Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Μηχανολόγων Μηχανικών

Υπολογισμός ροής στο εσωτερικό αγωγού μεταβλητής διατομής

Εργασία Προαιρετική (2)
στο μάθημα $7^{ου}$ εξαμήνου:
Υπολογιστική Ρευστομηχανική

Ονοματεπώνυμο: Βάββας Αλέξιος
Πατρώνυμο: Ιωάννης
Α.Μ.: mc20050
Ημ/νία παράδοσης: 07 / 01 / 2024

Αθήνα, Ιανουάριος 2024

Πίνακας Περιεχομένων

GitHub Repository:

https://github.com/AlexiosVavvas/cfd_variable_diam_pipe_2o_proairetiko

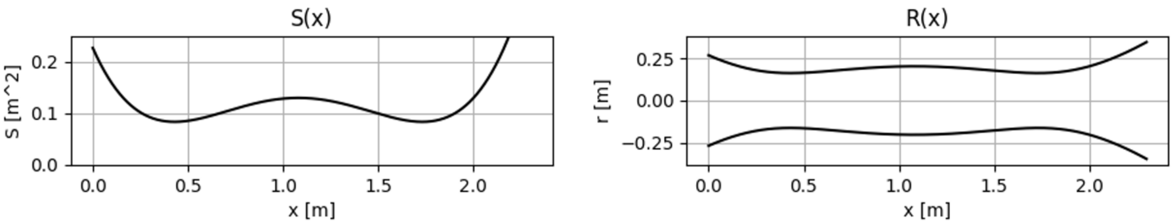
Κατάλογος Διαγραμμάτων

Εισαγωγή

Σκοπός του παρόντος, όπως θα αναλυθεί και στη συνέχεια, είναι η μελέτη ροής στο εσωτερικό αγωγού μεταβλητής διατομής με στενώσεις. Για να επιτευχθεί αυτό γράφηκε κώδικας βασισμένος στη μέθοδο των χαρακτηριστικών και τον προσεγγιστικό επιλύτη του Roe. Ο κώδικας είναι γραμμένος σε python για 2 λόγους. Αφενός για ευκολότερη και γρηγορότερη συγγραφή του, και αφετέρου για λόγους οπτικούς αφού ζητούνται πολλά διαγράμματα και βίντεο με την εξέλιξη της ροής, πράγματα των οποίων η δημιουργία είναι μακράν ευκολότερη εκεί. Αυτό που ακολουθεί είναι μια παρουσίαση της μεθοδολογίας που ακολουθήθηκε, περιγραφή της γενικής ιδέας του αλγορίθμου με διαγράμματα ροής και τέλος παρουσίαση και σχολιασμός των αποτελεσμάτων.

Περιγραφή Προβλήματος

Σκοπός μας είναι ο υπολογισμός της ροής στο εσωτερικό αγωγού μεταβλητής διατομής με 2 στενώσεις, παραδείγματα των οποίων φαίνονται στα παρακάτω διαγράμματα:



Εικόνα 1: Διαγραμματική απεικόνιση εμβαδού διατομής και ακτίνα συναρτήσει της απόστασης x

Το ρευστό θεωρείται συμπιεστό (ιδανικό αέριο) και η ροή ατριβής. Επίσης θεωρούμε ότι οι κατανομές των ρευστομηχανικών μεγεθών σε κάθε διατομή είναι σταθερές οπότε μπορεί να λυθεί το πρόβλημα ως μονοδιάστατο (με χωρική ανεξάρτητη μεταβλητή την συντεταγμένη x). Το εμβαδό της διατομής S μεταβάλλεται με την απόσταση x και η εξίσωση για αυτό δίνεται από το παρακάτω:

$$S(x) = k + a\ \ y^2\ (y^2 - b^2)$$
$$y = x - c$$

Όπου οι σταθερές προκύπτουν από τον αριθμό μητρώου του φοιτητή και είναι ως ακολούθως:

$$k = 0.13$$

$$a = 0.26$$

$$b = 0.92$$

$$c = 1.08$$

Το μήκος του αγωγού επιλέχθηκε αυθαίρετα $2.3\ m$ για να κλείνει όμορφα η άκρη και να περιέχει και το δεύτερο στένωμα. Ακόμα, δεδομένα αποτελούν και οι οριακές συνθήκες για τις οποίες ισχύουν τα ακόλουθα:

$$(P_0, T_0)_{\alpha\varepsilon\rho o\varphi\upsilon\lambda\alpha\kappa\acute{\iota}o\upsilon} = (3.7\ bar, 275\ K)$$

$$P_{out} = 1.5\ bar$$

$$T_{atm} = 273.15\ K$$

Οι εξισώσεις που θα λύσουμε έχουν ως εξής:

$$\frac{\partial}{\partial t}\begin{pmatrix}\tilde{\rho}\\ \tilde{\rho}u\\ \tilde{\rho}E\end{pmatrix} + \frac{\partial}{\partial x}\begin{pmatrix}\tilde{\rho}u\\ \tilde{\rho}u^2 + \tilde{p}\\ \tilde{\rho}Hu\end{pmatrix} = \begin{pmatrix}0\\ \dfrac{\tilde{p}}{S}\dfrac{dS}{dx}\\ 0\end{pmatrix}, \qquad \tilde{\rho} = \rho\ S, \tilde{p} = p\ S$$

Το σχήμα αυτό είναι γνωστό ως η συντηρητική μορφή των εξισώσεων. Διαιρώντας παντού με τη διατομή S, μπορούμε να απλοποιήσουμε τα πράγματα και να καταλήξει η έκφραση ως εξής:

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho u\\ \rho E\end{pmatrix}+\frac{\partial}{\partial x}\begin{pmatrix}\rho u\\ \rho u^2+p\\ \rho H u\end{pmatrix}=\begin{pmatrix}0\\ \frac{p}{S}\frac{dS}{dx}\\ 0\end{pmatrix}$$

