Matrix Portal Flow Visualizer

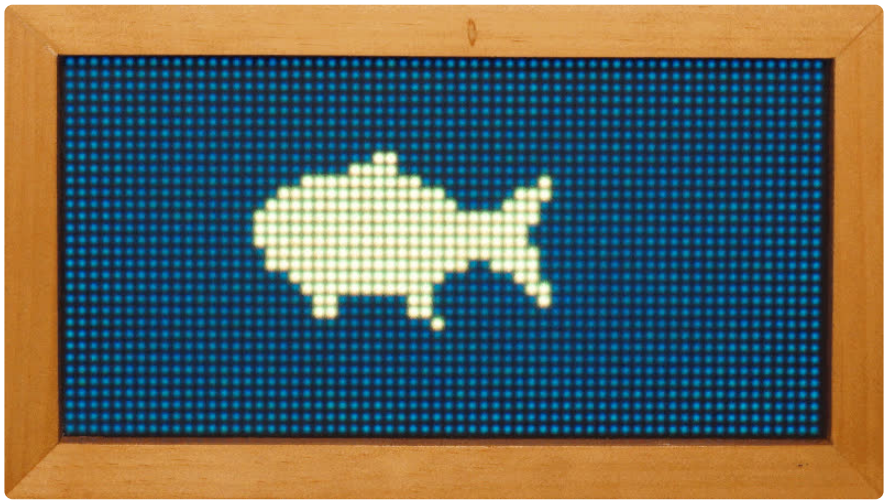
Created by Carter Nelson

https://learn.adafruit.com/matrix-portal-flow-visualizer

Last updated on 2025-02-24 05:59:24 PM EST

Table of Contents

Overview



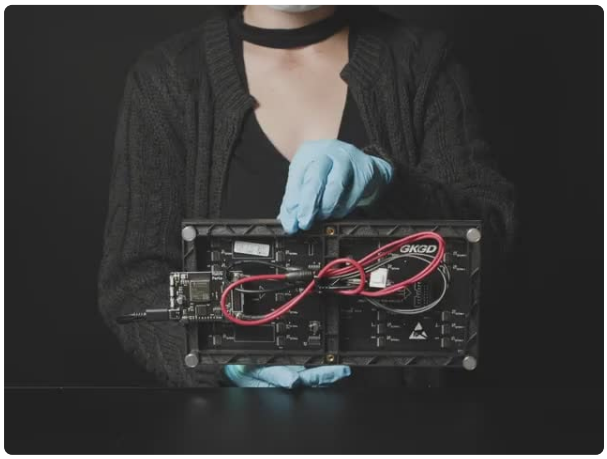
In the field of fluid dynamics, the Navier-Stokes equations (https://adafru.it/OD6) are considered to be the governing equations for fluid flow. They are a mess to deal with directly. However, by ignoring the effects of viscosity (https://adafru.it/OD7) and density changes (compressibility (https://adafru.it/OD8)), the equations reduce down to a much more tractable form known as Laplace's equation (https://adafru.it/19AB), which has applications in more than just fluid dynamics. When used in fluid dynamics, the resulting flow is referred to as potential flow (https://adafru.it/ODa).

Directly trying to solve the Navier-Stokes equations is the realm of high end computing (https://adafru.it/ODb) and PhD level research (https://adafru.it/ODc). By contrast, potential flow problems are commonly assigned as undergraduate homework. Only a minimum amount of computing power is needed. There's even enough power in a little microcontroller, like the ones found on a Matrix Portal, Feathers, and other microcontroller boards.

In this guide, we'll use a Matrix Portal (http://adafru.it/4745) to create a flow field visualizer based on potential flow. We'll cover a couple of different approaches. The goal here isn't to teach aerodynamics though. We'll provide some details, but this is more of a just-for-fun project that can create some fun and interesting animations.

Hardware

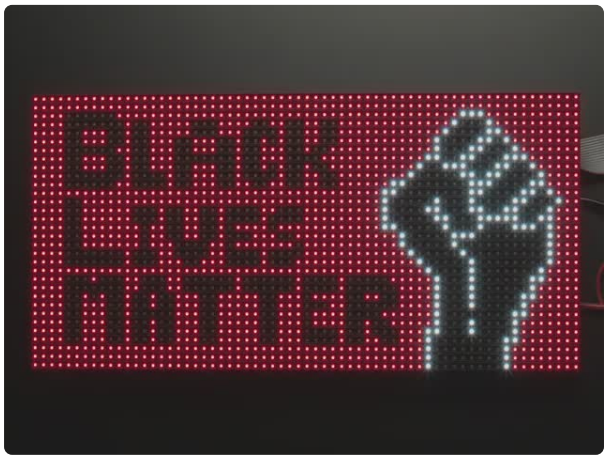
If you have an AdaBox 016 (https://adafru.it/ODd), then you have everything you need for this project. Otherwise, the parts are available a la carte:



Adafruit Matrix Portal - CircuitPython Powered Internet Display

Folks love our wide selection of RGB matrices and accessories, for making custom colorful LED displays... and our RGB Matrix Shields...

https://www.adafruit.com/product/4745



64x32 RGB LED Matrix - 4mm pitch

Bring a little bit of Times Square into your home with this sweet 64 x 32 square RGB LED matrix panel. These panels are normally used to make video walls, here in New York we see them...

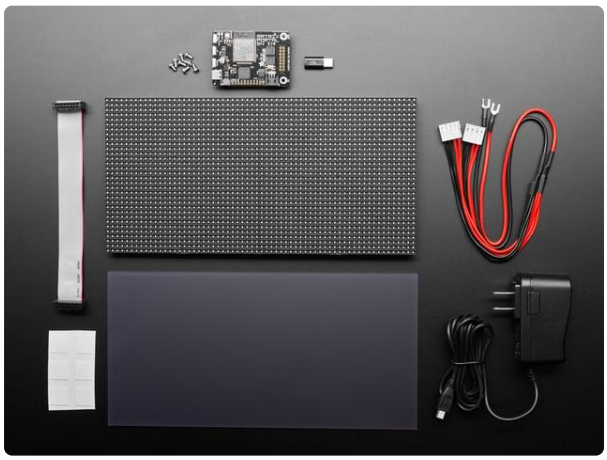
https://www.adafruit.com/product/2278

Black LED Diffusion Acrylic Panel - 10.2" x 5.1"

nice whoppin' rectangular slab of some lovely black acrylic to add some extra diffusion to your LED Matrix project. This material is 2.6mm (0.1") thick and is made of...

https://www.adafruit.com/product/4749

Or as a kit:

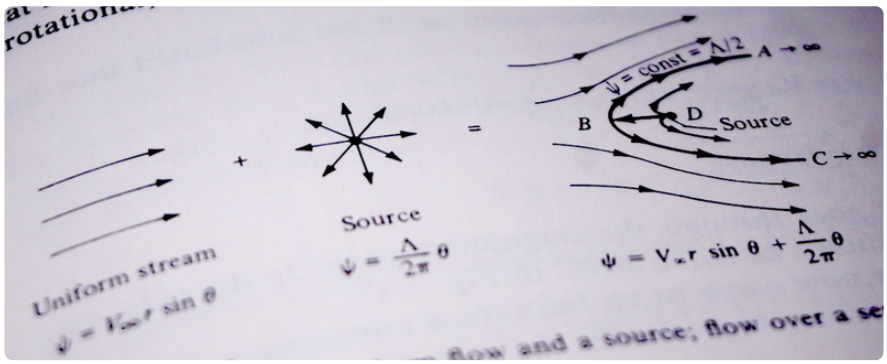


Adafruit Matrix Portal Starter Kit - ADABOX 016 Essentials

If you missed out on AdaBox016, it's not too late for you to pick up the parts necessary to build many of the projects! It...

https://www.adafruit.com/product/4812

Flow Singularities



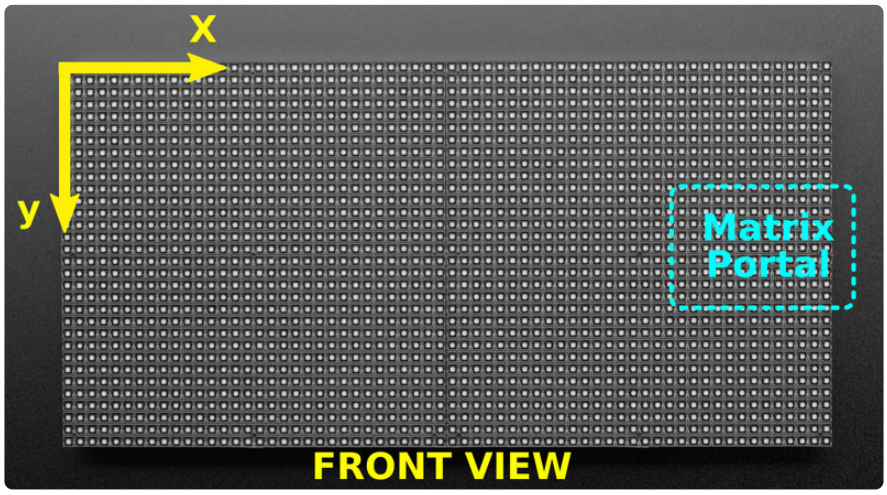
One way to use potential flow is to create a set of basic flow elements. Each of these basic flow elements, often called "singularities", is a specific solution to the governing Laplace's equation. The idea is to use them as building blocks to create and describe various flow fields.

This idea works because if A and B are each a solution to Laplace's equation, then so is A+B. And you can keep adding as many as you want, A+B+C, etc.

So they are sort of like aerodynamic legos. Click them together to create different flow fields.

Coordinate System

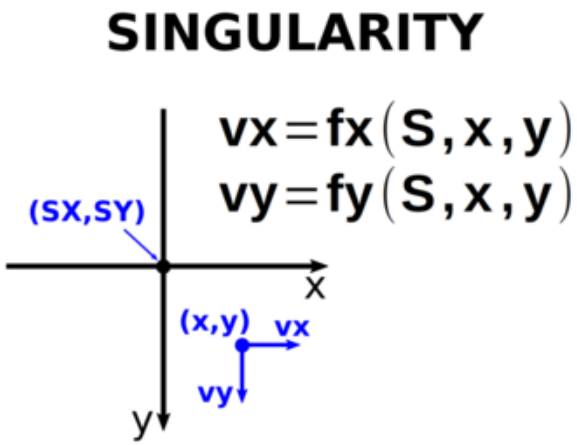
We will work in the native display coordinate system as shown below. This is a little different than the typical coordinate system used in engineering. But these are just conventions. As long as the coordinate system being used is known and accounted for, the results are the same.



We will also stick with just a cartesian notation for everything, even though polar coordinates are cleaner for some of the singularities. Cartesian coordinates will be used in the code as well. So by sticking with cartesian notation, it also makes it easier to understand the code (hopefully).

Singularities

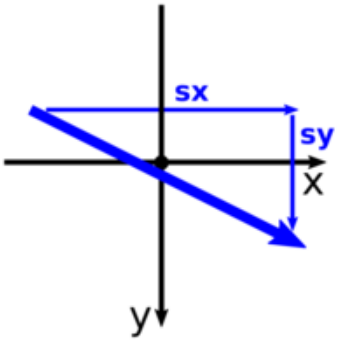
Each singularity has a location **(SX, SY)** and a strength **S**. The velocity **(vx, vy)** induced at some point **(x, y)** is then described by functions **fx(), fy()** specific to the singularity type.



(SX, SY) = singularity location in global coordinate system (i.e., on the matrix)
(x, y) = point relative to singularity location
vx, vy = velocity components induced at point **(x, y)** by singularity

So let's define our basic flow elements. There are four of them.

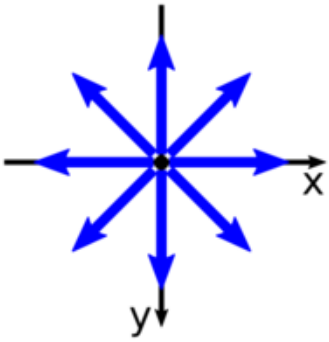
FREESTREAM



$$vx = sx$$
$$vy = sy$$

Freestream flow is just flow everywhere moving in the same direction. It's unique in that it doesn't have a location. Instead, you just define the two components **sx** and **sy**. For "straight" flow, left to right, you'd set **sy=0**, for example.

SOURCE / SINK

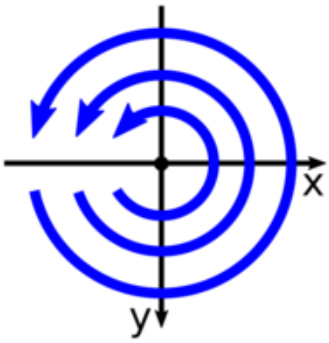


$$vx = S\frac{x}{x^2+y^2}$$
$$vy = S\frac{y}{x^2+y^2}$$

A **source** shoots out in all directions from its origin location. A **sink** is just a source with negative strength, and then all the lines go in instead of out.

