MIMO for Matlab: A Toolbox for Simulating MIMO Communication Systems

Ian P. Roberts

***Abstract*—We present MIMO for Matlab (MFM), a software package for Matlab that aims to simplify the simulation of multiple-input multiple-output (MIMO) communication systems research while facilitating reproducibility, consistency, and community-driven customization. MFM offers users an object-oriented solution for simulating a variety of MIMO systems including conventional sub-6 GHz, massive MIMO, millimeter wave, and terahertz communication. Out-of-the-box, MFM supplies users with a variety of widely used channel and path loss models from academic literature and cellular and local area network standards; if a particular channel or path loss model is not provided by MFM, custom models can be created and integrated by following a few simple rules. The complexity and overhead associated with simulating networks of multiple devices can be severely lowered with MFM versus raw Matlab code, especially when users want to investigate various channel models, path loss models, precoding/combining schemes, or other system-level parameters. MFM's heavy-lifting to automatically collect and distribute channel state information, compute interference, and report performance metrics relieves users of otherwise tedious tasks and instills confidence and consistency in the results of simulation. The use-cases of MFM vary widely from networks of hundreds of devices; to simple point-to-point communication; to serving as a channel generator; to radar, sonar, and underwater acoustic communication.**

I. Introduction

Research and education on multiple-input multiple-output (MIMO) communication systems are built on linear equations of the form

$$\hat{\mathbf{s}} = \sqrt{P_{\rm tx}} \cdot G \cdot \mathbf{W}^{*}\mathbf{H}\mathbf{F}\mathbf{s} + \mathbf{W}^{*}\mathbf{n} \tag{1}$$

sometimes termed "symbol-level" or "single-letter" formulations. To communicate a symbol vector $\mathbf{s}$ over some channel matrix $\mathbf{H}$, a transmitter with power $P_{\rm tx}$ applies a precoding matrix $\mathbf{F}$ while a receiver applies a combining matrix $\mathbf{W}$ to recover an estimate of the symbol vector $\hat{\mathbf{s}}$. Along the way, path loss $1/G^2$ weakens the transmitted signal and additive noise $\mathbf{n}$ further corrupts the received signal. While these linear models greatly simplify the sophistication of today's communication systems, simulating MIMO concepts and research can become prohibitively complex and overwhelming when the system and/or network grow to even a moderate size.

This has motivated us to create MIMO for Matlab (MFM)[1], a toolbox for simulating MIMO communication systems. MFM is written in an object-oriented fashion and comes with a collection physical layer tools including a variety of channel models, transmitters, receivers, and antenna arrays that can be used for sub-6 GHz, millimeter wave (mmWave), terahertz (THz), and beyond.

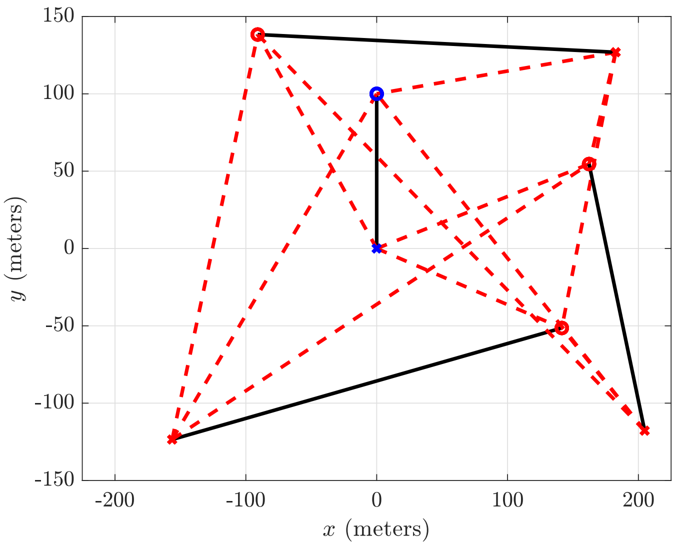

Fig. 1. A network of eight devices scattered in space simulated in MFM. Four transmit-receive pairs (shown as $\times$'s and $\circ$'s, respectively) use the same time-frequency resources in their attempt to individually communicate. In this context, the blue transmit-receive pair is considered the "desired" link while the other three links are considered interference. The transmit signal from each of the three transmitting interferers (red $\times$'s) plagues the desired receiver (blue $\circ$) with interference.

MFM has support from the antenna/spatial domain all the way up to a network of users. By design, MFM has been created to be used at any level within its capabilities. For instance, MFM can be used at its lowest level for antenna array research, as a channel simulator, or as a path loss simulator. At its highest level, MFM can be used to simulate a network of many users and automatically compute effects they have on each other due to interference. In between, a simple point-to-point communication link can be simulated, allowing users to develop, implement, and evaluate novel precoding and combining schemes.