Η οποία και είναι μία απ' τις μορφές που θα χρησιμοποιηθούν στη συνέχεια, και θα αναφερόμαστε σε αυτή ως η συντηρητική. Ακόμα υπάρχει η πρωταρχική / πρωτογενής μορφή:

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ u\\ p\end{pmatrix}+\begin{pmatrix}u & \rho & 0\\ 0 & u & \frac{1}{\rho}\\ 0 & \rho\, c^2 & u\end{pmatrix}\cdot\frac{\partial}{\partial x}\begin{pmatrix}\rho\\ u\\ p\end{pmatrix}=\begin{pmatrix}-\frac{\rho u}{S}\frac{dS}{dx}\\ 0\\ -\frac{\rho u c^2}{S}\frac{dS}{dx}\end{pmatrix}$$

Η συγκεκριμένη διευκολύνει την ανάλυση ιδιοτιμών:

$$\tilde{A}=\begin{pmatrix}u & \rho & 0\\ 0 & u & \frac{1}{\rho}\\ 0 & \rho\, c^2 & u\end{pmatrix}\ \Rightarrow \begin{pmatrix}\lambda_1\\ \lambda_2\\ \lambda_3\end{pmatrix}=\begin{pmatrix}u\\ u+c\\ u-c\end{pmatrix}$$

Επομένως, με βάση τα παραπάνω ορίζουμε:

$$u_C=\{u_{c1}\quad u_{c2}\quad u_{c3}\}^T=\{\rho\quad \rho u\quad \rho E\}^T$$

$$u_P=\{u_{p1}\quad u_{p2}\quad u_{p3}\}^T=\{\rho\quad u\quad p\}^T$$

Προφανώς μετατρέπουμε από τη μια γραφή στην άλλη ως έχει:

$$u_{c1}=u_{p1}$$

$$u_{c2}=u_{p1}\cdot u_{p2}$$

$$u_{c3}=\frac{u_{p3}}{\gamma-1}+\frac{1}{2}\,u_{p1}\,u_{p2}^2$$

και

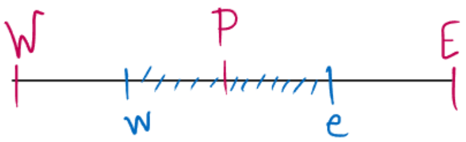
$$u_{p1}=u_{c1}$$

$$u_{p2}=\frac{u_{c2}}{u_{c1}}$$

$$u_{p3}=(\gamma-1)\left(u_{c3}-\frac{1}{2}\frac{u_{c2}^2}{u_{c1}}\right)$$

Διακριτοποίηση χωρίου και σκιαγράφηση πορείας

Για τη διακριτοποίηση του χωρίου και την επίλυση των εξισώσεων θα χρησιμοποιήσουμε τη μέθοδο των πεπερασμένων όγκων.



Αρχική εξίσωση:

$$\frac{\partial U}{\partial t} + \frac{\partial F(U)}{\partial x} = Q(U)$$

Ολοκληρώνοντας στο παραπάνω σκιαγραμμισμένο χωρίο παίρνουμε:

$$\int \frac{\partial U}{\partial t} dx + \int \frac{\partial F(U)}{\partial x} dx = \int Q(U)\, dx$$

$$\Delta x\, \frac{\partial \bar{U}_\iota}{\partial t} + F|_{i+\frac{1}{2}} - F|_{i-\frac{1}{2}} = \Delta x\, \bar{Q}_\iota$$

$$\Delta x\, \frac{\partial \bar{U}_\iota}{\partial t} + F_e - F_w = \Delta x\, \bar{Q}_\iota$$

Ορίζω:

$$R(U_i) = F_e - F_w$$

Και έχουμε:

$$\Delta x\, \frac{\partial \bar{U}_\iota}{\partial t} + R(U_i) = \Delta x\, \bar{Q}_\iota$$

$$\frac{\partial \bar{U}_\iota}{\partial t} = \bar{Q}_\iota - \frac{R(U_i)}{\Delta x}$$

Με

$$F_e = \frac{1}{2}\,(F_P + F_E) - \frac{1}{2}\,|A_e|\,(U_P - U_E), \qquad |A_e| = R_e|\Lambda_e|R_e^{-1}$$

$$F_w = \frac{1}{2}\,(F_W + F_P) - \frac{1}{2}\,|A_w|\,(U_W - U_P), \qquad |A_w| = R_w|\Lambda_w|R_w^{-1}$$

Όπως θα δούμε αμέσως.

Υπολογισμός παροχών (Fluxes) στην επιφάνεια

Για τον υπολογισμό του F σε κάποιο face, έχουμε:

$$F_{face} = \frac{1}{2}(F_L + F_R) - \frac{1}{2}\ |A| \cdot (U_L - U_R)$$

$$F_{face} = \frac{1}{2}(F_L + F_R) - \frac{1}{2}\ R\ |\Lambda|\ L \cdot (U_L - U_R)$$

