ENGR 520 Homework 2

Anthony Su

April 29, 2025

Collaboration statement: I did not collaborate with others on this assignment. I utilized large language models for code debugging.

1 Dynamic Mode Decomposition

1.a Full Clean Data

The dynamic mode decomposition (DMD) was performed on the vorticity of flow past a cylinder. A snapshot visualizing the flow field is shown in Figure 1. The first 21 eigenvalues of this DMD solution are shown in Figure 2.

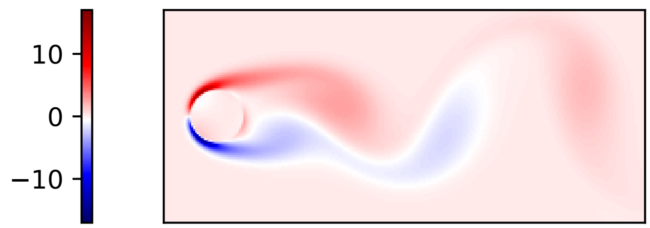


Figure 1: Snapshot of clean data

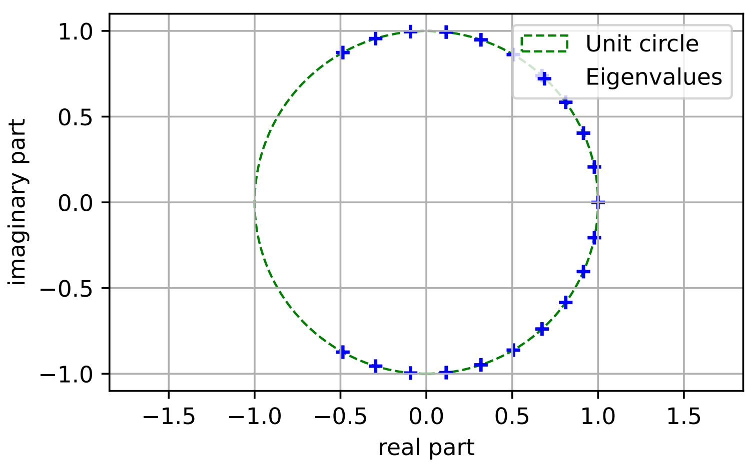


Figure 2: Eigenvalues of clean data

For a discrete-time system, eigenvalues on the unit circle represent purely oscillatory modes (with no damping/dissipation). This is physically equivalent to energy conservation.

1.b Full Noisy Data

Noise of three different magnitudes was added to the vorticity data and the DMD was performed again for each magnitude of noise. Snapshots visualizing the impact of noise are shown in Figures 3, 4, and 5. The first 21 eigenvalues of these DMD solutions are shown in Figures 6, 7, and 8.

The DMD eigenvalues of the noisy data are misleading because they do not lie on the unit circle, but rather inside of it. This implies energy dissipation, which is not actually occurring in the underlying data; this is an artifact of the added noise. The more significant first few modes are less sensitive to noise than the less significant later modes.