A. Modern Communication Systems Support

MFM supports conventional fully-digital MIMO transceivers, such as those typical in sub-6 GHz systems. Next-generation communication systems—such as massive MIMO, mmWave, and THz communication—rely on hybrid digital/analog beamforming. MFM supports hybrid digital/analog transceivers out-of-the-box and includes

I. P. Roberts is with the Wireless Networking and Communications Group at the University of Texas at Austin, Austin, Texas, USA. Corresponding author: I. P. Roberts (ipr@utexas.edu). Last updated: January 21, 2021.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1610403. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[1]Available online at `https://mimoformatlab.com`.

channel and path loss models for simulating tomorrow's systems. Moreover, it supports fully-connected and arbitrary partially-connected hybrid beamforming architectures, such as sub-array architectures. In addition, MFM captures phase shifter resolution and attenuator resolution (if desired) that exists digitally-controlled analog beamforming networks; infinite resolution (e.g., analog-controlled) phase shifters and attenuators can also be implemented if desired.

B. A Common Object-Oriented Framework

By using MFM as a common framework across the research community, researchers can share their MFM scripts and objects to facilitate reproducibility, broadening the impact of their work, and instilling confidence in their results. Thanks to its object-oriented design, MFM objects created by users can easily be easily shared and implemented across the research community. The network from a user's MFM simulation, for example, can easily be shared by simply exporting the simulation's network object. Arrays, transmitters, and receivers can be created, for instance, to model (to a degree) their practical counterparts and can then likewise be shared across the community.

C. Customization and Expansion

Through its object-oriented design, MFM was designed to accommodate customizations and expansions that a user sees fit. For example, if a particular channel model that a user needs is not provided in MFM, users can create their own by following a few simple rules. Once created, the custom channel model can be easily shared and then incorporated into MFM by others across the research community. If particular customizations/additions to MFM are widely used, there are avenues for them to be incorporated into future versions of MFM.

D. Computation of Performance Metrics

System performance can be evaluated automatically by MFM behind-the-scenes. Metrics such as mutual information and symbol estimation error are computed by MFM and its ability to aggregate contributions of interference network-wide.

E. Beyond the Physical Layer

While MFM currently only executes communication systems at the physical layer, its framework facilitates other areas of research. For example, research on scheduling or stochastic geometry can leverage MFM's abstraction and handling of the physical layer implementations, allowing such users to focus solely on the scope of their research without the headache of implementing physical layer communication network-wide.

F. Resources and Documentation

MFM has been extensively documented online at `https://mimoformatlab.com`. Where possible, mathematical descriptions of MFM's components and functions have been provided. In addition, a collection of video tutorials have been published to YouTube and included on the MFM website. A variety of example scripts are included with MFM, all of which have companion video tutorials and written guides, to help acquaint users with typical MFM usage at its different levels. The resources and documentation surrounding MFM will continue to grow.

G. Citing MFM

We are interested in tracking the reach it has and applications it serves to better improve MFM in the future. If you use MFM, please cite this paper and also the package itself using

```
@misc{mfm,
  author = {Ian P. Roberts},
  title  = {{MIMO} for {MATLAB}: A
            Toolbox for Simulating
            {MIMO} Communication
            Systems in {MATLAB}},
  howpublished =
       {\url{http://mimoformatlab.com}},
  month = nov,
  year = 2020
}
```

II. Overview

MFM is a collection of MATLAB scripts that can be used together, to varying degrees, to simulate MIMO communication systems. The MFM framework simplifies generating channels/network realizations, executing precoding and combining strategies, and evaluating communication system performance. With MFM, users can focus their attention on the aspects of MIMO communication that are *relevant to them* since MFM can handle the rest. For example, users interested in creating MIMO precoding and combining strategies may want to examine their strategies across many channel and path loss models. MFM can enable such by providing a collection of common channel and path loss models, which can be used interchangeably network-wide with ease. In addition, MFM's heavy-lifting can relieve users of the headache associated with tasks such as computing interference and collecting channel state information, which grow daunting and overwhelming with networks of moderate size.

A. Objects

MFM was created in an object-oriented fashion, and thus, its power lay in its objects summarized as follows:

- The `array` object is used to represent antenna arrays. Antenna arrays in MFM can be constructed as uniform linear arrays (ULAs), uniform planar arrays (UPAs), or any other arbitrary array the user wishes.
- The `channel` objects represent over-the-air channels between transmit antennas and receive antennas. MFM supplies users with a collection of widely used channel models and supports user creation of models not included out-of-the-box.

- The path_loss objects represent over-the-air path loss experienced by a signal propagating from a transmitter to a receiver. Like the channel models, MFM supplies users with a collection of widely used path loss models and supports the creation of additional ones.
- The transmitter object is used to represent a fully-digital MIMO transmitter. The transmitter_hybrid object is used to represent a hybrid digital/analog MIMO transmitter. Precoding, applying a transmit power, and ultimately transmitting a symbol vector are all the responsibility of the transmitter and transmitter_hybrid objects.
- The receiver object is used to represent a fully-digital MIMO receiver. The receiver_hybrid object is used to represent a hybrid digital/analog MIMO receiver. Combining, introducing additive noise, and ultimately estimating a symbol vector are all the responsibility of the receiver and receiver_hybrid objects.
- The device object represents a wireless terminal that has transmit and/or receive capability. Those having both are termed *transceivers*.
- The link object captures propagation between a transmitting device and a receiving device. A channel and path_loss objects of a link are used to describe this propagation. Since link objects connect device objects—rather than transmitter and receiver objects directly—they actually have a pair of channel objects and a pair of path_loss objects. One channel-path_loss pair is for the *forward* link from the first device's transmitter to the second device's receiver. When applicable, the second channel-path_loss pair is for the *reverse* link from the second device's transmitter to the first device's receiver.
- The network_mfm object is the collection of device objects and the link objects connecting them.