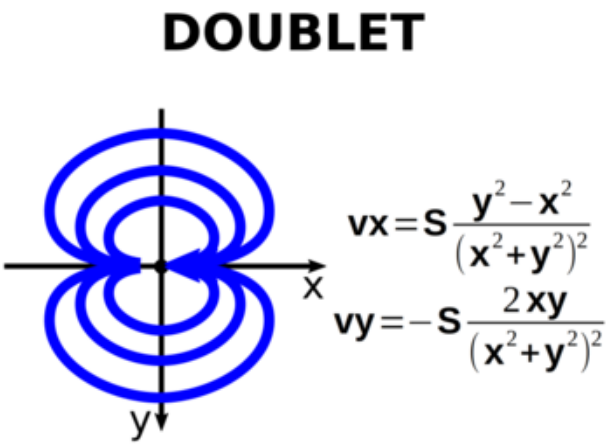
S > 0 = source
S < 0 = sink

VORTEX



$$vx = -S\frac{y}{x^2+y^2}$$
$$vy = S\frac{x}{x^2+y^2}$$

A **vortex** is like a whirlpool. Flow spins in a circle around the origin location. The sign of the strength sets the direction of rotation.



A **doublet** is a source/sink pair brought together so they sit on top of each other. It's a bit of an oddball, but has its use.

And that's our set of aerodynamic legos. Now let's play with them.

Install CircuitPython

CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the **CIRCUITPY** drive to iterate.

Set up CircuitPython Quick Start!

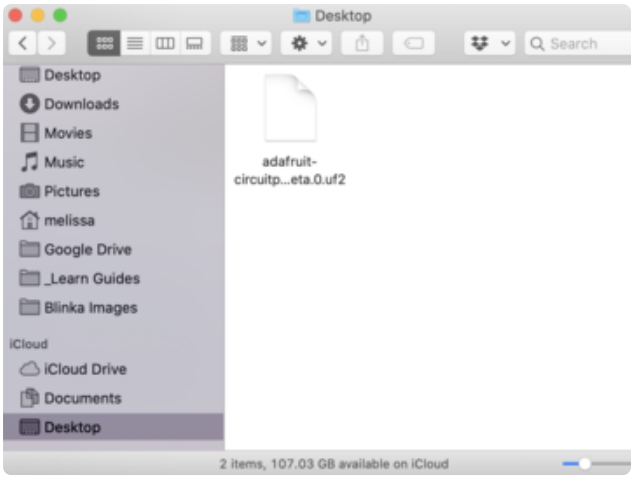
Follow this quick step-by-step for super-fast Python power :)

Download the latest version of CircuitPython for this board via circuitpython.org

https://adafru.it/Nte

Further Information

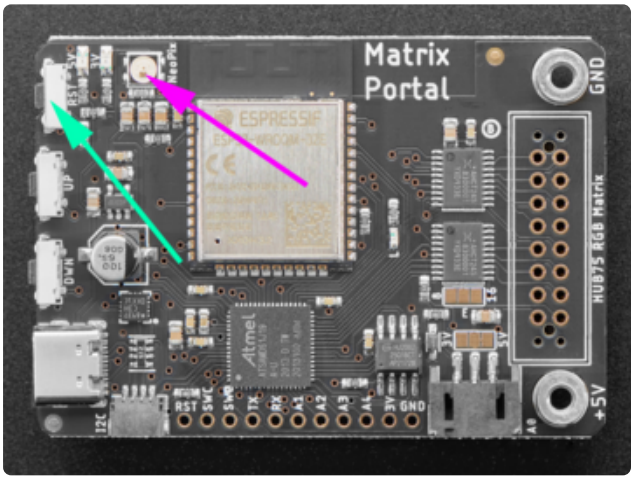
For more detailed info on installing CircuitPython, check out Installing CircuitPython (https://adafru.it/Amd).



Click the link above and download the latest UF2 file.

Download and save it to your desktop (or wherever is handy).

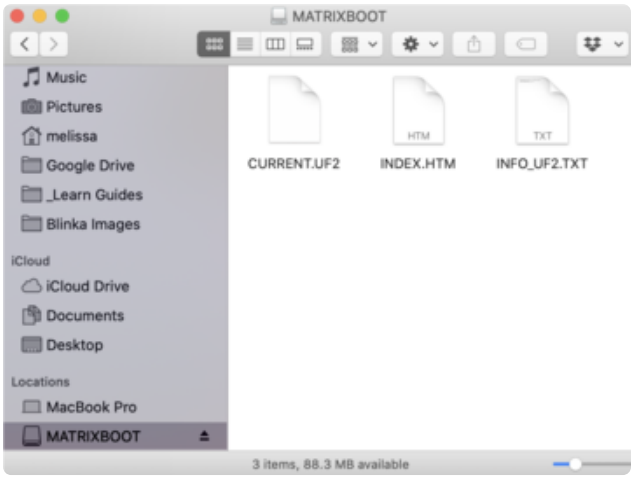
Plug your MatrixPortal M4 into your computer using a known-good USB cable.



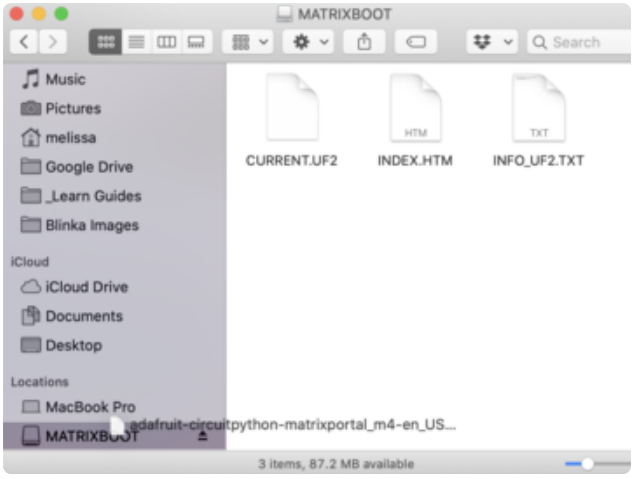
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the **Reset** button (indicated by the green arrow) on your board, and you will see the NeoPixel RGB LED (indicated by the magenta arrow) turn green. If it turns red, check the USB cable, try another USB port, etc.

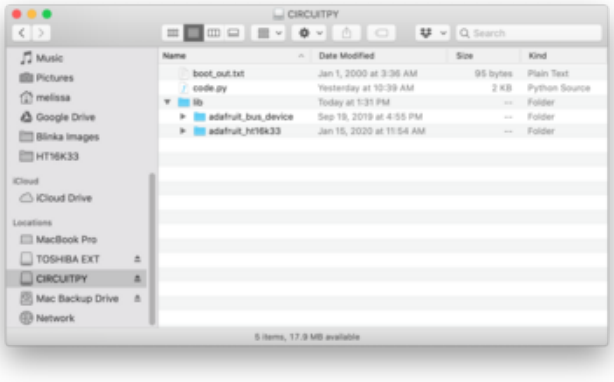
If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!



You will see a new disk drive appear called **MATRIXBOOT**.