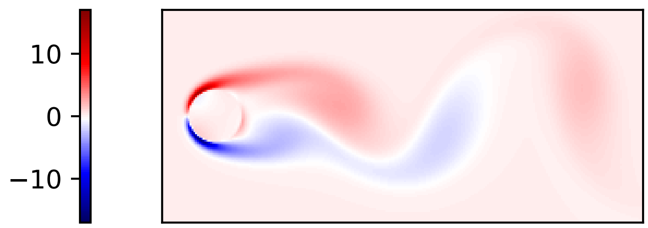


Figure 3: Snapshot of 1% noisy data

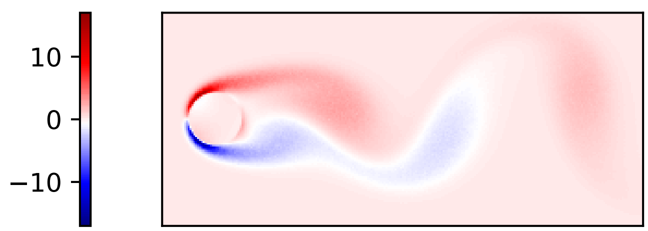


Figure 4: Snapshot of 10% noisy data

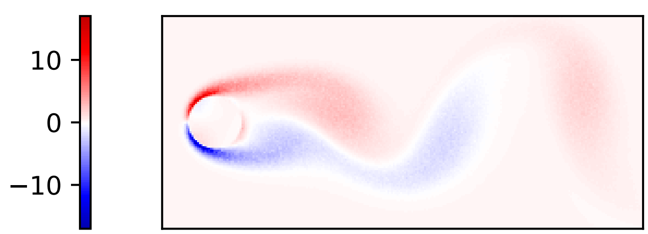


Figure 5: Snapshot of 20% noisy data

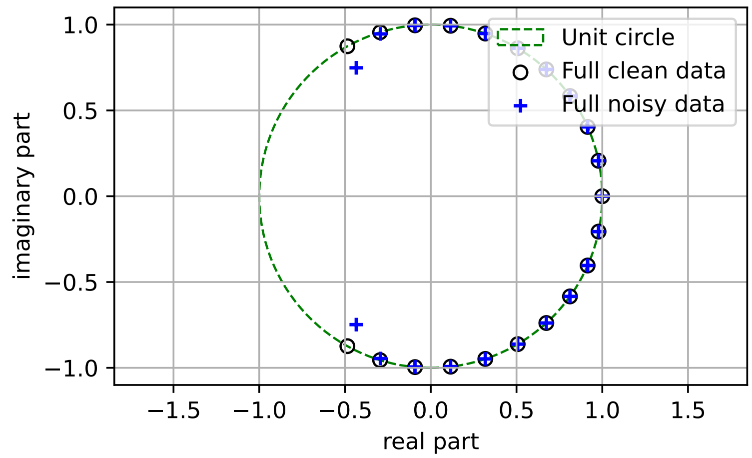


Figure 6: Eigenvalues of 1% noisy data

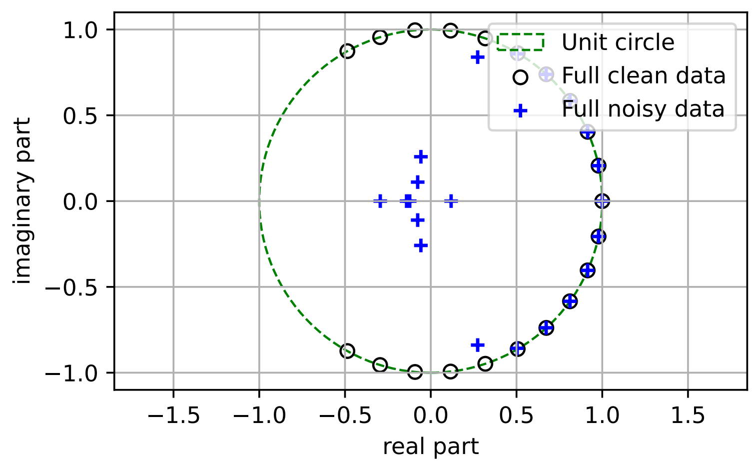


Figure 7: Eigenvalues of 10% noisy data

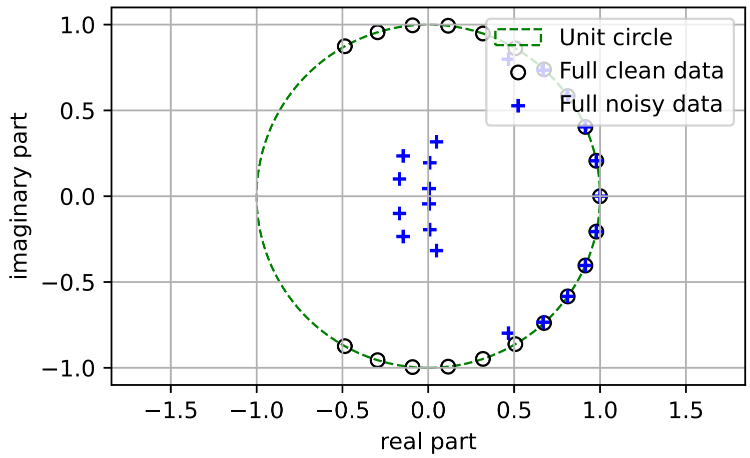


Figure 8: Eigenvalues of 20% noisy data

1.c Subset of Clean Data

The vorticity data was truncated so that its length was 75% of the vortex shedding period and DMD was performed on this truncated data.

The DMD eigenvalues of the truncated data also lie within the unit circle, falsely implying dissipation. Unlike for the noisy data, the sensitivity of the various modes to this source of error is more evenly distributed across the modes (but the higher modes are still more sensitive).

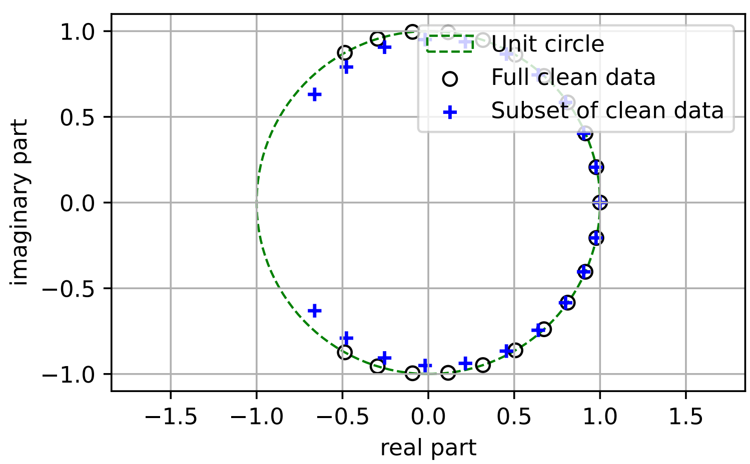


Figure 9: Eigenvalues of clean data subset

Physics-Informed Dynamic Mode Decomposition

In this flow field, physics that would be useful to include would be:

- periodicity (energy conservation)
- shift-invariance

- mass continuity

2 Physics-Informed Neural Networks

2.a Loss Function

The physics-informed loss function L is defined as:

$$L(\theta) = \text{MSE}(\hat{u}, u) + \text{MSE}(\hat{v}, v) + \text{MSE}\left(\frac{\partial \hat{u}}{\partial x} + \frac{\partial \hat{v}}{\partial y}, 0\right)$$

where

- $\text{MSE}(a, b) = \frac{1}{n}\sum_{i=1}^{n}(b-a)^2$
- λ is the weight of the physics loss term
- n is the number of training data points
- m is the number of virtual points (for computing physics error)
- $\boldsymbol{u}$ is the true velocity vector
- $\hat{\boldsymbol{u}}$ is the estimated velocity vector computed by the model $f_\theta(\boldsymbol{x})$

The first two terms of L are the mean-squared error (MSE) of the model f_θ across all n training data points. The third term of L is the mean-squared *physics* error (flow divergence) of f_θ across all m virtual points.

2.b Full Data

Interpolation outperforms my PINN with the full data.

Table 1: Full data model performance

flow	method	data loss
linear potential vortex	neural network	6.478e-05
linear potential vortex	linear interpolation	4.545e-33
Taylor-Green potential vortex	neural network	1.164e+00
Taylor-Green potential vortex	linear interpolation	1.123e+00

2.c Half Data

My PINN should outperform linear interpolation when trained on half of the data and tested on the other half. I think there's an issue with my linear potential vortex interpolation implementation.

Table 2: Half data model extrapolation performance

flow	method	data loss
linear potential vortex	PINN	1.760e-01
linear potential vortex	linear interpolation	1.573e-31
Taylor-Green potential vortex	PINN	1.101e+00
Taylor-Green potential vortex	linear interpolation	2.180e+00

2.d Symmetry

The symmetry can be enforced in the linear potential vortex with the following loss term:

$$\mathrm{MSE}\left(\hat{u}, -\hat{u}_r\right) + \mathrm{MSE}\left(\hat{v}, -\hat{v}_r\right)$$

and in the Taylor-Green potential vortex with the following loss term:

$$\mathrm{MSE}\left(\hat{u}, 1 - \hat{u}_m\right) + \mathrm{MSE}\left(\hat{v}, \hat{v}_m\right)$$

where

- $(\hat{u}, \hat{v}) = f_\theta(x, y)$
- $(\hat{u}_r, \hat{v}_r) = f_\theta(-x, -y)$
- $(\hat{u}_m, \hat{v}_m) = f_\theta(-x, y)$

My PINN with symmetry loss should outperform linear interpolation when trained on half of the data and tested on the other half. I think there's an issue with my linear potential vortex interpolation implementation.

Enforcing symmtetry leads to improvement over the PINN without symmetry for the linear potential vortex.

Table 3: Half data model extrapolation performance

flow	method	data loss
linear potential vortex	PINN with symmetry	2.268e-04
linear potential vortex	linear interpolation	1.573e-31
Taylor-Green potential vortex	PINN with symmetry	1.068e+00
Taylor-Green potential vortex	linear interpolation	2.180e+00

Code Appendix

Note that code for Section 2.a was modified and re-run for Section 2.b and similarly for Section 2.c, so only code for Section 2.c is shown.