B. Using MFM

MFM can be thought of as having a four-layer hierarchy as illustrated by the example in Fig. 2. At the foundation of MFM are antenna arrays and channels. Antenna arrays are useful on their own for applications in surrounding array signal processing. array and channel objects together can be used for simple channel generation. Moving up a layer, we have transmitters and receivers, which can be used by themselves for operations such as precoder power normalization, constraints in analog beamforming, and introducing additive noise. Up another layer are the device and link objects, which can be used directly to conveniently simulate simple systems of up to a few devices. Finally, at the highest layer of MFM exists the network_mfm object which is used to aggregate many devices and their links (both desired and interference). The network_mfm object is particularly useful in its ability to execute precoding and combining network-wide, set network-wide channel and path loss models, compute interference arriving at each receiver, collect and distribute channel state information, and report performance metrics.

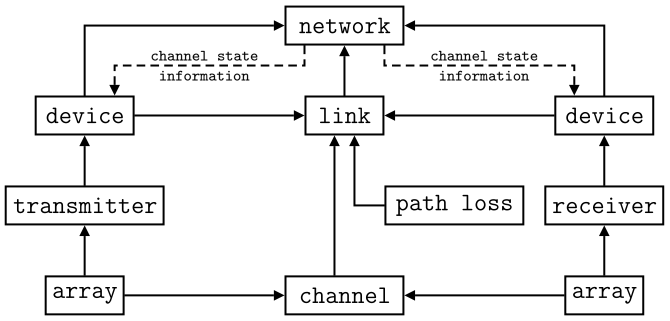

Fig. 2. Example architecture of an MFM script simulating point-to-point communication between a transmitting device and receiving device.

C. Notes on Object-Oriented Programming in MATLAB

Since MFM is object-oriented, it is important to understand some basic concepts of object-oriented programming in MATLAB, which has similarities to other languages like Python. Each object's methods are either "static" or "ordinary". Static methods are those that can be called without needing an instance of the object; in other words, they do not take the object as an input argument. Ordinary methods, which are far more common in MFM than static methods, are called on an instance of the object, meaning they require the object as an input argument.

It is very important that users of MFM understand how MATLAB copies objects and passes them functions. In general, MATLAB copies objects so-called "by reference", where assignment of a variable as an existing object does not create a new object but rather a reference to it. Changes made to the object will be reflected in both the "original" object and its reference. In some cases, MFM will copy the object so-called "by value", where the copied object is identical to the existing one (at the time of copying) but does not have any underlying reference/connection, meaning changes can be made to one without affecting the other. To copy objects by value, MFM has included the function val = copy_object(obj), which returns a copy of obj by value in val. Note that everything within obj is copied by value, meaning any of its properties that may be objects are also copied by value.

D. Downloading and Setting Up MFM

The latest version of MFM can be cloned via Git using

```
git clone git@gitlab.com:iprnq9/mfm.git
```

or downloaded directly from its GitLab page[2]. Once downloaded, to begin using MFM, simply open MATLAB, navigate to within the mfm/ directory, and execute mfm.setup() in the command window.

E. Important Conventions and Practices

To improve consistency, MFM employs several conventions and practices, which we summarize as follows:

- All angular values are in radians.

[2]https://gitlab.com/iprnq9/mfm

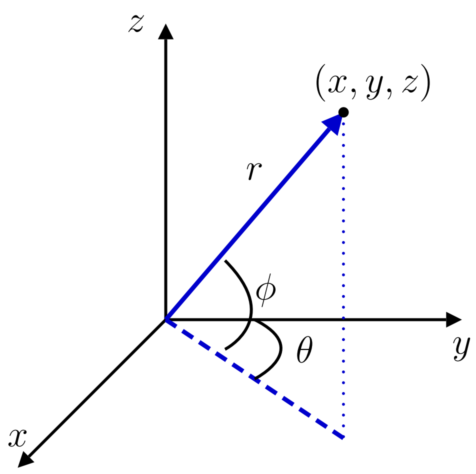

Fig. 3. The 3-D geometry conventions used by MFM.

- All functions and variables are in so-called "snake_case".
- Object functions commonly start with `set_`, `get_`, `check_`, `show_`, `enforce_`, and `compute_`.

As described by Fig. 3, MFM uses an azimuth-elevation convention to describe directions in 3-D. MFM defines azimuth—typically referred to as θ—as the angle from the y-axis to the projection of the vector of interest onto the x-y plane. Azimuth angles range from $-\pi$ to π. Elevation—typically reffered to as ϕ—is defined as the angle from the x-y plane to the vector of interest. Elevation angles range from $-\pi/2$ to $\pi/2$.