Drag the **adafruit_circuitpython_etc.uf2** file to **MATRIXBOOT.**

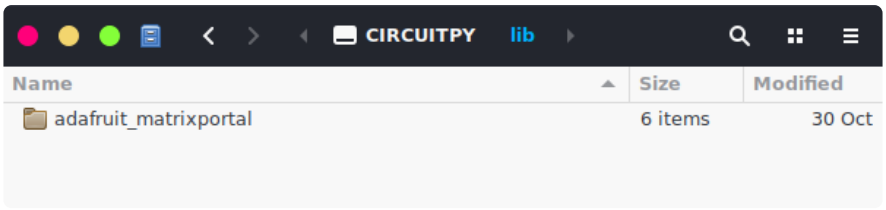


The LED will flash. Then, the **MATRIXBOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

That's it, you're done! :)

Singularity Visualizer

Here's the code for our singularity viewer. It lets you define the type, location, and strength of one or more singularities. To see the results, you specify the starting location for one or more streamlines (https://adafru.it/OE5). Then the program computes the resulting streamlines and animates them.

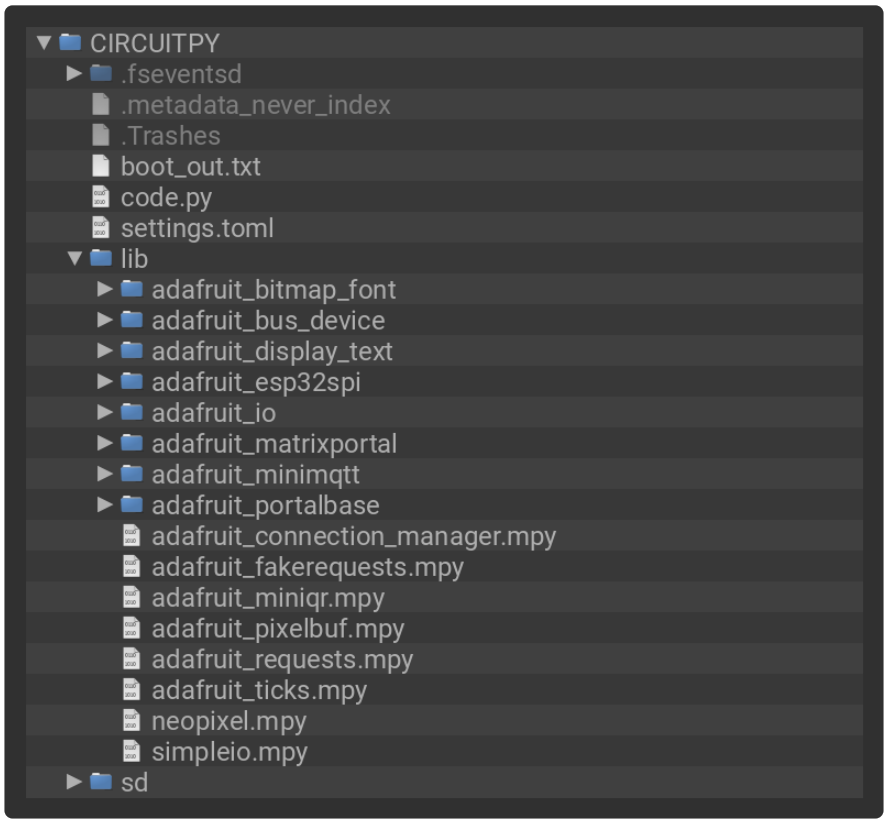


Installing Project Code

To use with CircuitPython, you need to first install a few libraries, into the lib folder on your **CIRCUITPY** drive. Then you need to update **code.py** with the example script.

Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **Matrix_Portal_Flow_Viewer/flow/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2020 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time
import math
import displayio
from adafruit_matrixportal.matrix import Matrix
```

```
#--| User Config |--------------------------------------
SINGULARITIES = (
    # type  location  strength
    ('freestream', None, (1, 0)),
    ('source', (26, 16), 3),
    ('source', (38, 16), -3),
    #('doublet', (32, 16), 1),
    #('vortex', (32, 16), 1),
)
SEEDS = (
#   (x, y) starting location
    (0, 0),
    (0, 3),
    (0, 6),
    (0, 9),
    (0, 12),
    (0, 15),
    (0, 17),
    (0, 20),
    (0, 23),
    (0, 26),
    (0, 29),
)
MATRIX_WIDTH = 64
MATRIX_HEIGHT = 32
BACK_COLOR = 0x000000 # background fill
SING_COLOR = 0xADAF00 # singularities
HEAD_COLOR = 0x00FFFF # leading particles
TAIL_COLOR = 0x000A0A # trailing particles
TAIL_LENGTH = 10      # length in pixels
DELAY = 0.01          # smaller = faster
#-----------------------------------------------------------

# matrix and displayio setup
matrix = Matrix(width=MATRIX_WIDTH, height=MATRIX_HEIGHT, bit_depth=6)
display = matrix.display
group = displayio.Group()
display.root_group = group

bitmap = displayio.Bitmap(display.width, display.height, 4)

palette = displayio.Palette(4)
palette[0] = BACK_COLOR
palette[1] = SING_COLOR
palette[2] = HEAD_COLOR
palette[3] = TAIL_COLOR

tile_grid = displayio.TileGrid(bitmap, pixel_shader=palette)
group.append(tile_grid)

# global to store streamline data
STREAMLINES = []

def compute_velocity(x, y):
    '''Compute resultant velocity induced at (x, y) from all singularities.'''
    vx = vy = 0
    for s in SINGULARITIES:
        if s[0] == 'freestream':
            vx += s[2][0]
            vy += s[2][1]
        else:
            dx = x - s[1][0]
            dy = y - s[1][1]
            r2 = dx*dx + dy*dy
            if s[0] == 'source':
                vx += s[2] * dx / r2
                vy += s[2] * dy / r2
            elif s[0] == 'vortex':
```

```
                vx -=  s[2] * dy / r2
                vy +=  s[2] * dx / r2
            elif s[0] == 'doublet':
                vx += s[2] * (dy*dy - dx*dx) / (r2*r2)
                vy -= s[2] * (2*dx*dy) / (r2*r2)
    return vx, vy

def compute_streamlines():
    '''Compute streamline for each starting point (seed) defined.'''
    for seed in SEEDS:
        streamline = []
        x, y = seed
        px = round(x)
        py = round(y)
        vx, vy = compute_velocity(x, y)
        streamline.append( ((px, py), (vx, vy)) )
        steps = 0
        while x < MATRIX_WIDTH and steps < 2 * MATRIX_WIDTH:
            nx = round(x)
            ny = round(y)
            # if we've moved to a new pixel, store the info
            if nx != px or ny != py:
                streamline.append( ((nx, ny), (vx, vy)) )
                px = nx
                py = ny
            vx, vy = compute_velocity(x, y)
            x += vx
            y += vy
            steps += 1
        # add streamline to global store
        STREAMLINES.append(streamline)