ENGR 520 Homework 2 Exercise 2-1

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors
import scipy.io
from pydmd import DMD, plotter
```

In [2]:
```
# Load data
def load_mat_as_variables(file_path):
    mat_data = scipy.io.loadmat(file_path)
    for key, value in mat_data.items():
        if not key.startswith("__"):  # skip metadata keys
            globals()[key] = value

file_path = "CYLINDER_ALL.mat"
load_mat_as_variables(file_path)

# Reshape
m = m[0][0]
n = n[0][0]
nx = nx[0][0]
ny = ny[0][0]
```

Variable	Type	Description
`UALL`	89351 x 151 array	$\boldsymbol{u}(t)$
`UEXTRA`	89351 x 1 array	$\boldsymbol{u}_0$
`VALL`	89351 x 151 array	$\boldsymbol{v}(t)$
`VEXTRA`	89351 x 1 array	$\boldsymbol{v}_0$
`VORTALL`	89351 x 151 array	$\boldsymbol{\omega}(t)$
`VORTEXTRA`	89351 x 1 array	$\boldsymbol{\omega}_0$
`m`	int	domain width
`n`	int	domain height
`nx`	int	domain width
`ny`	int	domain height

In [3]:
```
# Plotting functions
def plot_frame(frame, title="", clim=None):
    if clim:
        values = np.linspace(*clim, 1000)
        norm = matplotlib.colors.Normalize(*clim)
    else:
        values = None
        norm = None
    fig, ax = plt.subplots(figsize=(5, 1.5))
    im = ax.imshow(np.rot90(np.reshape(frame, (ny, nx))), origin="lower", cmap="seismic")
    fig.colorbar(im, values=values, location="left")
    ax.set_title(title)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    return fig, ax


def plot_eigs(eigs, title=""):
    fig, ax = plt.subplots(figsize=(5, 3)
                           )
    ax.add_patch(plt.Circle((0, 0), 1, color="green", fill=False, label="Unit circle", linestyle="--"))
    ax.scatter(np.real(eigs), np.imag(eigs), c="b", marker="+", label="Eigenvalues")
    ax.legend(loc="upper right")
    ax.set_xlabel("real part")
    ax.set_ylabel("imaginary part")
    limit = np.max(np.ceil(np.absolute(eigs)))
```

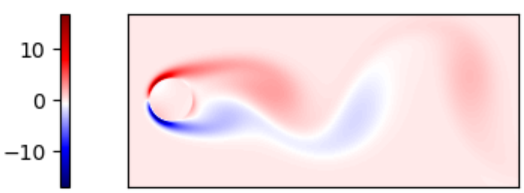
```
    ax.set_xlim((-limit, limit))
    ax.set_ylim((-limit, limit))
    ax.axis('equal')
    ax.set_title(title)
    ax.grid(True)
    return fig, ax


def plot_modes(modes, title=""):
    mask = [-1] + list(range(len(modes.T)-2, 0, -2))
    figs = []
    axs = []
    for idx, mode in enumerate(modes.T[mask]):
        fig, ax = plot_frame(np.real(mode), title=title+f" Mode {idx}")
        figs.append(fig)
        axs.append(ax)
    return figs, axs


def plot_compare_eigs(eigs_1, eigs_2, label_1="" , label_2="", title=""):
    fig, ax = plt.subplots(figsize=(5, 3))
    ax.add_patch(plt.Circle((0, 0), 1, color="green", fill=False, label="Unit circle", linestyle="--"))
    ax.scatter(np.real(eigs_1), np.imag(eigs_1), marker="o", edgecolor="k", facecolor="none", label=label_1
    ax.scatter(np.real(eigs_2), np.imag(eigs_2), c="b", marker="+", label=label_2)
    ax.legend(loc="upper right")
    ax.set_xlabel("real part")
    ax.set_ylabel("imaginary part")
    limit = np.max(np.ceil(np.absolute(np.concatenate((eigs_1, eigs_2)))))
    ax.set_xlim((-limit, limit))
    ax.set_ylim((-limit, limit))
    ax.axis('equal')
    ax.set_title(title)
    ax.grid(True)
    return fig, ax
```

a. Clean Data

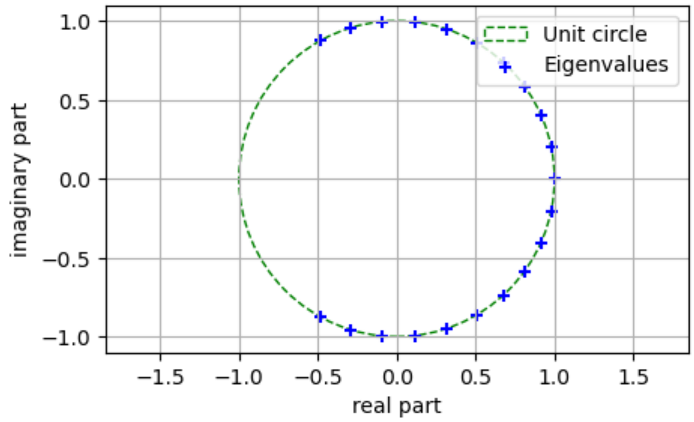
In [4]:
```
dmd = DMD(svd_rank=21, sorted_eigs="real")
dmd.fit(VORTALL)
eigs_clean = dmd.eigs
```