Explicitly, the 3-D geometry can be summarized as follows, where r is the radial distance, θ is the azimuth angle, and ϕ is the elevation angle. A vector of length r in the azimuth-elevation direction (θ, ϕ) can be converted to Cartesian components (x, y, z) as

$$x = r \sin\theta \cos\phi \tag{2}$$
$$y = r \cos\theta \cos\phi \tag{3}$$
$$z = r \sin\phi \tag{4}$$

A vector having Cartesian components (x, y, z) can be described as having length r in the azimuth-elevation direction (θ, ϕ) via

$$r = \sqrt{x^2 + y^2 + z^2} \tag{5}$$
$$\theta = \arctan\left(\frac{x}{y}\right) \tag{6}$$
$$\phi = \arctan\left(\frac{z}{\sqrt{x^2 + y^2}}\right) \tag{7}$$

F. Documentation

The entire MFM toolbox has been documented inline, meaning users can leverage MATLAB's `help` command to learn more about a function or object. The inline documentation for each function, for example, includes a description of the function, example usage, input arguments, return values, and function notes. In addition, documentation for the entire MFM package along with examples, how-to's, and clarifying remarks are available on its website `https://mimoformatlab.com`.

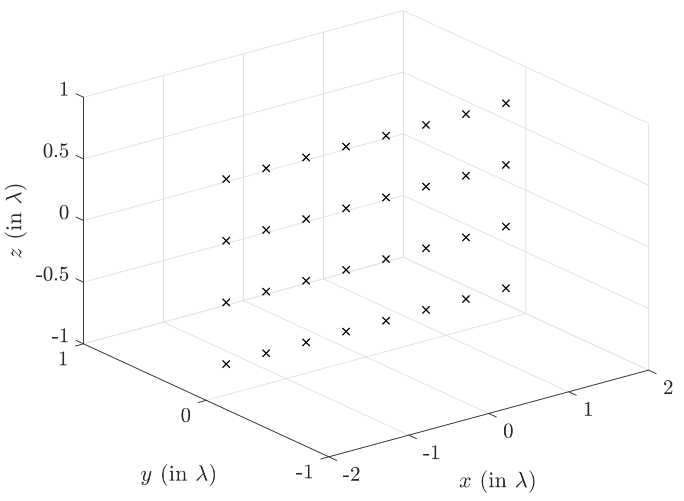

Fig. 4. A 4×8 UPA created in MFM.

III. Array Object

The `array` object, as one may expect, is at the core of MFM. Arrays can be constructed in a few different ways:

- `a = array.create()` creates an empty array with no elements, after which elements can be added to the array.
- `a = array.create(N,ax)` creates a half-wavelength, ULA with `N` elements where `ax` is either `'x'` (default), `'y'`, or `'z'` specifying which axis to create the ULA along.
- `a = array.create(M,N,plane)` creates a half-wavelength, UPA with `M` rows of `N` elements where `plane` is either `'xz'` (default), `'xy'`, or `'yz'` specifying which plane to create the UPA in.

A. Viewing the Array

The elements of an array `a` can be viewed in 2-D using `a.show_2d()` or `a.show_2d([],plane)` where `plane` is either `'xz'` (default), `'xy'`, or `'yz'` specifying which plane to show. Its elements can also be viewed in 3-D using `a.show_3d()` as was used in Fig. 4.

As evidenced by Fig. 4, the location of an array's elements are defined in units of carrier wavelengths. This is attributed to the fact that an array's behavior and characterization are (mostly[3]) described by its geometry relative to carrier wavelength. This convenience makes the antenna arrays in MFM agnostic to carrier wavelength.

B. Modifying an Array

The `array` object comes with several useful functions for modifying the array after it has been created. To simulate a typical MIMO communication system, creating an `array`

[3]Wideband array behavior may be included in future additions to MFM.

using the ULA or UPA method will often suffice, without significant modifications. MFM, however, does give users the ability to modify the array in various ways, which may be particularly useful when using the `array` object on its own or in settings not strictly related to MIMO (e.g., array signal processing, radar, sonar).

1) Adding Elements: To add elements to an antenna array `a`, one can use `a.add_element(x,y,z)`, where `x`, `y`, and `z` are vectors of x, y, and z coordinates (in wavelengths) for each element to be added.

2) Removing Elements: To remove elements individually, one can use `a.remove_element(idx)`, where `idx` is the index of the element to be removed; if `idx` is not passed, the last element of the array will be removed.

3) Translating the Array: To translate the array in space by some `x`, `y`, and `z` (in wavelengths), one can use

```
a.translate(x,y,z)
```

If no arguments are passed, then `a.translate()` will center the array at the origin.

4) Rotating the Array: To rotate the array along the x, y, and z axes by some `theta_x`, `theta_y`, and `theta_z` radians, respectively, one can use

```
a.rotate(theta_x,theta_y,theta_z)
```

C. Array Response

Some array configurations—such as half-wavelength uniform linear and planar arrays—have well-known expressions for their response as a function of direction. While such array configurations also happen to be the most commonly used, MFM supports arbitrary antenna arrays. In other words, MFM does not restrict the type of arrays a user can construct. To do so, MFM computes the array response numerically based on the relative positioning of the array elements. MFM uses the convention that the array response is of the form as follows.