Όπου,

$$L = \begin{pmatrix} 1 - \frac{1}{2}(\gamma-1)\frac{u_x^2}{c_x^2} & (\gamma-1)\frac{u_x}{c_x^2} & -\frac{\gamma-1}{c_x^2} \\ \left(\frac{1}{2}\ (\gamma-1)\ u_x^2 - u_x \cdot c_x\right)\frac{1}{\rho_x\ c_x} & (c_x - (\gamma-1)\ u_x)\frac{1}{\rho_x\ c_x} & \frac{\gamma-1}{\rho_x\ c_x} \\ -\left(\frac{1}{2}\ (\gamma-1)\ u_x^2 + u_x \cdot c_x\right)\frac{1}{\rho_x\ c_x} & (c_x + (\gamma-1)\ u_x)\frac{1}{\rho_x\ c_x} & -\frac{\gamma-1}{\rho_x\ c_x} \end{pmatrix}$$

$$R = \begin{pmatrix} 1 & \frac{\rho_x}{2\ c_x} & -\frac{\rho_x}{2\ c_x} \\ u_x & \frac{1}{2}(u_x + c_x)\frac{\rho_x}{c_x} & -\frac{1}{2}(u_x - c_x)\frac{\rho_x}{c_x} \\ \frac{1}{2}u_x^2 & \left(\frac{1}{2}\ u_x^2 + u_x\ c_x + \frac{c_x^2}{\gamma-1}\right)\frac{\rho_x}{2\ c_x} & -\left(\frac{1}{2}\ u_x^2 - u_x\ c_x + \frac{c_x^2}{\gamma-1}\right)\frac{\rho_x}{2\ c_x} \end{pmatrix}$$

Όπου ρ_x, u_x, c_x, αντικαθιστούμε με:

$$\rho_x = \sqrt{\rho_L} \cdot \sqrt{\rho_R}$$

$$u_x = \frac{u_L\sqrt{\rho_L} + u_R\sqrt{\rho_R}}{\sqrt{\rho_L} + \sqrt{\rho_R}}$$

$$H_x = \frac{H_L\sqrt{\rho_L} + H_R\sqrt{\rho_R}}{\sqrt{\rho_L} + \sqrt{\rho_R}}$$

$$c_x = \sqrt{(\gamma-1)(H_x - 0.5\ u_x^2)}$$

Και

$$|\Lambda| = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix} = \begin{pmatrix} u_x & 0 & 0 \\ 0 & u_x + c_x & 0 \\ 0 & 0 & u_x - c_x \end{pmatrix}$$

Runge – Kutta για βηματισμό στον χρόνο

Τώρα που έχουμε τρόπο να υπολογίζουμε τις παροχές στα σύνορα, άρα και χρονικές παραγώγους μέσω της εξίσωσης μας, μπορούμε να μεταβούμε από τη μια χρονική στιγμή στην άλλη. Για να το πετύχουμε αυτό χρησιμοποιούμε το παρακάτω σχήμα RK, για το οποίο θεωρούμε τις σταθερές:

$$rk[1:4] = [0.1084, 0.2602, 0.5052, 1]$$

Όπου μια εξίσωση:

$$\frac{dy}{dt} = f(y)$$

Επιλύεται 4 φορές με αρχική συνθήκη $y_0 = y(t)$:

$$y_k = y_0 + dt \cdot rk(k) \cdot f(y_{k-1})$$

Συγκεκριμένα,

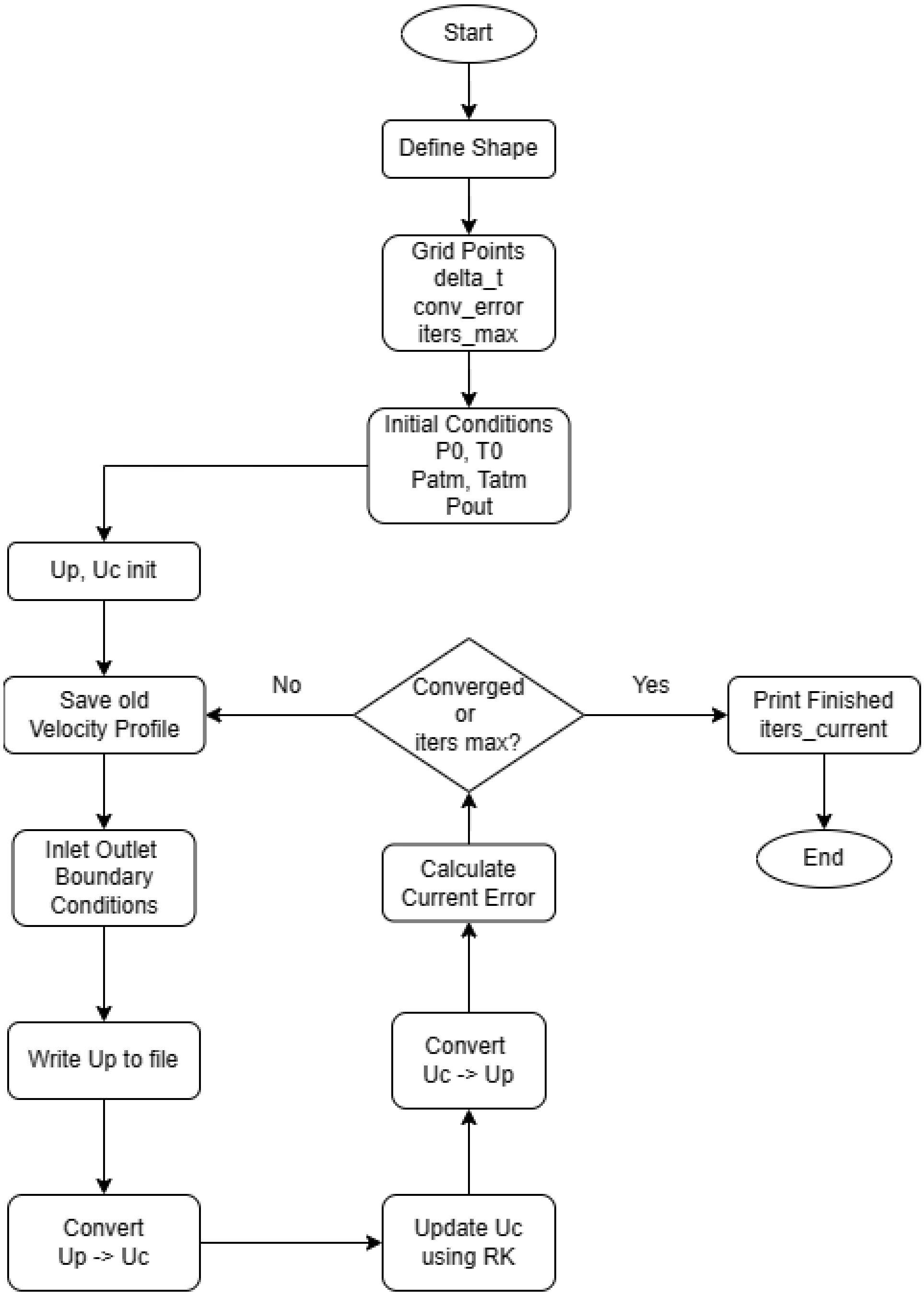
$$\left.\frac{\partial \overrightarrow{U_c}}{\partial t}\right|_k = \left.\overrightarrow{U_c}\right|_{k=0} + \Delta t \cdot rk(k) \cdot \left(\vec{Q}\left(\overrightarrow{U_c}\right) - \frac{\vec{R}\left(\overrightarrow{U_c}\right)}{\Delta x} \right)$$