```
/home/exurl/anaconda3/lib/python3.11/site-packages/pydmd/snapshots.py:73: UserWarning: Input data condition
number 9585725.906000633. Consider preprocessing data, passing in augmented data
matrix, or regularization methods.
  warnings.warn(
```

In [5]:
```
# Plot snapshot
fig_snapshot, _ = plot_frame(VORTALL[:, 0], clim=(-17, 17))
fig_snapshot.savefig(f"fig1a_snapshot.pdf", bbox_inches="tight")
```



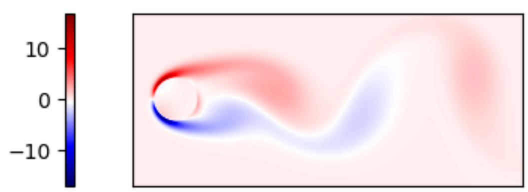
In [6]:
```
# Plot eigenvalues
fig, _ = plot_eigs(dmd.eigs)
fig.savefig("fig1a_eigs.pdf", bbox_inches="tight")
```

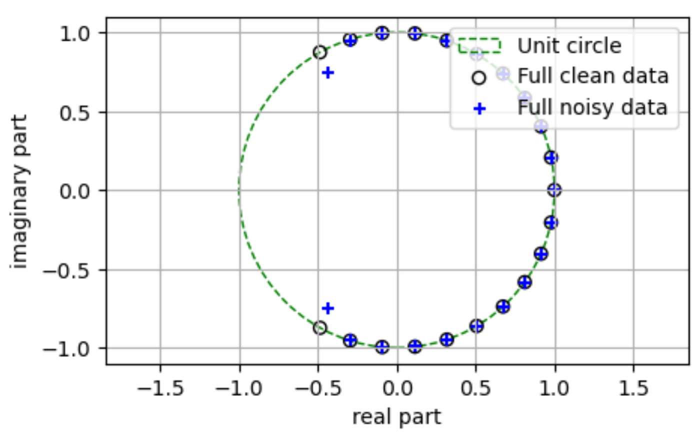


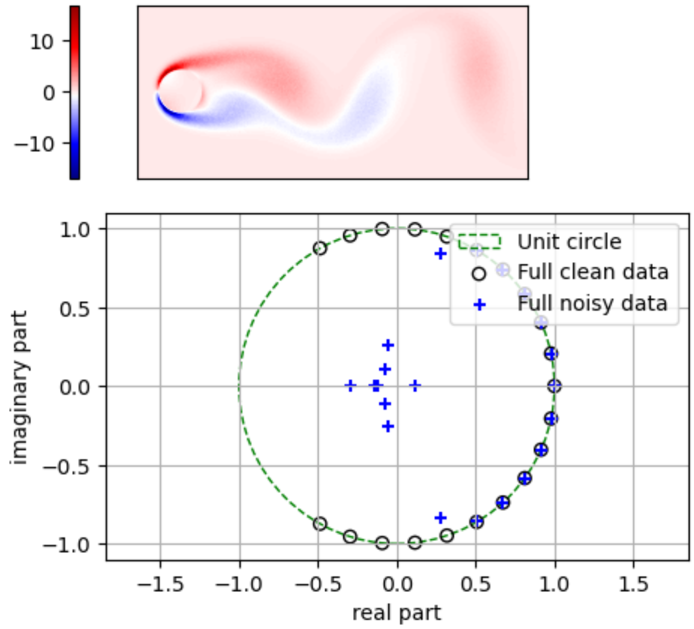
b. Noisy Data

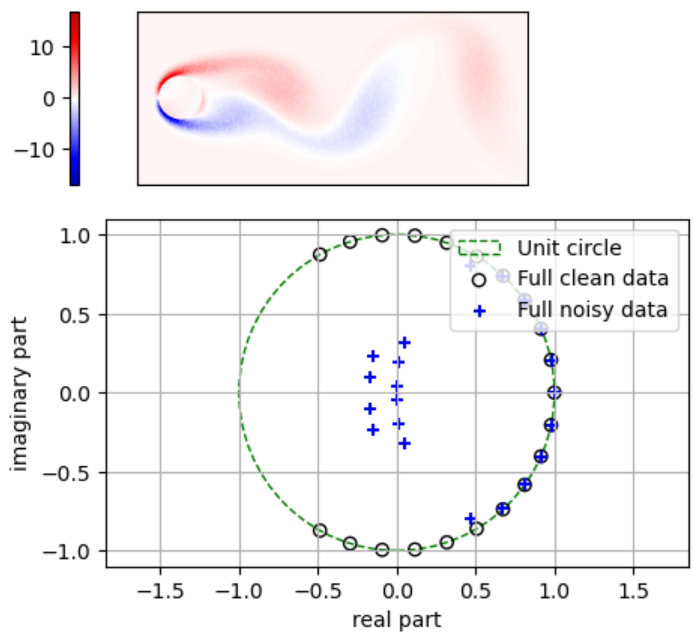
In [7]:
```python
# Initialize RNG
rng = np.random.default_rng()
```

In [8]:
```python
noise_magnitudes = [0.01, 0.1, 0.2]
for idx, noise_mag in enumerate(noise_magnitudes):
    # Create noisy data
    scale = noise_mag * np.linalg.norm(VORTALL) / np.sqrt(VORTALL.size)
    VORTALL_noisy = VORTALL + VORTALL * rng.normal(loc=0, scale=scale, size=VORTALL.shape)

    # Compute DMD
    dmd = DMD(svd_rank=21, sorted_eigs="real")
    dmd.fit(VORTALL_noisy)

    # Plot snapshot
    fig_snapshot, _ = plot_frame(VORTALL_noisy[:, 0], clim=(-17, 17))
    fig_snapshot.savefig(f"fig1b_snapshot_{idx+1}.pdf", bbox_inches="tight")

    # Plot eigenvalues
    fig_eigs, _ = plot_compare_eigs(eigs_clean, dmd.eigs, "Full clean data", "Full noisy data")
    fig_eigs.savefig(f"fig1b_eigs_{idx+1}.pdf", bbox_inches="tight")
```





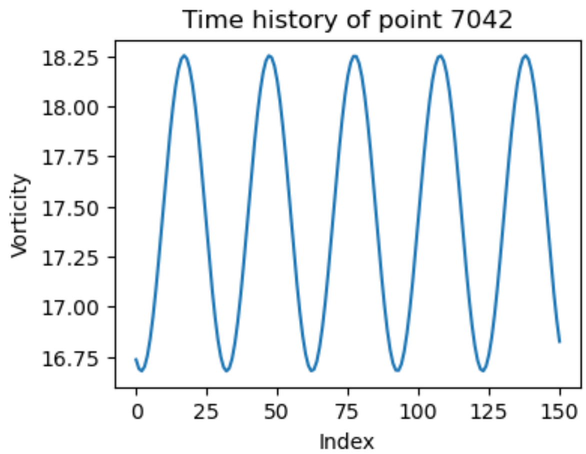




c. Clean Data Subset

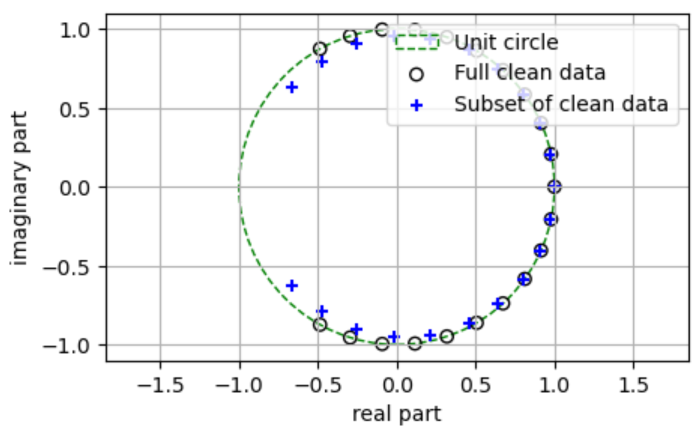
In [9]:

```
# Plot oscillation over time
idx_pt = np.argmax(VORTALL[:, 0])
fig, ax = plt.subplots(figsize=(4, 3))
plt.plot(VORTALL[idx_pt, :])
plt.xlabel("Index")
plt.ylabel("Vorticity")
plt.title(f"Time history of point {idx_pt}")
plt.show()
```



In [10]:
```
# Determine vortex shedding period
vort_fft = np.abs(np.fft.fft(VORTALL[idx_pt, :]))
idx = np.argmax(vort_fft)
multiple = vort_fft[idx]
period = len(VORTALL) / multiple
```

In [11]:
```
# Create subset data
idx_end = round(period * 0.75)
VORTALL_subset = VORTALL[:, :idx_end]