The relative phase shift experienced by the i-th array element located at some (x_i, y_i, z_i) from the origin due to a plane wave in the direction (θ, ϕ) is

$$a_i(\theta,\phi) = \exp\left(\mathrm{j}\cdot\frac{2\pi}{\lambda}\cdot\zeta(x_i,y_i,z_i,\theta,\phi)\right) \quad (8)$$

where λ is the carrier wavelength and

$$\zeta(x,y,z,\theta,\phi) = x\sin\theta\cos\phi + y\cos\theta\cos\phi + z\sin\phi \quad (9)$$

Instead of referencing the true origin $(0,0,0)$ to compute the array response, MFM refers the location of each antenna element to that of the *first element* in the antenna array. Thus, the array response vector is constructed by collecting the relative phase shift seen by each of the array's N_{a} elements as

$$\mathbf{a}(\theta,\phi) = \begin{bmatrix} a_1(\theta,\phi) \\ a_2(\theta,\phi) \\ \vdots \\ a_{N_{\mathrm{a}}}(\theta,\phi) \end{bmatrix} \cdot \frac{1}{a_1(\theta,\phi)} \quad (10)$$

While MFM could use the true origin $(0,0,0)$ as the relative origin, this would require the array to have knowledge of its location in 3-D space; to make the `array` object agnostic of such, we have chosen this convention. This merely shifts each transmit/receive signal across antennas by the same amount and does not disturb the underlying importance of the array response itself: the *relative* phase shifts across antennas.

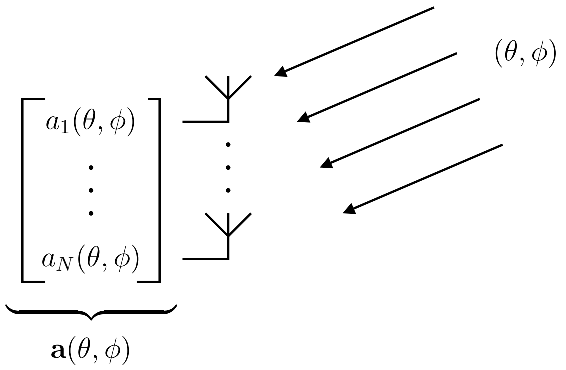

Fig. 5. The array response.

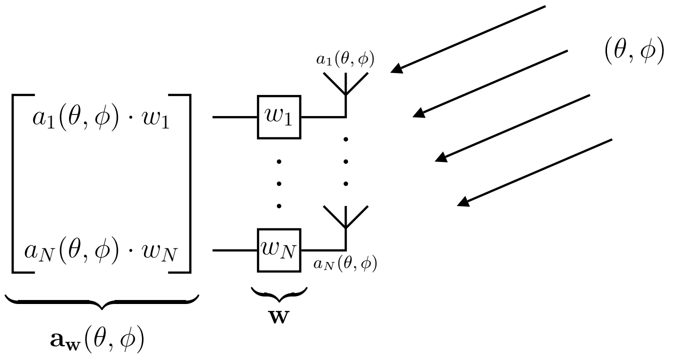
(θ,ϕ)
$a_1(\theta,\phi)$
$a_1(\theta,\phi)\cdot w_1$
w_1
$a_N(\theta,\phi)\cdot w_N$
w_N
$a_N(\theta,\phi)$
$\mathbf{w}$
$\mathbf{a}_{\mathbf{w}}(\theta,\phi)$

Fig. 6. The weighted array response.

The array response of an array `a` in a particular azimuth θ and elevation ϕ can be obtained via

```
v = a.get_array_response(theta,phi)
```

D. Array Weights and Beamforming

To weight the N_{a} elements of an antenna array `a`, one can use `a.set_weights(w)`, where `w` is a vector of N_{a} complex weights. With these weight applied, the *weighted* array response $\mathbf{a}_{\mathbf{w}}(\theta,\phi)$ defined as

$$\mathbf{a}_{\mathbf{w}}(\theta,\phi) = \begin{bmatrix} a_1(\theta,\phi)\cdot w_1 \\ a_2(\theta,\phi)\cdot w_2 \\ \vdots \\ a_{N_{\mathrm{a}}}(\theta,\phi)\cdot w_{N_{\mathrm{a}}} \end{bmatrix} \cdot \frac{1}{a_1(\theta,\phi)} \quad (11)$$

can be obtained via

```
v = a.get_weighted_array_response(az,el)
```

Note that the weights contained in `w` are applied as is and are not conjugated beforehand. This can be described mathematically by stating that the gain of an array with weights $\mathbf{w}$ in the direction (θ,ϕ) is