def show_singularities():
    '''Draw the singularites.'''
    for s in SINGULARITIES:
        try:
            x, y = s[1]
            bitmap[round(x), round(y)] = 1
        except: # pylint: disable=bare-except
            pass # just don't draw it

def show_streamlines():
    '''Draw the streamlines.'''
    for sl, head in enumerate(HEADS):
        try:
            streamline = STREAMLINES[sl]
            index = round(head)
            length = min(index, TAIL_LENGTH)
            # draw tail
            for data in streamline[index-length:index]:
                x, y = data[0]
                bitmap[round(x), round(y)] = 3
            # draw head
            bitmap[round(x), round(y)] = 2
        except: # pylint: disable=bare-except
            pass # just don't draw it

def animate_streamlines():
    '''Update the current location (head position) along each streamline.'''
    reset_heads = True
    for sl, head in enumerate(HEADS):
        # get associated streamline
        streamline = STREAMLINES[sl]
        # compute index
        index = round(head)
        # get velocity
        if index < len(streamline):
            vx, vy = streamline[index][1]
            reset_heads = False
```

```
        else:
            vx, vy = streamline[-1][1]
        # move head
        HEADS[sl] += math.sqrt(vx*vx + vy*vy)
    if reset_heads:
        # all streamlines have reached the end, so reset to start
        for index, _ in enumerate(HEADS):
            HEADS[index] = 0

def update_display():
    '''Update the matrix display.'''
    display.auto_refresh = False
    bitmap.fill(0)
    show_singularities()
    show_streamlines()
    display.auto_refresh = True

#==========
# MAIN
#==========
print('Computing streamlines...', end='')
compute_streamlines()
print('DONE')
HEADS = [0]*len(STREAMLINES)
print('Flowing...')
while True:
    animate_streamlines()
    update_display()
    time.sleep(DELAY)
```

How to Use

Look at the top of the code for the section commented as `User Config`. These are the lines you can change to configure the resulting flow field displayed. The main ones are **SINGULARITIES** and **SEEDS**.

- **SINGULARITIES** - Add a tuple containing the type, location, and strength for each singularity you want. You generally always want a **freestream** element, otherwise flow won't "flow" through the display. The available types are:
    - **freestream**
    - **source** (remember, a **sink** is just a source with a negative strength)
    - **vortex**
    - **doublet**
- **SEEDS** - These define the streamlines. Add a tuple of **(x, y)** starting location for each streamline you want.

The code is setup to use a 64x32 matrix size. If you have a different size, change **MATRIX_WIDTH** and **MATRIX_HEIGHT**.

The rest affect the general aesthetics of the flow animation:

- **BACK_COLOR** - Background fill color.

- **SING_COLOR** - Each singularity is shown as a single pixel of this color.
- **HEAD_COLOR** - The pixel color for the leading pixel in a streamline trace.
- **TAIL_COLOR** - The pixel color for the tail of the streamline trace.
- **TAIL_LENGTH** - How many pixels long the streamline trace will be.
- **DELAY** - How fast the animation runs. Lower numbers are faster.

Singularity Strength

So what are good values for the singularity strengths? 1? 50000? We aren't working with any actual units. So you can't set the freestream flow to be 55 mile per hour, for example. You can set any values you want, but you may find the results don't appear or animate well on the actual matrix if you set them too large.

In general, keep things in the single digit range. We'll provide some examples next, so also notice the strength values used in those.

Example Flows

Here are some basic examples that can help get you started. For each one we give you the lines of code to use at the top of the code to define the singularities and the seeds for the streamlines. That way you can just copy-paste them in.

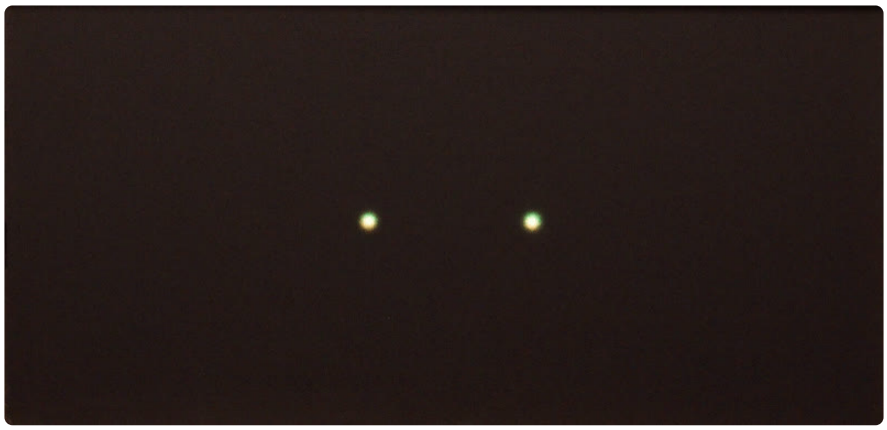
Flow Over an Ovoid

These all called Rankine bodies or Rankine ovals after Macquorn Rankine (https://adafru.it/OE6). The flow is comprised of **freestream** plus a **source** and **sink** pair.

```
#--| User Config |--------------------------------------
SINGULARITIES = (
    # type  location  strength
    ('freestream', None, (1, 0)),
    ('source', (26, 16), 3),
    ('source', (38, 16), -3),
)
SEEDS = (
#   (x, y) starting location
    (0, 0),
    (0, 3),
    (0, 6),
    (0, 9),
    (0, 12),
    (0, 15),
    (0, 17),
    (0, 20),
    (0, 23),
    (0, 26),
    (0, 29),
)
MATRIX_WIDTH = 64
MATRIX_HEIGHT = 32
BACK_COLOR = 0x000000 # background fill
SING_COLOR = 0xADAF00 # singularities
HEAD_COLOR = 0x00FFFF # leading particles
```

```
TAIL_COLOR = 0x000A0A # trailing particles
TAIL_LENGTH = 10      # length in pixels
DELAY = 0.01          # smaller = faster
#---------------------------------------------------------
```

The result should look like this:



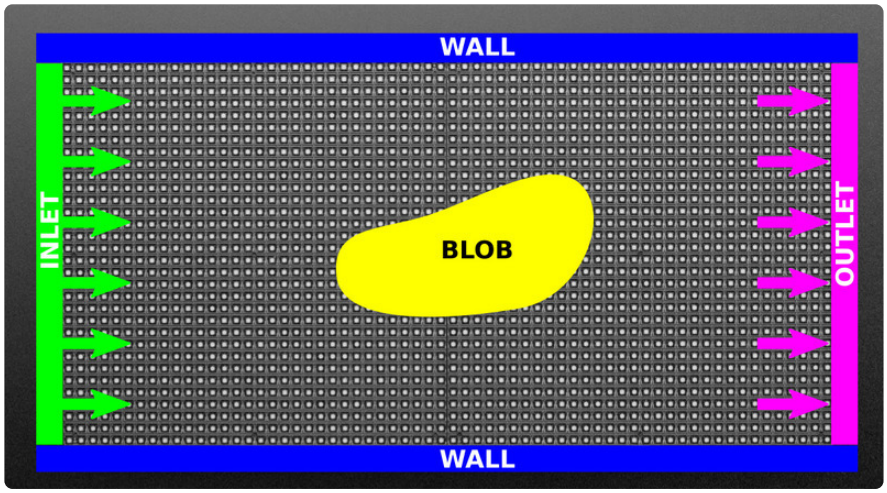
Lifting Flow Over a Cylinder

This is called a cylinder (https://adafru.it/OE7) instead of a sphere since the flow is two dimensional. We use a **doublet** to generate the main cylinder flow. To simulate lift we use a **vortex**. The two are located at the same location. And of course we still need our good 'ole friend the **freestream**.