# Compute DMD
dmd = DMD(svd_rank=21, sorted_eigs="real")
dmd.fit(VORTALL_subset)

# Plot eigenvalues
fig, _ = plot_compare_eigs(eigs_clean, dmd.eigs, "Full clean data", "Subset of clean data")
fig.savefig(f"fig1c_eigs.pdf", bbox_inches="tight")
```



ENGR 520 Homework 2 Exercise 2-2

In [1]:
```
import numpy as np
from numpy.typing import ArrayLike
from matplotlib.figure import Figure
from matplotlib.axes import Axes
import matplotlib.pyplot as plt
import torch
from tqdm import trange
from scipy.interpolate import LinearNDInterpolator, RegularGridInterpolator
```

In [2]:
```
# Use CUDA if available
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Device: {DEVICE}")
```

```
Device: cpu
```

Generate Data

In [3]:
```
def linear_flow_func(x, y):
    """Linear potential vortex"""
    u = -y
    v = x
    return u, v


def tg_flow_func(x, y):
    """Taylor-Green potential vortex"""
    u = np.sin(2 * np.pi * x) * np.cos(2 * np.pi * y)
    v = -np.cos(2 * np.pi * x) * np.sin(2 * np.pi * y)
    return u, v
```

In [4]:
```
# Generate linear training data
xlim = (-0.95, -0.05)
ylim = (-0.9, 0.9)
nx = 10
ny = 10
N_train_linear = nx * ny
x_vec_train_linear = np.linspace(*xlim, nx)
y_vec_train_linear = np.linspace(*ylim, ny)
x_grid_train_linear, y_grid_train_linear = np.meshgrid(
    x_vec_train_linear,
    y_vec_train_linear,
    )
u_grid_train_linear, v_grid_train_linear = linear_flow_func(x_grid_train_linear, y_grid_train_linear)
print(f"linear training data xy shape: {x_grid_train_linear.shape}")
print(f"linear training data uv shape: {u_grid_train_linear.shape}")

# Generate linear test data
xlim = (0.005, 0.995)
ylim = (-0.99, 0.99)
nx = 100
ny = 100
N_test_linear = nx * ny
x_vec_test_linear = np.linspace(*xlim, nx)
y_vec_test_linear = np.linspace(*ylim, ny)
x_grid_test_linear, y_grid_test_linear = np.meshgrid(
    x_vec_test_linear,
    y_vec_test_linear,
    )
u_grid_test_linear, v_grid_test_linear = linear_flow_func(x_grid_test_linear, y_grid_test_linear)
print(f"linear testing data xy shape: {x_grid_test_linear.shape}")
print(f"linear testing data uv shape: {u_grid_test_linear.shape}")

# Generate Taylor-Green training data
xlim = (0.025, .475)
ylim = (0.05, .95)
nx = 10
ny = 10
N_train_tg = nx * ny
```

```python
x_vec_train_tg = np.linspace(*xlim, nx)
y_vec_train_tg = np.linspace(*ylim, ny)
x_grid_train_tg, y_grid_train_tg = np.meshgrid(
    x_vec_train_tg,
    y_vec_train_tg,
    )
u_grid_train_tg, v_grid_train_tg = tg_flow_func(x_grid_train_tg, y_grid_train_tg)
print(f"Taylor-Green training data xy shape: {x_grid_train_tg.shape}")
print(f"Taylor-Green training data uv shape: {u_grid_train_tg.shape}")

# Generate Taylor-Green test data
xlim = (0.0025, 0.9975)
ylim = (0.005, 0.995)
nx = 100
ny = 100
N_test_linear = nx * ny
x_vec_test_tg = np.linspace(*xlim, nx)
y_vec_test_tg = np.linspace(*ylim, ny)
x_grid_test_tg, y_grid_test_tg = np.meshgrid(
    x_vec_test_tg,
    y_vec_test_tg,
    )
u_grid_test_tg, v_grid_test_tg = linear_flow_func(x_grid_test_tg, y_grid_test_tg)
print(f"Taylor-Green testing data xy shape: {x_grid_test_tg.shape}")
print(f"Taylor-Green testing data uv shape: {u_grid_test_tg.shape}")
```

```
linear training data xy shape: (10, 10)
linear training data uv shape: (10, 10)
linear testing data xy shape: (100, 100)
linear testing data uv shape: (100, 100)
Taylor-Green training data xy shape: (10, 10)
Taylor-Green training data uv shape: (10, 10)
Taylor-Green testing data xy shape: (100, 100)
Taylor-Green testing data uv shape: (100, 100)
```

In [5]:

```python
# Convert linear training data to TensorDataset
x_train_tensor_linear = torch.from_numpy(x_grid_train_linear)
y_train_tensor_linear = torch.from_numpy(y_grid_train_linear)
u_train_tensor_linear = torch.from_numpy(u_grid_train_linear)
v_train_tensor_linear = torch.from_numpy(v_grid_train_linear)
linear_train_dataset = torch.utils.data.TensorDataset(
    x_train_tensor_linear,
    y_train_tensor_linear,
    u_train_tensor_linear,
    v_train_tensor_linear,
    )

# Convert linear test data to TensorDataset
x_test_tensor_linear = torch.from_numpy(x_grid_test_linear)
y_test_tensor_linear = torch.from_numpy(y_grid_test_linear)
u_test_tensor_linear = torch.from_numpy(u_grid_test_linear)
v_test_tensor_linear = torch.from_numpy(v_grid_test_linear)
linear_test_dataset = torch.utils.data.TensorDataset(
    x_test_tensor_linear,
    y_test_tensor_linear,
    u_test_tensor_linear,
    v_test_tensor_linear,
    )

# Convert Taylor-Green training data to TensorDataset
x_train_tensor_tg = torch.from_numpy(x_grid_train_tg)
y_train_tensor_tg = torch.from_numpy(y_grid_train_tg)
u_train_tensor_tg = torch.from_numpy(u_grid_train_tg)
v_train_tensor_tg = torch.from_numpy(v_grid_train_tg)
tg_train_dataset = torch.utils.data.TensorDataset(
    x_train_tensor_tg,
    y_train_tensor_tg,
    u_train_tensor_tg,
    v_train_tensor_tg,
    )

# Convert Taylor-Green test data to TensorDataset
x_test_tensor_tg = torch.from_numpy(x_grid_test_tg)
y_test_tensor_tg = torch.from_numpy(y_grid_test_tg)
u_test_tensor_tg = torch.from_numpy(u_grid_test_tg)
```

```
v_test_tensor_tg = torch.from_numpy(v_grid_test_tg)
tg_test_dataset = torch.utils.data.TensorDataset(
    x_test_tensor_tg,
    y_test_tensor_tg,
    u_test_tensor_tg,
    v_test_tensor_tg,
    )
```

In [6]:
```
# Linear training data loader
batch_size = 100  # <-- HYPERPARAMETER
linear_train_loader = torch.utils.data.DataLoader(
    linear_train_dataset,
    batch_size=batch_size,
    shuffle=True,  # <-- HYPERPARAMETER
    )

# Linear test data loader
batch_size = 10000  # <-- HYPERPARAMETER
linear_test_loader = torch.utils.data.DataLoader(
    linear_test_dataset,
    batch_size=batch_size,
    shuffle=True,  # <-- HYPERPARAMETER
    )