$$g(\theta,\phi) = \mathbf{w}^{\mathrm{T}}\mathbf{a}(\theta,\phi) \quad (12)$$

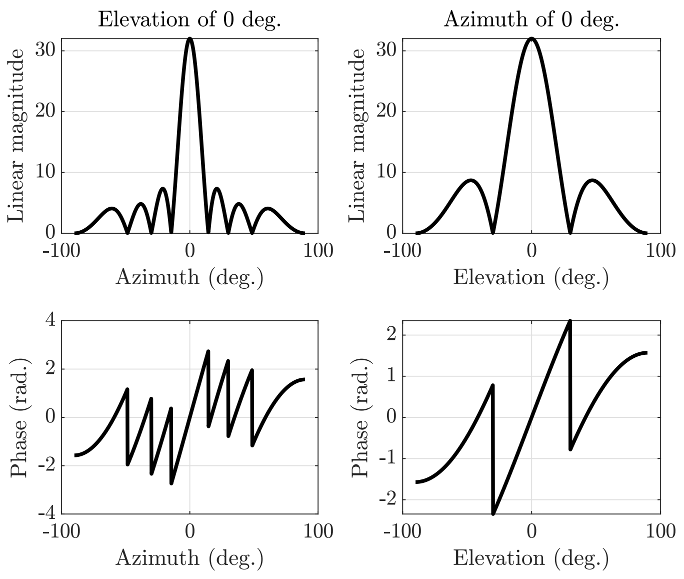

Fig. 7. The azimuth and elevation array patterns of a 4×8 UPA. As expected, the azimuth pattern is sharper given the UPA has twice as many elements horizontally as it does vertically.

where $(\cdot)^{\mathrm{T}}$ denotes transpose (not conjugate transpose). Therefore, to so-called conjugate beamform (i.e., matched filter) in the direction of (θ, ϕ), one would take $\mathbf{w} = \mathbf{a}(\theta, \phi)^{\mathrm{c}}$, where $(\cdot)^{\mathrm{c}}$ denotes element-wise conjugation. To achieve this in MFM, this would simply be

```
v = a.get_array_response(theta,phi)
w = conj(v)
a.set_weights(w)
```

To evaluate the gain achieved by a weighted array `a` in the direction (`theta`,`phi`), one could use the following.

```
a.get_array_gain(theta,phi)
```

As will be discussed, the precoding and combining executed by MFM does not use this beamforming feature of `array` objects, even hybrid digital/analog precoding and combining architectures. Instead, MFM executes precoding and combining at the `transmitter` and `receiver` objects, respectively.

E. Plotting the Array Pattern

An extremely convenient feature of the `array` object in MFM is its plotting functionality. Once an array `a` is created, its array pattern can be viewed in a variety of ways. A few particularly useful ways are:

- `a.show_array_pattern()` displays the array's magnitude and phase responses as a function of azimuth and elevation angles in each of their respective cuts.
- `a.show_polar_array_pattern_azimuth()` displays the array's magnitude response in polar form as a function of azimuth angle at an elevation angle of 0.
- `a.show_polar_array_pattern_elevation()` displays the array's magnitude response in polar form as a function of elevation angle at an azimuth angle of 0.

F. Important Conventions

Note that MFM uses the receive array response convention, as evidenced by the positive j term rather of a negative one in (8). Users should be cautious of this if using the `array` object outside of MFM. In addition, it should be noted that MFM does not normalize the array response vector to unit norm, as is sometimes convention. This choice was made to accurately capture the physical meaning behind the array response induced by a planar wave front.

IV. Channel Objects

The `channel` objects in MFM are used to capture the over-the-air mixing that takes place across transmit antennas and receive antennas, leading to a channel matrix $\mathbf{H}$. Following convention in MIMO literature, channel matrices $\mathbf{H}$ in MFM are always of size $N_{\mathrm{r}} \times N_{\mathrm{t}}$, where N_{r} antennas are at the receiver and N_{t} antennas are at the transmitter. Currently, MFM only supports these frequency-flat channels but work is ongoing to extend its support to frequency-selective ones. It is important to keep in mind that while MFM has MIMO in its name, it also supports single-input single-output (SISO) scenarios to an extent, particularly in regards to some channel models (e.g., the Rayleigh-faded channel).

A. Provided Channel Models

MFM currently provides the following common channel models out-of-the-box, making it convenient for users to generate model-based channel matrices.

1) Rayleigh-Faded Channel: A Rayleigh-faded channel can be created via `c = channel.create('Rayleigh')`, whose entries are drawn from a standard complex Normal distribution, described as

$$[\mathbf{H}]_{i,j} \sim \mathcal{N}_{\mathbb{C}}(0,1) \ \forall \ i,j \tag{13}$$

2) LOS Channel: A far-field line-of-sight (LOS) channel, comprised of a single, direct path between a transmitter and receiver, can be constructed via `c = channel.create('LOS')`.

$$\mathbf{H} = \beta \cdot \mathbf{a}_{\mathrm{rx}}(\mathrm{AoA})\, \mathbf{a}_{\mathrm{tx}}(\mathrm{AoD})^{*} \tag{14}$$

3) Rician Channel: A Rician-faded channel can be constructed via `c = channel.create('Rician')`, which can be described as the mixture between a LOS channel $\mathbf{H}^{\mathrm{LOS}}$ and a Rayleigh-faded channel $\mathbf{H}^{\mathrm{Ray}}$ via

$$\mathbf{H} = \sqrt{\frac{\kappa}{\kappa+1}}\mathbf{H}^{\mathrm{LOS}} + \sqrt{\frac{1}{\kappa+1}}\mathbf{H}^{\mathrm{Ray}} \tag{15}$$

where the Rician factor κ captures the amount of power in the LOS portion relative to the Rayleigh-faded portion.

