solution. However, neural network solutions maintain high accuracy in any dimension irrespective of grid dimensions. The potential flow over a cylinder in polar finite difference scheme showed better result compared to neural network solution [10].

REFERENCES

[1] John Kitchin, "/kitchingroup.cheme.cmu.edu/blog/2017/11/27/Solving-BVPs-with-a-neural-network-and-autograd/", 11/27/2017.

[2] J.P. Narain, "Exploring Blasius and Falkner-Skan Equations with Python", American Journal of Fluid Mechanics, Feb 6, 2021.

[3] William M. Deen, "Analysis of Transport Phenomena", Oxford University Press, 1998".

[4] Ostrach,"An analysis of laminar free-convection flow and heat transfer about a flat plate parallel to the direction of the generating body force", NACA Tech. Rept. No 1111, 1953.

[5] Sparrow and Gregg 5 "J. Heat Transfer", 1956, v. 78, p. 435.

[6] Ryan Adams, "autograd", HIPS/autograd on github.com", March 5, 2015.

[7] Diederik P. Kingma, Jimmy B,"Adam: A Method for Stochastic Optimization", arxiv, 2015.

[8] Powell, M. J. D., "A Fortran Subroutine for Solving Systems of Nonlinear Algebraic Equations," *Numerical Methods for Nonlinear Algebraic Equations*, P. Rabinowitz, ed., Ch. 7, 1970.

[9] Moré, J. J., B. S. Garbow, K. E. Hillstrom," *User Guide for MINPACK 1"*, Argonne National Laboratory, Rept. ANL-80-74, 1980.

[10] J.P. Narain," Solution of Poisson's Equation In Higher Dimensions Using Simple Artificial Neural Network", American Journal of Computational and Applied Mathematics, 2021, 11(3): 51-59.

[11] "https://aquaulb.github.io/book_solving_pde_mooc/solving_pde_mooc/notebooks/05_IterativeMethods/05_01_Iteration_and_2D.html"

[12] A. Shiferaw, and R.C. Mittal,"An efficient direct metod to solve the three dimensional Poisson's equation", American Journal of Computational mathematics, December 2011.

[13] A.M. Keuthe and C.Y. Chow., "Foundations of Aerodyanics," Fifth Edition, 1998.