Η εφαρμογή του συγκεκριμένου κομματιού σε python φαίνεται εδώ:

```python
# U_C : 3xN
def customRKstep(U_C, delta_x, delta_t, a, b, c, k):

    # Define rk constants
    rk = [0.1084, 0.2602, 0.5052, 1]

    N = U_C.shape[1]
    U_ = np.copy(U_C)    # Resulting U_C

    # For every RK constant
    for j in range(len(rk)):
        U_P_ = ucToUp(U_)

        # For every internal node
        for i in range(1, N-1):
            U_[:, i] = U_C[:, i] + delta_t * rk[j] *
                  (calcQ(i, U_P_, delta_x, a, b, c, k) - calcR(i, U_P_)/delta_x)

    return U_  # U_C

```

Αλγόριθμος επίλυσης

Πριν ξεκινήσουμε την παρουσίαση του κώδικα και των επιμέρους του στοιχείων, ίσως έχει νόημα να παρατηρήσουμε λίγο σε ένα διάγραμμα ροής την αλληλουχία διαδικασιών που πραγματοποιούμε.



Εικόνα 2: Διάγραμμα ροής στο οποίο φαίνεται το σκεπτικό πίσω απ' το κυρίως κομμάτι κώδικα

Παρουσίαση Αποτελεσμάτων

Σύμφωνα με τον κανόνα για εύρεση αρχικών συνθηκών που είχε δοθεί και τον αριθμό μητρώου μου, η πίεση αεροφυλακίου θα ήταν 0 bar. Με αυτό προέκυπτε πρόβλημα καθώς βγαίναν μετά από κάνα δυο επαναλήψεις διάφορα υπόριζα αρνητικά. Για τον λόγο αυτό και αποφάσισα να αυξήσω την πίεση εισόδου καθώς και την διαφορά πίεσης αεροφυλακίου από το 0.7 bar στα παρακάτω. Συνολικά γίναν αρκετές δοκιμές. Οι ουσιαστικότερες ήταν 2, μια για μήκος αγωγού 2.2 m και μία για μήκος 1 m.

Μια στένωση στον αγωγό

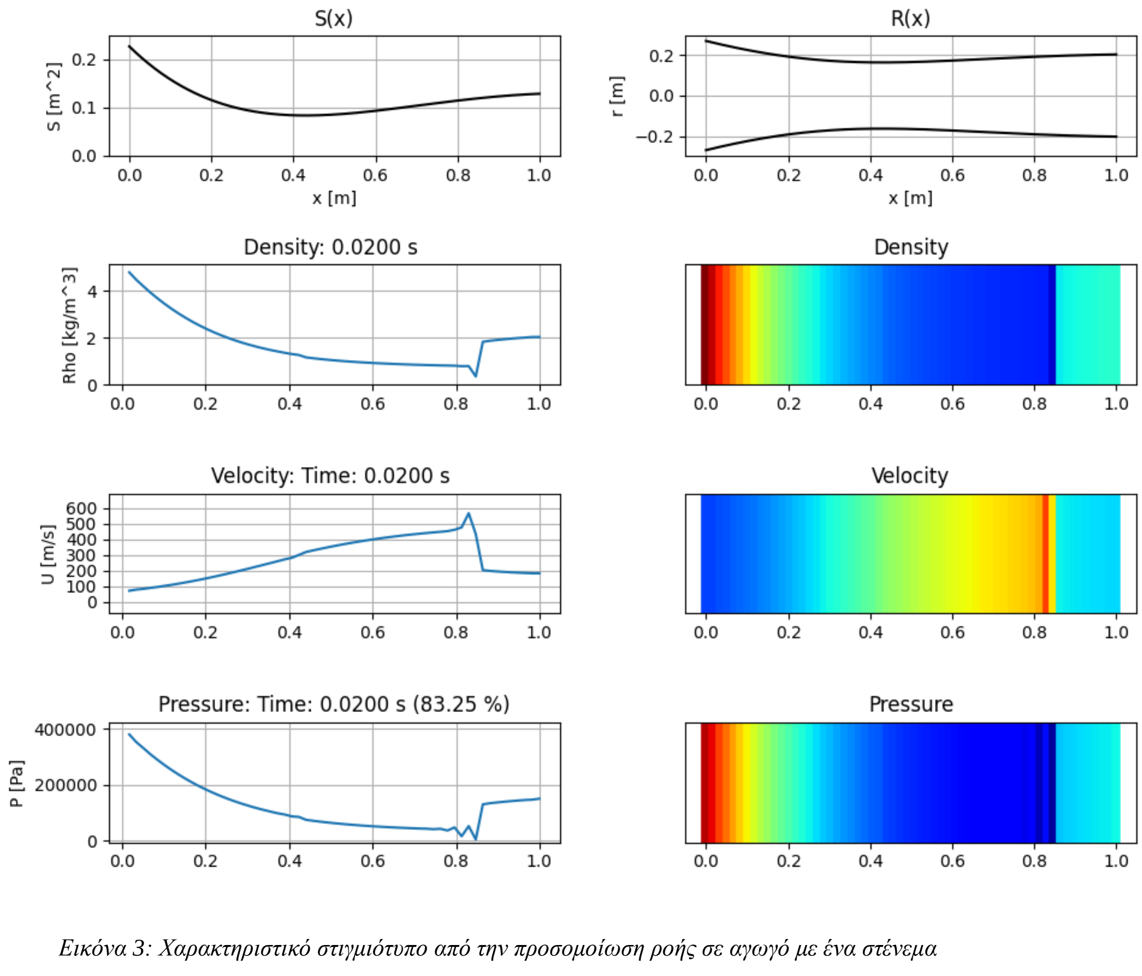
Αρχικές Συνθήκες:

$$P_0 = 4\ bar, \qquad T_0 = 275\ K$$

$$P_{atm} = P_{out} = 1.5\ bar, \qquad T_{atm} = 273.15\ K$$

$$\Delta t = 10^{-6}\ sec$$

$$\#\kappa ó\mu\beta\omega\nu = 70$$



Εικόνα 3: Χαρακτηριστικό στιγμιότυπο από την προσομοίωση ροής σε αγωγό με ένα στένεμα

Στο παραπάνω σχήμα βλέπουμε ξεκάθαρα τις συνθήκες που επικρατούν στο εσωτερικό του αγωγού με ένα στένεμα (ίδια μορφή με τον επόμενο αλλά κομμένο νωρίτερα), σε φάση πολύ κοντά στη μόνιμη κατάσταση. Παρατηρούμε ξεκάθαρα αναπτυγμένη ροή με εμφάνιση κρουστικού κύματος στη περιοχή ανάμεσα στην έξοδο και στον λαιμό. Παρατηρούμε επίσης πως επιβεβαιώνεται το γεγονός ότι η ροή στον λαιμό γίνεται ηχητική πιάνοντας Mach ίσο με τη μονάδα.

Αυτό που αξίζει να σημειωθεί είναι η αντίφαση μεταξύ του αριθμού των κόμβων που καταλήξαμε να χρησιμοποιούμε και του χρονικού βήματος. Παρατηρήσαμε πως αυξάνοντας το πλήθος των κόμβων σε πάνω από 150 ο κώδικας δυσκολευόταν αρκετά βηματίσει στον χρόνο. Για τον λόγο αυτό και βρήκαμε πως μια επιλογή ανάμεσα στο 50 και στο 100 έδινε τα κατάλληλα αποτελέσματα με σχετικά ικανοποιητική ταχύτητα. Ακόμα, στην αρχή μας έκανε εντύπωση το γεγονός ότι για να τρέξει απαιτούσε πολύ μικρό χρονικό βήμα της τάξης του 10^{-6}. Λογικό δεδομένου ότι το φαινόμενο που παρατηρούμε εξελίσσεται ραγδαία. Χρειαζόταν κατά μέσο όρο 20 ms για να πιάσει μόνιμη κατάσταση (αρκεί η διαφορά πίεσης να μην ήταν μεγαλύτερη από 5 με 6 bar όπου και παρατηρούσαμε ταλαντώσεις, περισσότερα για τις οποίες θα πούμε στη συνέχεια).

Δυο διαδοχικές στενώσεις

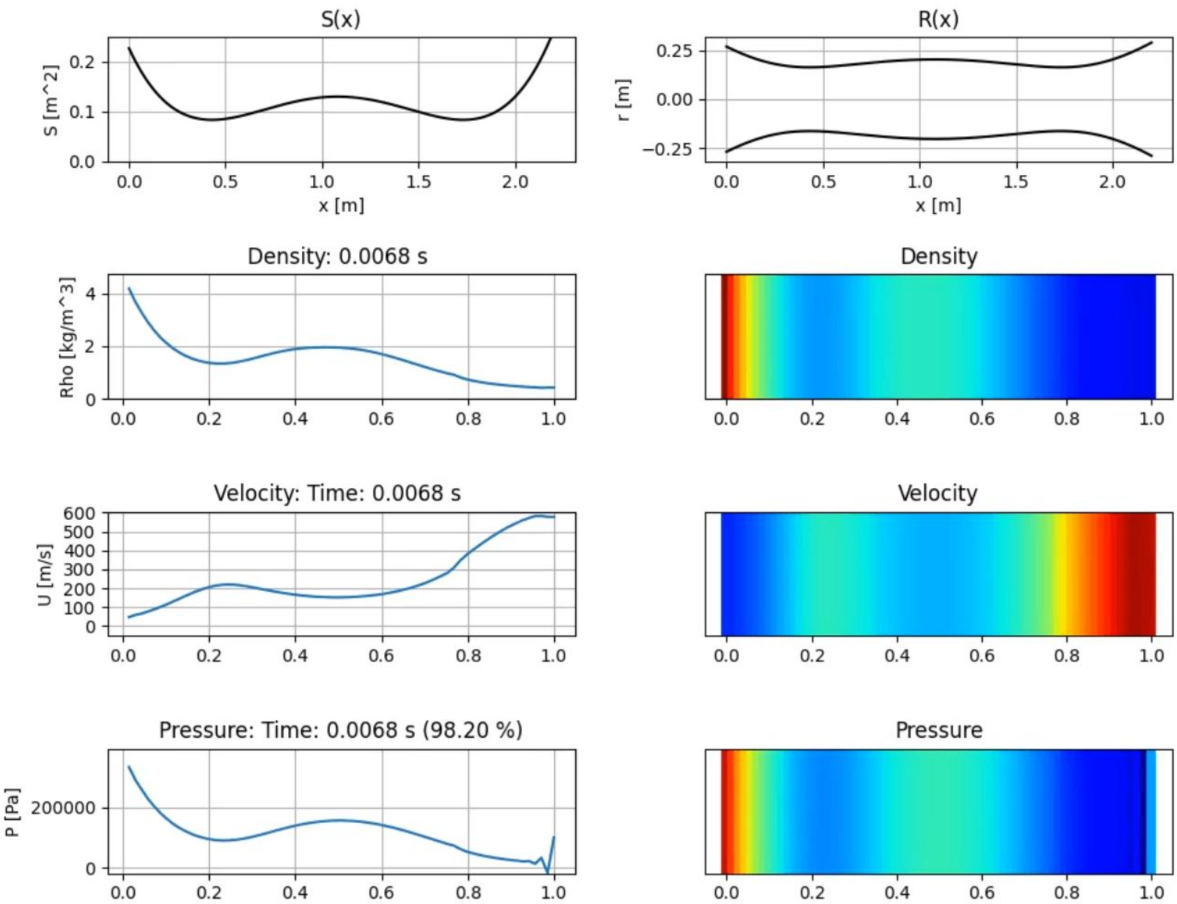
Αρχικές συνθήκες και σημαντικά μεγέθη ίδια με πριν. Συγκρίνουμε σχήμα αγωγού:

$$P_0 = 4\ bar, \qquad T_0 = 275\ K$$

$$P_{atm} = P_{out} = 1.5\ bar, \qquad T_{atm} = 273.15\ K$$

$$\Delta t = 10^{-6}\ sec$$

$$\#κόμβων = 70$$



Εικόνα 4: Χαρακτηριστικό στιγμιότυπο από την προσομοίωση ροής σε αγωγό με δυο διαδοχικά στενέματα

Και στις 2 περιπτώσεις, στην αρχή του χρόνου υπάρχει μια μεγάλη ασυνέχεια στον πρώτο κόμβο αμέσως μετά το αεροφυλάκιο. Εκεί, από τη μια μεριά η πίεση είναι αυτή του αεροφυλακίου, ενώ αμέσως μετά υπάρχει η εξόδου κατά μήκος όλου του αγωγού. Επομένως με το άνοιγμα της βάνας παρατηρούμε την πίεση αμέσως να αυξάνεται και ένα κύμα ταχύτητας να διαδίδεται. Η πίεση με την ταχύτητα κυμαίνονται ουσιαστικά ανάποδα όπως είναι λογικό, αυξάνεται τοπικά η μία, μειώνεται η άλλη. Ηχητική, αν μπορεί, η ροή γίνεται στον εκάστοτε λαιμό και μετά είτε επιβραδύνει είτε επιταχύνει αναλόγως με το αν τα κατάφερε να πιάσει Mach ίσο με 1. Μετά από μερικά κύματα μπρος πίσω, η ροή ισορροπεί και φτάνουμε στο προφίλ που παρατηρούμε παραπάνω σε μόνιμη κατάσταση με κύμα κρούσης ή χωρίς.

Παράρτημα – Κώδικες

Ορισμός μεταβλητών σχήματος

```
# S(x) = k + a * x_^2 * (x_^2 - b^2)
# x_ = x - c

# Shape Constants Definition
k = 0.13
a = 2*k
b = 0.92
c = 2-b

L = 2.2    # [m] - Length of the tube

```

Βασική Επίλυση Ροής

```python
import numpy as np
from myFunctions import *
from constantsToUse import *

# Create folders if they don't exist
checkOrCreateFolderResults("results")
checkOrCreateFolderResults("results/current_run_csvs")
checkOrCreateFolderResults("results/animation")

# Remove any previous results
os.system("rm results/current_run_csvs/*.csv")

# -------------------------------
N = 70                        # Number of grid points
delta_t = 1e-6                # Time step
SKIP_FRAMES = 30              # Number of frames to skip when saving animation
SKIP_PRINTS = 10              # Number of frames to skip when saving animation
CONV_E = 3e-2                 # Convergence criteria
N_TIME_STEPS_MAX = 50000      # Number of time steps to solve./

# Plot S(x), R(x) and Save them
plotAndSaveS(a, b, c, k, L)

# Initial Conditions
P0 = 4e5                # [Pa] - Aerofilakio
T0 = 275                # [K]  - Aerofilakio
Pout = 1.5e5            # [Pa] - Outlet
Patm = Pout             # [Pa] - Atmospheric
Tatm = 273.15           # [K]  - Atmospheric
# -------------------------------

# Grid Definition
delta_x = L / (N - 1)

# General Initialization
U_P = np.zeros((3, N))
U_C = np.zeros((3, N))
U_u_old = np.zeros((1, N))      # u at previous time step

U_P[0, :] = Patm/ (287 * Tatm)  # Initial Density - Ideal Gas Law
U_P[1, :] = 0                   # Initial Velocity
U_P[2, :] = Patm                # Initial Pressure

# Boundary Conditions
U_P[0, 0] = P0 / (287 * T0)  # rho 0
U_P[1, 0] = U_P[1, 1]        # u 0
U_P[2, 0] = P0               # p 0

U_P[0, N-1] = U_P[0, N-2]    # rho Outlet
U_P[1, N-1] = U_P[1, N-2]    # u Outlet
U_P[2, N-1] = Pout           # p Outlet

# Time Loop
t = 0
i = 0

# Save initial state, initial conditions and shape constants
np.savetxt(f"results/current_run_csvs/U_P_init.csv", np.hstack((np.linspace(0, 1, N).reshape(N, 1), np.transpose(U_P))), delimiter=",")
np.savetxt("results/INIT_CONDITIONS.csv", np.array([["P0", "T0", "Pout"], [P0, T0, Pout]]), delimiter=",", fmt="%s")
np.savetxt("results/SHAPE_CONSTS.csv", np.array([["k", "a", "b", "c", "L"], [k, a, b, c, L]]), delimiter=",", fmt="%s")