4) Ray/Cluster Channel: A ray/cluster channel, comprised of clusters of discrete rays, can be constructed via `c = channel.create('ray-cluster')`.

$$\mathbf{H} = \sqrt{\frac{1}{N_{\mathrm{ray}}N_{\mathrm{cl}}}} \sum_{u=1}^{N_{\mathrm{ray}}} \sum_{v=1}^{N_{\mathrm{cl}}} \beta_{uv} \mathbf{a}_{\mathrm{rx}}(\mathrm{AoA}_{uv})\, \mathbf{a}_{\mathrm{tx}}(\mathrm{AoD}_{uv})^{*} \tag{16}$$

5) Spherical-Wave Channel: A spherical-wave channel, often used to represent ideal near-field propagation, can be constructed via `c = channel.create('spherical-wave')`.

$$[\mathbf{H}]_{v,u} = \frac{\gamma}{r_{u,v}} \exp\left(-\mathrm{j}2\pi \frac{r_{u,v}}{\lambda}\right) \tag{17}$$

where $r_{u,v}$ is the distance between the u-th transmit antenna and the v-th receive antenna, λ is the carrier wavelength, and γ ensures that the channel is normalized such that $\mathbb{E}\left[\|\mathbf{H}\|_{\mathrm{F}}^2\right] = N_{\mathrm{t}}N_{\mathrm{r}}$. Note that this near-field model is deterministic for a given relative transmit-receive array geometry.

B. Setting the Propagation Velocity

The propagation velocity, often taken to be 3×10^8 m/s for electromagnetic propagation, can be set using

```
c.set_propagation_velocity(vel)
```

where `vel` is the propagation velocity (in m/s). While MFM was created for conventional electromagnetic-based wireless communication, affording users the ability to set the propagation velocity may lend MFM support to other fields such as underwater acoustic communication where the propagation velocity of sound in the ocean is often taken to be around 1.5×10^3 m/s, for example.

C. Setting the Carrier Frequency

The carrier frequency of the signals propagating through a channel are set using

```
c.set_carrier_frequency(fc)
```

where `fc` is the carrier frequency (in Hz). Setting the carrier frequency will automatically compute the carrier wavelength according to the propagation velocity.

D. Setting the Transmit and Receive Arrays

Informing `channel` objects of the transmit and receive arrays informs MFM of the size of the channel matrix and also provides geometric channel models—such as the LOS channel and ray/cluster channel—with the array responses automatically. To do this, simply use

```
c.set_arrays(array_transmit,array_receive)
```

where `array_transmit` and `array_receive` are the transmit and receive array objects, respectively.

E. Enforcing a Strict Channel Energy Normalization

Since the energy that a channel matrix is normalized to can distort interpretations and conclusions drawn when considering values such as large-scale signal-to-noise ratio (SNR), it is common to normalize the energy of a channel matrix on average or on each realization. MFM supports the latter, normalizing each channel matrix realization to a fixed energy (squared Frobenius norm), whereas normalizing the channel energy on average is left up to each specific channel implementation. However, it should be noted that all channels in MFM are normalized such that

$$\mathbb{E}\left[\|\mathbf{H}\|_{\mathrm{F}}^2\right] = N_{\mathrm{t}} \cdot N_{\mathrm{r}} \tag{18}$$

Users can scale the realizations accordingly if they desire a different average energy normalization.

To enforce a strict channel energy normalization *for each realization*, MFM's `channel` objects are all equipped with the following.

```
c.set_force_channel_energy...
  _normalization(force)
```

where `force` is `true` to force energy normalization and `false` to not. By default, this will force each channel matrix $\mathbf{H}$ to be normalized such that

$$\|\mathbf{H}\|_{\mathrm{F}}^2 = N_{\mathrm{t}} \cdot N_{\mathrm{r}} \tag{19}$$

To change the value that the energy of the channel realizations are normalized to, simply use

```
c.set_normalized_channel_energy(val)
```

where `val` is the energy that the channel will be normalized to (instead of $N_{\mathrm{t}} \cdot N_{\mathrm{r}}$).

F. Channel-Specific Setup

Setting the propagation velocity, carrier frequency, transmit and receive arrays, and energy normalization is common across all channel models. Beyond this, the setup associated with each channel is unique. For example, setting the Rician factor κ is necessary in the Rician channel but not other channels. For a complete overview of each channel and its setup, please refer to the MFM website and additional resources.

G. Invoking a Channel Realization

Once a `channel` object `c` has been created and properly set up, a realization of the channel is merely one line of code.

```
H = c.realization()
```

This is especially convenient for Monte Carlo simulations, where channel realizations are placed within a loop, as below.

```
for i = 1:N
    ...
    H = c.realization()
    ...
end
```

Any stochastics associated with the channel model will be redrawn from their respective distributions when constructing the channel matrix on each realization.