```
#--| User Config |-------------------------------------
SINGULARITIES = (
    # type  location  strength
    ('freestream', None, (1, 0)),
    ('doublet', (32, 16), 3),
    ('vortex', (32, 16), 4),
)
SEEDS = (
#   (x, y) starting location
    (0, 0),
    (0, 3),
    (0, 6),
    (0, 9),
    (0, 12),
    (0, 15),
    (0, 17),
    (0, 20),
    (0, 23),
    (0, 26),
    (0, 29),
)
MATRIX_WIDTH = 64
MATRIX_HEIGHT = 32
BACK_COLOR = 0x000000 # background fill
SING_COLOR = 0xADAF00 # singularities
HEAD_COLOR = 0x00FFFF # leading particles
TAIL_COLOR = 0x000A0A # trailing particles
TAIL_LENGTH = 10      # length in pixels
DELAY = 0.01          # smaller = faster
#---------------------------------------------------------
```

What does this one look like? Well...run it and see :)

Flow Solver Viewer



The flow singularity approach is nice and easy but is sort of backwards. You specify the singularities and the resulting flow field is then computed. More often, you have some blob and want to know the resulting flow over it. To do this, a different approach is taken. You define the shape of your blob and other flow field parameters and then Laplace's equation is solved numerically. The solution is the resulting flow field, so this is called a "flow solver".

The trade off here is complexity. Not only for the underlying solution process (differential equation solver), but also for actually using the flow solver (grid generator, boundary conditions, etc.).

However, we found this really neat Python based solver:

2D Potential Flow Solver

https://adafru.it/OE8

This is super easy to use since the setup is simply a matrix. You fill the matrix with one of the possible flow elements, and then send it to the solver. The solver outputs the resulting flow field, also as a matrix.

This pairs nicely with the RGB matrix. The idea is to use the RGB matrix as a viewer for the flow solution. The solver won't actually run on the Matrix Portal* so we'll do something like this:

1. Install the flow solver on a PC.
2. Define our blob and run the solver on the PC.
3. Output flow solution to a file.

4. Copy solution file to Matrix Portal.
5. Run viewer script on Matrix Portal.

*it probably could though, given enough time to port the code to CircuitPython.

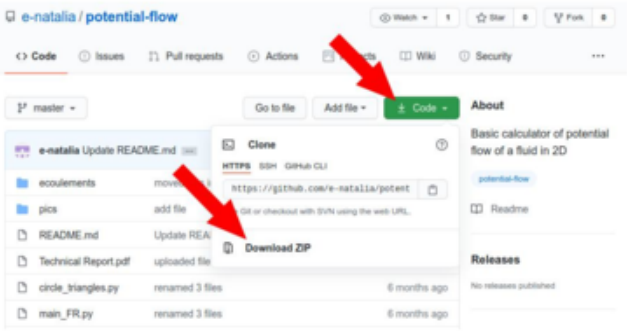
Install Required Software

Do this software setup on the PC you want to run the flow solver on.

You'll need the following software. Click the links to go to installation instructions.

- **Python** (https://adafru.it/deW) - you may already have this installed.
- **NumPy** (https://adafru.it/O1D) - used by the flow solver.
- **Pillow/PIL** (https://adafru.it/OE9) - used to read BMP defining input geometry.

Then we need to install the flow solver software (https://adafru.it/OE8). If you are familiar with git, you can just clone the solver repo linked above. Otherwise, you can download the solver code as follows:



Go to the repo on Github (https://adafru.it/OE8).
Click the green **Code** button.
Click **Download ZIP** and save the file.
Unzip the contents to a folder location on your PC.

We'll work in the folder location where you saved the solver. So remember it.

Define Input Geometry

This is done with a BMP file. It should be a black and white image the same size as the RGB matrix. **See the next section for details about creating this file.**

Solver Script

Run this on your PC.

OK, now grab the solver script which will read in your BMP file, run the solver, and then output the results. To make things easy to run, save a copy of this file in the **same folder as flow solver** that was download above.

```
# SPDX-FileCopyrightText: 2020 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

#======
# NOTE: Run this on your PC, not the Matrix Portal.
#======
import sys
import numpy as np
from PIL import Image
from ecoulements import systeme

# load geometry
grid = np.where(np.asarray(Image.open(sys.argv[1])), 1, 0)

# add inlet / outlet flows
inlet = np.array([2] * grid.shape[0])
outlet = np.array([3] * grid.shape[0])
grid = np.hstack((inlet[:, None], grid, outlet[:, None]))

# add upper/ lower walls
wall = np.array([0] * grid.shape[1])
grid = np.vstack((wall, grid, wall))

# solve
_, VX, VY, _ = systeme.sol(grid)

# save results to file
OUTFILE = "flow_solution.py"
with open(OUTFILE , "w") as fp:
    fp.write("nan = None\n")
    fp.write("solution = {\n")
    fp.write('"VX":\n')
    fp.write(str(VX[1:-1, 1:-1].tolist()))
    fp.write(',\n"VY":\n')
    fp.write(str(VY[1:-1, 1:-1].tolist()))
    fp.write("\n}\n")