```

```python
# Solving in Time
conv_error = 1
np.savetxt(f"results/N_TIME_STEPS.csv",
   np.array([N_TIME_STEPS_MAX, delta_t, SKIP_FRAMES]), delimiter=",")

print("Solving in Time...")
while i < N_TIME_STEPS_MAX and (conv_error > CONV_E or i % SKIP_FRAMES != 0):

    # Print progress
    if i % SKIP_PRINTS == 0:
        print(f"Solving Time Step {i} of {N_TIME_STEPS_MAX}
                 at t = {t:.6f} \t {i/N_TIME_STEPS_MAX * 100:.2f}% \t \
                 conv_error = {conv_error:.2e}/{CONV_E:.1e} ->
                 {(CONV_E)/conv_error * 100:.2f}% ")

    # Save old Velocity Profile
    U_u_old = np.copy(U_P[1, :])

    # Boundary Conditions
    # Inlet
    U_P[0, 0] = P0 / (287 * T0)  # rho 0
    U_P[1, 0] = U_P[1, 1]        # u 0
    U_P[2, 0] = P0               # p 0
    # Outlet
    U_P[0, N-1] = U_P[0, N-2]    # rho Outlet
    U_P[1, N-1] = U_P[1, N-2]    # u Outlet
    U_P[2, N-1] = Pout           # p Outlet

    # Write U_P to file
    if i % SKIP_FRAMES == 0:
        np.savetxt(f"results/current_run_csvs/U_P_timestep_{i}.csv",
        np.hstack((np.linspace(0, 1, N).reshape(N, 1), np.transpose(U_P))), delimiter=",")

    # Calculate U_C
    U_C = upToUc(U_P)

    # Update U_C using RK
    U_C = customRKstep(U_C, delta_x, delta_t, a, b, c, k)

    # Convert U_C to U_P
    U_P = ucToUp(U_C)

    # Calc current conv_error
    conv_error = np.max(np.abs(U_P[1, :] - U_u_old))

    # Update time and iteration
    t = t + delta_t
    i = i + 1

np.savetxt(f"results/N_TIME_STEPS.csv", np.array([i, delta_t, SKIP_FRAMES]), delimiter=",")

if i == N_TIME_STEPS_MAX:
    print("Maximum number of time steps reached!")
else:
    print("Convergence reached!")

print(f"Finished at t = {t:.4f}, iter = {i}/{N_TIME_STEPS_MAX}")

```

Βασικές συναρτήσεις που χρησιμοποιήθηκαν

```python
import numpy as np
import matplotlib.pyplot as plt

gamma = 1.4
gamma1 = gamma - 1

'''
Equation:

 dUi      - 1
---- =  -------- * R(U_i) + Q_i
 dt      DeltaX


Uc = [rho, rho*u, rho*E]
Up = [rho, u, p]

The following functions need as input:
    - roeTilda      : uL - uR (uPrime)
    - solveRiemann  : uL - uR (uPrime)
    - F             : Up - 3x1
    - calcR         : Up - 3xN
    - calcQ         : Up - 3xN

'''

# Surface Area
def S(x, a, b, c, k):
    x_ = x - c
    return k + a * x_**2 * (x_**2 - b**2)
def dSdX(x, a, b, c, k):
    x_ = x - c
    return 4 * a * x_**3 - 2 * a * b**2 * x_

def plotAndSaveS(a, b, c, k, L):
    x = np.linspace(0, L, 1000)
    y = S(x, a, b, c, k)

    plt.plot(x, y, 'k-')
    plt.title("Surface Area S(x)")
    plt.grid()
    plt.xlabel("x [m]")
    plt.ylabel("S [m^2]")
    plt.ylim(0, 1.1 * np.max(y))
    plt.savefig("results/S(x).png")
    plt.close()

    r = np.sqrt(y/np.pi)
    plt.plot(x, r, 'k-')
    plt.plot(x, -r, 'k-')
    plt.title("Radius r(x)")
    plt.xlabel("x [m]")
    plt.ylabel("r [m]")
    plt.grid()
    plt.savefig("results/r(x).png")
    plt.close()