# Taylor-Green training data loader
batch_size = 100  # <-- HYPERPARAMETER
tg_train_loader = torch.utils.data.DataLoader(
    tg_train_dataset,
    batch_size=batch_size,
    shuffle=True,  # <-- HYPERPARAMETER
    )

# Taylor-Green test data loader
batch_size = 10000  # <-- HYPERPARAMETER
tg_test_loader = torch.utils.data.DataLoader(
    tg_test_dataset,
    batch_size=batch_size,
    shuffle=True,  # <-- HYPERPARAMETER
    )
```

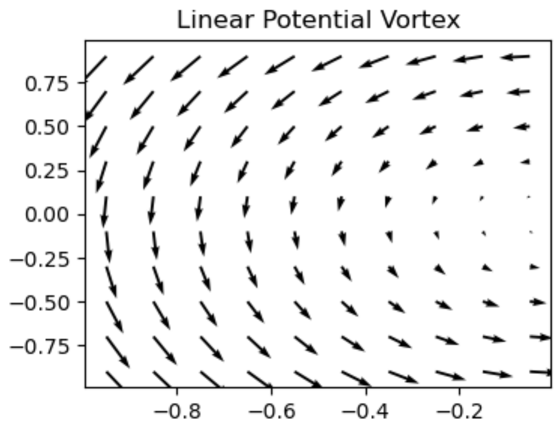
In [7]:
```
# Validate data attributes
sample_data = next(iter(linear_train_loader))[0]
print(f"Shape: {sample_data.shape}")
print(f"Type: {sample_data.dtype}")
```

```
Shape: torch.Size([10, 10])
Type: torch.float64
```

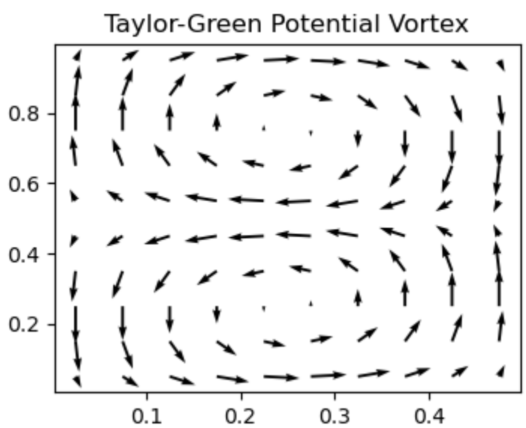
Visualize Data

In [8]:
```
def plot_flow_field(x: ArrayLike, y: ArrayLike, u: ArrayLike, v: ArrayLike, title: str=None) -> tuple[Figu
    """Plot vector-valued function

    Args:
        x (ArrayLike): (N) array
        y (ArrayLike): (N) array
        u (ArrayLike): (N) array
        v (ArrayLike): (N) array

    Returns:
        fig (Figure): Matplotlib figure
        ax (Axes): Matplotlib axes
    """
    fig, ax = plt.subplots(figsize=(4, 3))
    ax.quiver(x, y, u, v, cmap="viridis")
    ax.set_title(title)
    return fig, ax
```

In [9]:
```
# Visualize linear potential vortex training data
_ = plot_flow_field(x_grid_train_linear, y_grid_train_linear, u_grid_train_linear, v_grid_train_linear, "Li
```



In [10]:
```
# Visualize linear Taylor-Green potential vortex training data
_ = plot_flow_field(x_grid_train_tg, y_grid_train_tg, u_grid_train_tg, v_grid_train_tg, "Taylor-Green Poter
```



Define Architecture

In [11]:
```
def my_nn_model() -> torch.nn.Module:
    """3 hidden layers of 32 nodes each"""
    model = torch.nn.Sequential(
        torch.nn.Linear(2, 32),
        torch.nn.ReLU(),  # <-- HYPERPARAMETER
        torch.nn.Linear(32, 32),
        torch.nn.ReLU(),  # <-- HYPERPARAMETER
        torch.nn.Linear(32, 32),
        torch.nn.ReLU(),  # <-- HYPERPARAMETER
        torch.nn.Linear(32, 2),
    )
    return model
```

Define Loss Functions

Data error:

$$E(x_i, y_i) = (\hat{u}_i - u) + (\hat{v}_i - v_i)$$

Physics error:

$$E_{\text{phys}}(x_i, y_i) = \frac{\partial \hat{u}_i}{\partial x} + \frac{\partial \hat{v}_i}{\partial y}$$

Symmetry error:

$$E_{\text{sym}}(x_i, y_i) = \left(\hat{u}_i + \hat{u}_j\right) + \left(\hat{v}_i - \hat{v}_j\right)$$

where $(x_k, y_k) = (-x_j, y_j)$

In [12]:
```python
physics_weight = 0.5  # <-- HYPERPARAMETER
symmetry_weight = 1  # <-- HYPERPARAMETER
loss_func = torch.nn.MSELoss()  # <-- HYPERPARAMETER
```

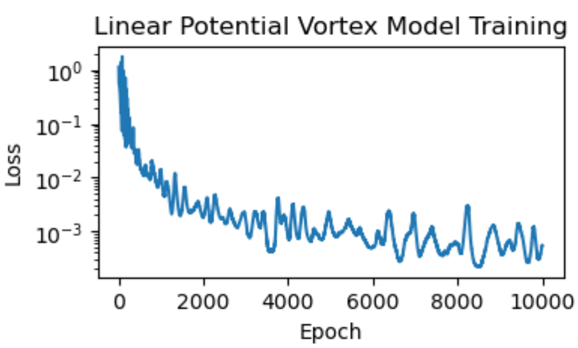
Train

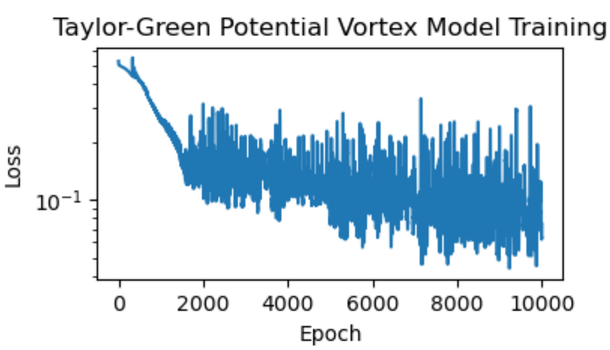
In [13]:
```python
# Initialize training
linear_model = my_nn_model().to(DEVICE)
tg_model = my_nn_model().to(DEVICE)
num_epochs = 10000  # <-- HYPERPARAMETER

linear_train_loss_hist = []
tg_train_loss_hist = []
for model, train_loss_hist, data_loader, symmetry in zip(
    [linear_model, tg_model],
    [linear_train_loss_hist, tg_train_loss_hist],
    [linear_train_loader, tg_train_loader],
    ["odd", "even",]
):
    optimizer = torch.optim.SGD(model.parameters(), lr=0.05)  # <-- HPYERPARAMETER