# done
print("DONE! Results saved to", OUTFILE)
```

To run the solver, use the following command:

```
python flow_runner.py geometry.bmp
```

where `geometry.bmp` is the BMP file you created to define your input geometry. You can use a different name if you want, but the BMP file should be located in the same folder location.

If it runs without errors, it should say **DONE** and the results will be in a new file named `flow_solution.py`.

Move on to the next section for how to view those results!

Viewer Script

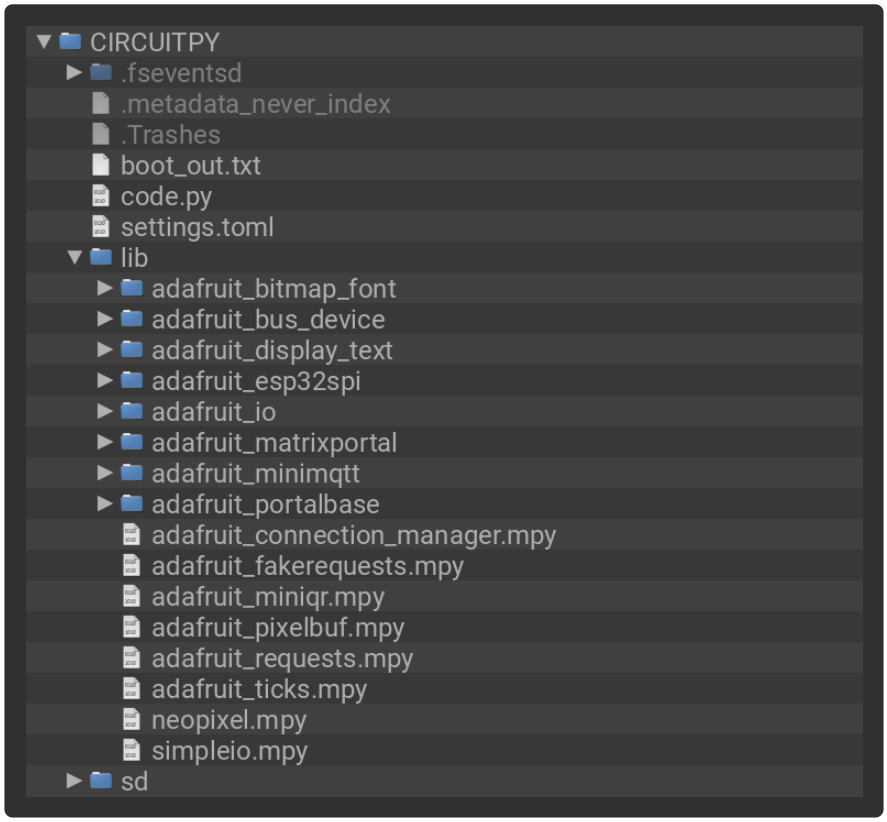
Run this on the Matrix Portal.

Installing Project Code

To use with CircuitPython, you need to first install a few libraries, into the lib folder on your **CIRCUITPY** drive. Then you need to update **code.py** with the example script.

Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **Matrix_Portal_Flow_Viewer/flow_viewer/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:



The CircuitPython libraries required are the same as for the singularity viewer sketch.

```
# SPDX-FileCopyrightText: 2020 Carter Nelson for Adafruit Industries
#
# SPDX-License-Identifier: MIT

#======
# NOTE: Run this on the Matrix Portal.
#======
```

```
import time
import math
import displayio
from adafruit_matrixportal.matrix import Matrix

from flow_solution import solution

#--| User Config |-------------------------------------
SEEDS = (
#   (x, y) starting location
    (0, 1),
    (0, 3),
    (0, 5),
    (0, 7),
    (0, 9),
    (0, 11),
    (0, 13),
    (0, 15),
    (0, 17),
    (0, 19),
    (0, 21),
    (0, 23),
    (0, 25),
    (0, 27),
    (0, 29),
)
BACK_COLOR = 0x000000 # background fill
SOLI_COLOR = 0xADAF00 # solids
HEAD_COLOR = 0x00FFFF # leading particles
TAIL_COLOR = 0x000A0A # trailing particles
TAIL_LENGTH = 10      # length in pixels
DELAY = 0.02          # smaller = faster
#-----------------------------------------------------------

# use solution to define other items
VX = solution['VX']
VY = solution['VY']
MATRIX_WIDTH = len(VX[0])
MATRIX_HEIGHT = len(VX)
# meh...too lazy to list comp
SOLIDS = []
for row in range(len(VX)):
    for col, v in enumerate(VX[row]):
        if v is None:
            SOLIDS.append((col, row))

# matrix and displayio setup
matrix = Matrix(width=MATRIX_WIDTH, height=MATRIX_HEIGHT, bit_depth=6)
display = matrix.display
group = displayio.Group()
display.root_group = group

bitmap = displayio.Bitmap(display.width, display.height, 4)

palette = displayio.Palette(4)
palette[0] = BACK_COLOR
palette[1] = SOLI_COLOR
palette[2] = HEAD_COLOR
palette[3] = TAIL_COLOR

tile_grid = displayio.TileGrid(bitmap, pixel_shader=palette)
group.append(tile_grid)

# global to store streamline data
STREAMLINES = []

def compute_streamlines():
    '''Compute streamline for each starting point (seed) defined.'''
    for seed in SEEDS:
```

```
        streamline = []
        x, y = seed
        px = round(x)
        py = round(y)
        vx = VX[py][px]
        vy = VY[py][px]
        streamline.append( ((px, py), (vx, vy)) )
        steps = 0
        while x < MATRIX_WIDTH and steps < 2 * MATRIX_WIDTH:
            nx = round(x)
            ny = round(y)
            # if we've moved to a new pixel, store the info
            if nx != px or ny != py:
                streamline.append( ((nx, ny), (vx, vy)) )
                px = nx
                py = ny
            if 0 <= nx < MATRIX_WIDTH and 0 <= ny < MATRIX_HEIGHT:
                vx = VX[ny][nx]
                vy = VY[ny][nx]
            if vx is None or vy is None:
                break
            x += vx
            y += vy
            steps += 1
        # add streamline to global store
        STREAMLINES.append(streamline)

def show_solids():
    for s in SOLIDS:
        try:
            x, y = s
            bitmap[round(x), round(y)] = 1
        except: # pylint: disable=bare-except
            pass # just don't draw it

def show_streamlines():
    '''Draw the streamlines.'''
    for sl, head in enumerate(HEADS):
        try:
            streamline = STREAMLINES[sl]
            index = round(head)
            length = min(index, TAIL_LENGTH)
            # draw tail
            for data in streamline[index-length:index]:
                x, y = data[0]
                bitmap[round(x), round(y)] = 3
            # draw head
            bitmap[round(x), round(y)] = 2
        except: # pylint: disable=bare-except
            pass # just don't draw it

def animate_streamlines():
    '''Update the current location (head position) along each streamline.'''
    reset_heads = True
    for sl, head in enumerate(HEADS):
        # get associated streamline
        streamline = STREAMLINES[sl]
        # compute index
        index = round(head)
        # get velocity
        if index < len(streamline):
            vx, vy = streamline[index][1]
            reset_heads = False
        else:
            vx, vy = streamline[-1][1]
        # move head
        HEADS[sl] += math.sqrt(vx*vx + vy*vy)
    if reset_heads:
        # all streamlines have reached the end, so reset to start
```

```
        for index, _ in enumerate(HEADS):
            HEADS[index] = 0

def update_display():
    '''Update the matrix display.'''
    display.auto_refresh = False
    bitmap.fill(0)
    show_solids()
    show_streamlines()
    display.auto_refresh = True