```

```python
# Convert from Uc to Up and vice versa
def ucToUp(uc):
    uc = np.atleast_2d(uc).T if uc.ndim == 1 else uc
    up1 = uc[0, :]
    up2 = uc[1, :]/uc[0, :]
    up3 = gamma1 * (uc[2, :] - 0.5 * uc[1, :]**2 / uc[0, :])
    return np.vstack((up1, up2, up3)).squeeze()

def upToUc(up):
    up = np.atleast_2d(up).T if up.ndim == 1 else up
    uc1 = up[0, :]
    uc2 = up[1, :] * up[0, :]
    uc3 = up[2, :] / gamma1 + 0.5 * up[0, :] * up[1, :]**2
    return np.vstack((uc1, uc2, uc3)).squeeze()


# Input upL, upR
# Returns [ρ_, u_, H_, c_]
def roeTilda(uL, uR):
    s_rhoL = np.sqrt(uL[0])
    s_rhoR = np.sqrt(uR[0])

    rho_ = s_rhoL * s_rhoR
    u_ = (s_rhoL * uL[1] + s_rhoR * uR[1]) / (s_rhoL + s_rhoR)

    hL = (upToUc(uL)[2] + uL[2])/uL[0]  # to check if it is correct
    hR = (upToUc(uR)[2] + uR[2])/uR[0]  # to check if it is correct
    H_ = (s_rhoL * hL + s_rhoR * hR) / (s_rhoL + s_rhoR)

    c_ = np.sqrt((gamma - 1) * (H_ - 0.5 * u_**2))

    return np.array([rho_, u_, H_, c_])


# Flux
def F(uP):
    F1 = uP[0] * uP[1]
    F2 = uP[0] * np.abs(uP[1]) * uP[1] + uP[2]
    # F2 = uP[0] * uP[1]**2 + uP[2]
    F3 = (upToUc(uP)[2] + uP[2]) * uP[1]
    return np.array([F1, F2, F3])


# Solve Riemann problem - |A|
# Input uL, uR
def solveRiemann(uL, uR):
    rho_, u_, H_, c_ = roeTilda(uL, uR)

    # Calculate eigenvalues
    lam1 = u_       # to check if it is correct
    lam2 = u_ + c_
    lam3 = u_ - c_
    Lamda = np.diag(np.abs([lam1, lam2, lam3]))

    # Calculate Right eigenvectors 3x3
    R = np.zeros((3,3))
    R[0, 0] = 1
    R[0, 1] = + 0.5 * rho_ / c_
    R[0, 2] = - 0.5 * rho_ / c_
    R[1, 0] = u_
    R[1, 1] = + 0.5 * (u_ + c_) * rho_ / c_
    R[1, 2] = - 0.5 * (u_ - c_) * rho_ / c_
    R[2, 0] = 0.5 * u_**2
    R[2, 1] = + (0.5 * u_**2 + u_*c_ + c_**2 / gamma1) * 0.5 * rho_ / c_
    R[2, 2] = - (0.5 * u_**2 - u_*c_ + c_**2 / gamma1) * 0.5 * rho_ / c_

```

```
    # Calculate Left eigenvectors 3x3
    L = np.zeros((3,3))
    L[0, 0] = 1 - 0.5 * (gamma - 1) * u_**2 / c_**2
    L[0, 1] = gamma1 * u_ / c_**2
    L[0, 2] = - gamma1 / c_**2
    L[1, 0] = + (0.5 * gamma1 * u_**2 - u_*c_) * 1 / (rho_ * c_)
    L[1, 1] = - (gamma1 * u_ - c_) / (rho_ * c_)
    L[1, 2] = gamma1 / (rho_ * c_)
    L[2, 0] = - (0.5 * gamma1 * u_**2 + u_*c_) * 1 / (rho_ * c_)
    L[2, 1] = + (gamma1 * u_ + c_) / (rho_ * c_)
    L[2, 2] = - gamma1 / (rho_ * c_)


    # Calculate resulting |A| -> R * Lamda * L
    A = np.matmul(np.matmul(R, Lamda), L)

    return A

# R = Fe - Fw
# U_P: 3xN
def calcR(i, U_P):
    uP = U_P[:, i]
    uE = U_P[:, i+1]
    uW = U_P[:, i-1]

    # Calculate As
    A_E = solveRiemann(uP, uE)
    A_W = solveRiemann(uW, uP)

    # Calculate Flux
    Fe = 0.5 * (F(uP) + F(uE)) + 0.5 * np.matmul(A_E, (uP - uE))
    Fw = 0.5 * (F(uW) + F(uP)) + 0.5 * np.matmul(A_W, (uW - uP))

    # print(f"iter_{i}: R = {Fe - Fw}")

    return Fe - Fw


# External Q
# U_P = 3xN
def calcQ(i, U_P, delta_x, a, b, c, k):

    p = U_P[2, i]
    S_i = S(i*delta_x, a, b, c, k)
    dSdX_i = dSdX(i*delta_x, a, b, c, k)

    Q = np.zeros(3)
    Q[1] = p / S_i * dSdX_i

    # print(f"iter_{i}: Q2 = {Q[1]}")
    return Q

# U_C : 3xN
def customRKstep(U_C, delta_x, delta_t, a, b, c, k):

    # Define rk constants
    rk = [0.1084, 0.2602, 0.5052, 1]

    N = U_C.shape[1]
    U_ = np.copy(U_C)     # Resulting U_C

    # For every RK constant
    for j in range(len(rk)):
        U_P_ = ucToUp(U_)

        # For every internal node
        for i in range(1, N-1):
            U_[:, i] = U_C[:, i] + delta_t * rk[j] *
                (calcQ(i, U_P_, delta_x, a, b, c, k) - calcR(i, U_P_)/delta_x)
     return U_  # U_C
```

```python

# Resulting Solution Boundaries
def resultingSolBoundaries(folder_filename, N_TIME_STEPS, SKIP_FRAMES):
    rho_low, rho_high = 0, 0
    u_low, u_high = 0, 0
    p_low, p_high = 0, 0

    # Read all files
    for i in range(0, N_TIME_STEPS, SKIP_FRAMES):
        U_P = np.loadtxt(f"{folder_filename}/U_P_timestep_{i}.csv", delimiter=",")
        U_P = np.transpose(U_P[:, 1:])

        rho = U_P[0, :]
        u = U_P[1, :]
        p = U_P[2, :]

        rho_low = np.min([rho_low, np.min(rho)])
        rho_high = np.max([rho_high, np.max(rho)])

        u_low = np.min([u_low, np.min(u)])
        u_high = np.max([u_high, np.max(u)])

        p_low = np.min([p_low, np.min(p)])
        p_high = np.max([p_high, np.max(p)])

    return rho_low, rho_high, u_low, u_high, p_low, p_high


import os
def checkOrCreateFolderResults(folder_path):
    # Check if the folder exists
    if not os.path.exists(folder_path):
        # Create the folder if it does not exist
        os.makedirs(folder_path)
        print(f"Folder {folder_path} created successfully in the current directory.")
    else:
        print(f"Folder {folder_path} already exists in the current directory.")

```