    # Train
    for idx_epoch in trange(num_epochs):
        # Set model state to "training"
        model.train()
        train_loss = 0.0

        # Iterate over batches
        for x_grid, y_grid, u_grid, v_grid in data_loader:
            # Load data
            x_grid = x_grid.to(DEVICE).float().requires_grad_(True)
            y_grid = y_grid.to(DEVICE).float().requires_grad_(True)
            u_grid = u_grid.to(DEVICE).float()
            v_grid = v_grid.to(DEVICE).float()

            # Clear gradients
            optimizer.zero_grad()

            # Predict
            uv_grid = torch.cat((x_grid.unsqueeze(-1), y_grid.unsqueeze(-1)), dim=-1)
            uv_prediction = model(uv_grid)
            u_grid_prediction = uv_prediction[:, :, 0]
            v_grid_prediction = uv_prediction[:, :, 1]

            # Predict mirror
            if symmetry == "even":
                xy_grid_mirror = torch.cat((1 - x_grid.unsqueeze(-1), y_grid.unsqueeze(-1)), dim=-1)
            elif symmetry == "odd":
                xy_grid_mirror = torch.cat((-x_grid.unsqueeze(-1), -y_grid.unsqueeze(-1)), dim=-1)
            uv_prediction_mirror = model(xy_grid_mirror)
            u_grid_prediction_mirror = uv_prediction_mirror[:, :, 0]
            v_grid_prediction_mirror = uv_prediction_mirror[:, :, 1]

            # Compute data loss
            u_loss = loss_func(u_grid_prediction, u_grid)
            v_loss = loss_func(v_grid_prediction, v_grid)
            u_loss.backward(retain_graph=True)
            v_loss.backward(retain_graph=True)

            # Compute symmetry loss
            u_symmetry_loss = loss_func(u_grid_prediction, -u_grid_prediction_mirror)
            if symmetry == "even":
                v_symmetry_loss = loss_func(v_grid_prediction, v_grid_prediction_mirror)
            elif symmetry == "odd":
                v_symmetry_loss = loss_func(v_grid_prediction, -v_grid_prediction_mirror)
            u_symmetry_loss.backward(retain_graph=True)
            v_symmetry_loss.backward(retain_graph=True)

            # Compute physics loss
            dudx = torch.autograd.grad(
                u_grid_prediction,
                x_grid,
                grad_outputs=torch.ones_like(u_grid_prediction),
```

```
                create_graph=True
            )[0]
            dvdy = torch.autograd.grad(
                v_grid_prediction,
                y_grid,
                grad_outputs=torch.ones_like(v_grid_prediction),
                create_graph=True
            )[0]
            divergence = dudx + dvdy
            physics_loss = loss_func(divergence, torch.zeros_like(divergence))
            physics_loss.backward()

            # Compute total loss
            train_loss += u_loss.item() + v_loss.item() + physics_weight * physics_loss.item() + symmetry_w
            optimizer.step()

        # Record iteration
        train_loss_hist.append(train_loss / len(data_loader))
```

```
100%|██████████| 10000/10000 [00:28<00:00, 356.87it/s]
100%|██████████| 10000/10000 [00:24<00:00, 406.47it/s]
```

In [14]:
```
# Plot training trajectory
for train_loss_hist, title in zip(
    [
        linear_train_loss_hist,
        tg_train_loss_hist,
    ],
    [
        "Linear Potential Vortex Model Training",
        "Taylor-Green Potential Vortex Model Training",
    ],
):
    fig, ax = plt.subplots(figsize=(4, 2))
    ax.semilogy(train_loss_hist)
    ax.set_xlabel("Epoch")
    ax.set_ylabel("Loss")
    ax.set_title(title)
```





Test

In [15]:
```
# Initialize testing
linear_test_loss = [0.0]  # convert from float to list to force pass-by-reference
linear_data_loss = [0.0]
tg_test_loss = [0.0]
```

```python
tg_data_loss = [0.0]
for model, test_loss, data_loss, data_loader in zip(
    [linear_model, tg_model],
    [linear_test_loss, tg_test_loss],
    [linear_data_loss, tg_data_loss],
    [linear_test_loader, tg_test_loader],
):
    # Set model state to "evaluation"
    model.eval()

    # Test
    for x_grid, y_grid, u_grid, v_grid in data_loader:
        # Load data
        x_grid = x_grid.to(DEVICE).float().requires_grad_(True)
        y_grid = y_grid.to(DEVICE).float().requires_grad_(True)
        u_grid = u_grid.to(DEVICE).float()
        v_grid = v_grid.to(DEVICE).float()

        # Clear gradients
        optimizer.zero_grad()

        # Predict
        xy_grid = torch.cat((x_grid.unsqueeze(-1), y_grid.unsqueeze(-1)), dim=-1)
        uv_prediction = model(xy_grid)
        u_grid_prediction = uv_prediction[:, :, 0]
        v_grid_prediction = uv_prediction[:, :, 1]

        # Compute data loss
        u_loss = loss_func(u_grid_prediction, u_grid)
        v_loss = loss_func(v_grid_prediction, v_grid)
        u_loss.backward(retain_graph=True)
        v_loss.backward(retain_graph=True)

        # Compute physics loss
        dudx = torch.autograd.grad(
            u_grid_prediction,
            x_grid,
            grad_outputs=torch.ones_like(u_grid_prediction),
            create_graph=True
        )[0]
        dvdy = torch.autograd.grad(
            v_grid_prediction,
            y_grid,
            grad_outputs=torch.ones_like(v_grid_prediction),
            create_graph=True
        )[0]
        divergence = dudx + dvdy
        physics_loss = loss_func(divergence, torch.zeros_like(divergence))

        # Compute total loss
        test_loss[0] += u_loss.item() + v_loss.item() + physics_weight * physics_loss.item()
        data_loss[0] +=u_loss.item() + v_loss.item()

linear_test_loss = linear_test_loss[0]
linear_data_loss = linear_data_loss[0]
tg_test_loss = tg_test_loss[0]
tg_data_loss = tg_data_loss[0]

print(f"Linear Potential Vortex Model Test Loss: {linear_test_loss:.3e}")
print(f"Linear Potential Vortex Model Data Loss: {linear_data_loss:.3e}")
print(f"Taylor-Green Potential Vortex Model Test Loss: {tg_test_loss:.3e}")
print(f"Taylor-Green Potential Vortex Model Data Loss: {tg_data_loss:.3e}")
```

```
Linear Potential Vortex Model Test Loss: 1.955e-03
Linear Potential Vortex Model Data Loss: 2.268e-04
Taylor-Green Potential Vortex Model Test Loss: 1.100e+00
Taylor-Green Potential Vortex Model Data Loss: 1.068e+00
```

Linear Interpolation Benchmark

In [16]:

```python
class InterpUV():
    """Vector-valued 2-D interpolator"""
    def __init__(self, x_vec, y_vec, u_grid, v_grid):
```

```
        self.u_interp = RegularGridInterpolator([x_vec.T, y_vec.T], u_grid.T, bounds_error=False, fill_valu
        self.v_interp = RegularGridInterpolator([x_vec.T, y_vec.T], v_grid.T, bounds_error=False, fill_valu

    def __call__(self, x, y):
        u = self.u_interp((x, y))
        v = self.v_interp((x, y))
        return u, v
```

In [17]:

```
# Interpolate linear potential vortex
interp_linear = InterpUV(x_vec_train_linear, y_vec_train_linear, u_grid_train_linear, v_grid_train_linear)
u_test_interp_linear, v_test_interp_linear = interp_linear(x_grid_test_linear, y_grid_test_linear)

# Interpolate Taylor-Green potential vortex
interp_tg = InterpUV(x_vec_train_tg, y_vec_train_tg, u_grid_train_tg, v_grid_train_tg)
u_test_interp_tg, v_test_interp_tg = interp_tg(x_grid_test_tg, y_grid_test_tg)
```

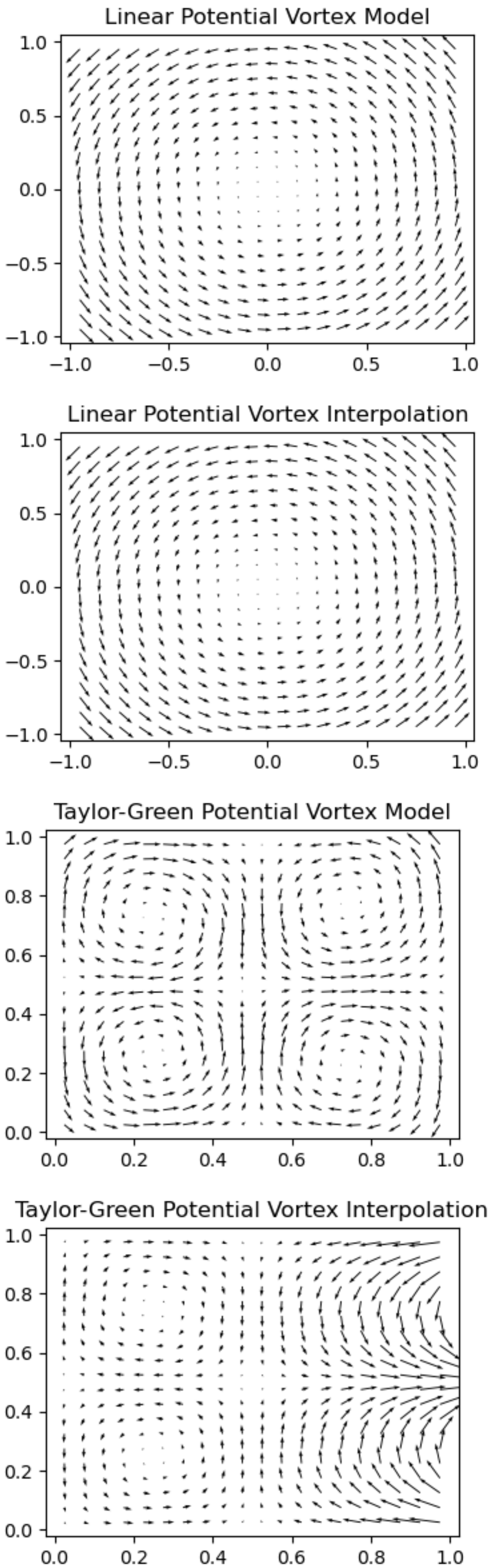
In [18]:

```
# Loss of linear potential vortex interpolation
u_loss_interp_linear = loss_func(torch.from_numpy(u_test_interp_linear), u_test_tensor_linear)
v_loss_interp_linear = loss_func(torch.from_numpy(v_test_interp_linear), v_test_tensor_linear)
data_loss_interp_linear = u_loss_interp_linear + v_loss_interp_linear

# Loss of Taylor-Green potential vortex interpolation
u_loss_interp_tg = loss_func(torch.from_numpy(u_test_interp_tg), u_test_tensor_tg)
v_loss_interp_tg = loss_func(torch.from_numpy(v_test_interp_tg), v_test_tensor_tg)
data_loss_interp_tg = u_loss_interp_tg + v_loss_interp_tg

print(f"Linear Potential Vortex Interpolation Data Loss: {data_loss_interp_linear:.3e}")
print(f"Taylor-Green Potential Vortex Interpolation Data Loss: {data_loss_interp_tg:.3e}")
```

```
Linear Potential Vortex Interpolation Data Loss: 1.573e-31
Taylor-Green Potential Vortex Interpolation Data Loss: 2.180e+00
```

In [19]:

```
# Linear potential vortex
xlim = (-0.95, 0.95)
ylim = (-0.95, 0.95)
nx = 20
ny = 20
N_train_linear = nx * ny
_x = np.linspace(*xlim, nx)
_y = np.linspace(*ylim, ny)
x, y = np.meshgrid(_x, _y)
xy_grid = torch.cat(
    (torch.from_numpy(x).float().unsqueeze(-1), torch.from_numpy(y).float().unsqueeze(-1)),
    dim=-1,
)
uv = linear_model(xy_grid)
u = uv[:, :, 0].detach().numpy()
v = uv[:, :, 1].detach().numpy()
_ = plot_flow_field(x, y, u, v, "Linear Potential Vortex Model")
plt.savefig("fig2d_linear_model.pdf")
_ = plot_flow_field(x, y, *interp_linear(x, y), "Linear Potential Vortex Interpolation")
plt.savefig("fig2d_linear_interp.pdf")

# Taylor-Green potential vortex
xlim = (0.025, 0.975)
ylim = (0.025, 0.975)
nx = 20
ny = 20
N_train_linear = nx * ny
_x = np.linspace(*xlim, nx)
_y = np.linspace(*ylim, ny)
x, y = np.meshgrid(_x, _y)
xy_grid = torch.cat(
    (torch.from_numpy(x).float().unsqueeze(-1), torch.from_numpy(y).float().unsqueeze(-1)),
    dim=-1,
)
uv = tg_model(xy_grid)
u = uv[:, :, 0].detach().numpy()
v = uv[:, :, 1].detach().numpy()
_ = plot_flow_field(x, y, u, v, "Taylor-Green Potential Vortex Model")
plt.savefig("fig2d_tg_model.pdf")
_ = plot_flow_field(x, y, *interp_tg(x, y), "Taylor-Green Potential Vortex Interpolation")
plt.savefig("fig2d_tg_interp.pdf")
```


Linear Potential Vortex Model
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
Linear Potential Vortex Interpolation
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
Taylor-Green Potential Vortex Model
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
Taylor-Green Potential Vortex Interpolation
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0