#==========
# MAIN
#==========
print('Computing streamlines...', end='')
compute_streamlines()
print('DONE')
HEADS = [0]*len(STREAMLINES)
print('Flowing...')
while True:
    animate_streamlines()
    update_display()
    time.sleep(DELAY)
```

After running the solver script on your PC, you should end up with a file named `flow_solution.py`. Copy that file to your **CIRCUITPY** folder. Then use the following script to view the results.

Customizing The Viewer

Similar to our singularity viewer, you can customize the behavior. Look for the `User Config` section near the top. The options are the same as for the singularity viewer, so see the information in that section.

Creating Bitmaps

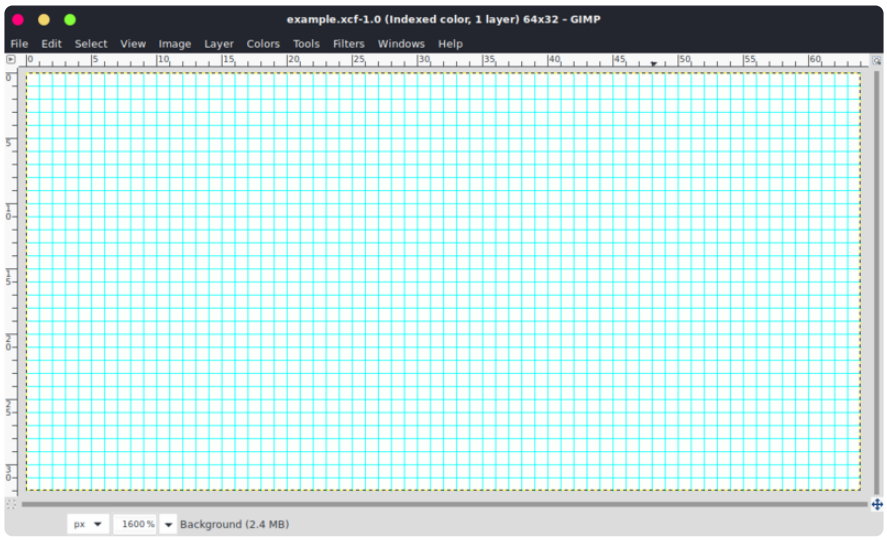
We use a bitmap (BMP) file to define the geometry to input into the flow solver discussed in the previous section. This lets you use image editing software like Photoshop or Gimp to easily draw whatever shapes you want. Here we give some brief info on how to do this.

The example here shows Gimp (https://adafru.it/GCa).

> When creating the bitmap, keep in mind that the flow will be from left to right.

Create Blank Bitmap

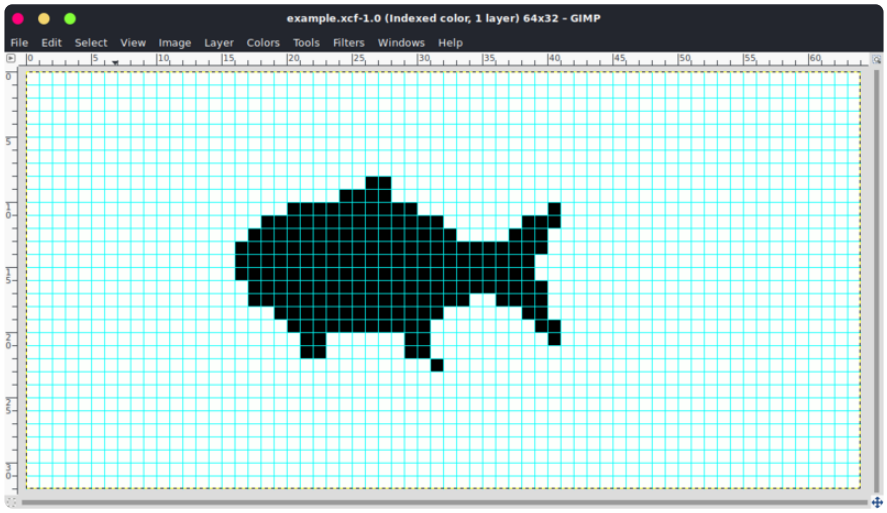
Create a new image (**File -> New...**) and set the pixel size to match the matrix size, for example 64 x 32. Set the Image mode to indexed (**Image -> Mode -> Indexed**) and set the maximum number of palette colors to 2. Then turn on grids (**View -> Show Grid**) and zoom in. You should have a nice blank canvas to start drawing your geometry.



Draw Geometry

Use the **Toolbox** palette to select the **Pencil** tool. In the Pencil options, set the **Brush** to **Pixel** and the **Size** to **1**.

You are now ready to draw your geometry one pixel at a time.



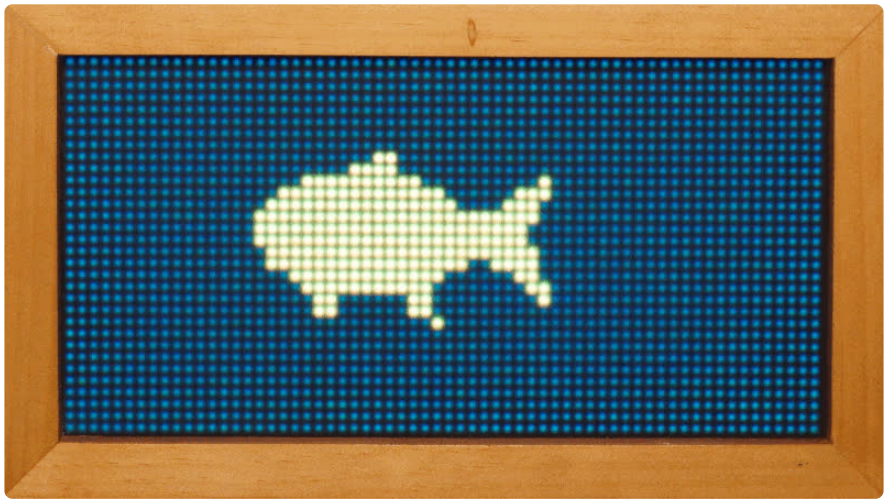
You can make more than one blob if you want. Here we just show a happy little fish.

The geometry should be solid with no internal open areas.

Export Geometry

When you are done and ready to save your geometry, export it as a BMP file. It's a good idea to also save it in the native Gimp (or Photoshop) format also, to make future edits easier. To export to a BMP file use **File -> Export As...** and save it.

You can then use the BMP as an input to flow solver runner script discussed in the previous section.



Examples

Here are a few ready to go BMP files you can use to get started.

fish.bmp

https://adafru.it/OEa

hearts.bmp

https://adafru.it/OEb

multi_element.bmp

https://adafru.it/OEc

But get creative and have fun with this. Enjoy :)