H. Creating Custom Channel Models

Users can create custom channel models as needed but are required to follow a few guidelines to ensure they integrate with the rest of MFM. First and foremost, any custom channel must be a child of the `channel` object. This is achieved by defining the custom channel object in the following fashion.

```
classdef channel_my_custom < channel
    ...
end
```

Custom channel objects should have names beginning with `channel_` and should be placed in the `obj/channels/` directory. To avoid issues, custom channel objects should avoid creating functions that supersede those found in the parent `channel` object. Recall that functions and properties of the `channel` object will be inherited by its subclasses. A custom channel object must contain a `realization()` function definition as follows.

```
function H = realization(obj)
    H = ... % realization definition
    obj.set_channel_matrix(H)
    H = obj.get_channel_matrix()
end
```

Once a custom channel has been made, it can be created via

```
c = channel_my_custom()
```

and can be used throughout MFM like any of the provided channel models. Note that a custom channel's setup should involve setting the propagation velocity, carrier frequency, and transmit and receive arrays, like all other channel models; these setup functions are inherited from the `channel` object. If interested in extending the `channel.create()` function with a custom string specifier for your custom channel (e.g., `c = channel.create('custom')`), please see visit the MFM website.

V. Path Loss Objects

Referencing the linear MIMO formulations of (1), MFM uses `path_loss` objects to handle the large-scale gain G between two devices. With deterministic path loss models, such as the classical Friis path loss formula, G can be calculated directly. With stochastic path loss models, such as those involving shadowing, G may depend on a random variable(s). MFM provides users with a variety of deterministic and stochastic path loss models and supports the ability for users to create their own custom path loss model.

A. Default Properties and Setup

To supply all path loss models with common parameters, each path loss model in MFM has the following properties that can be set as follows.

- The carrier frequency used by a `path_loss` object `p` can be set via `p.set_carrier_frequency(fc)`.
- The propagation velocity can be set via `p.set_propagation_velocity(vel)`.
- The carrier wavelength used by a `path_loss` object is set automatically when setting the carrier frequency and propagation velocity.
- The distance of the path can be set via `p.set_distance(d)`.

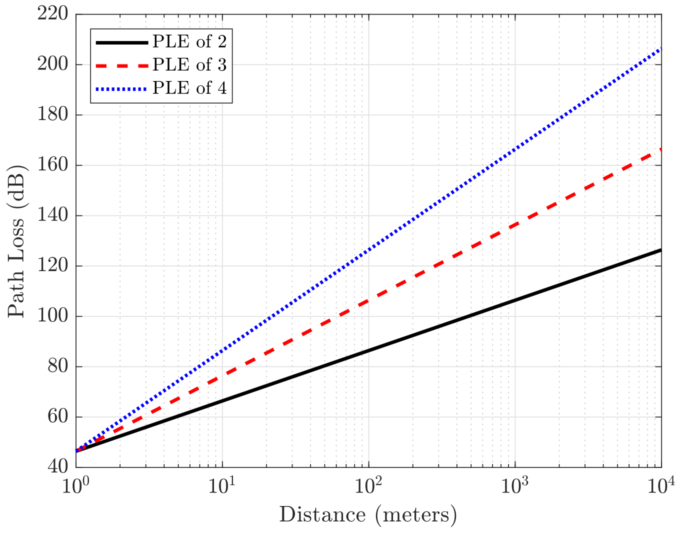

Fig. 8. Free-space path loss for $\eta = 2, 3, 4$.

B. Provided Path Loss Models

- `path_loss.create('FSPL')` creates an object capturing free-space path loss described as
$$G^2 = \left(\frac{\lambda}{4\pi}\right)^2 \times \left(\frac{1}{d}\right)^\eta \tag{20}$$
where λ is the carrier wavelength, d is the distance of the path, η is the path loss exponent, and $1/G^2$ is the power loss of the path. When users set $\eta = 2$, this resorts to the classical Friis path loss formula. The path loss exponent can be set via `p.set_path_loss_exponent(eta)`.
- To incorporate log-normal shadowing into the free-space path loss model, the following can be used
`p = path_loss.create('FSPL+LNS')`
which implements the following
$$G^2 = \left(\frac{\lambda}{4\pi}\right)^2 \times \left(\frac{1}{d}\right)^\eta \times \gamma \tag{21}$$
where $10 \cdot \log_{10}(\gamma) \sim \mathcal{N}\left(0, \sigma_\gamma^2\right)$ captures log-normal shadowing. The log-normal shadowing variance σ_γ^2 can be set via `p.set_log_normal_shadowing_variance(s)`.
- The two-slope path loss model described as
$$\frac{1}{G^2} = \begin{cases} L_0 \times \left(\frac{d}{d_0}\right)^{\eta_1}, & d \le d_0 \\ L_0 \times \left(\frac{d}{d_0}\right)^{\eta_2}, & d > d_0 \end{cases} \tag{22}$$
can be created via `path_loss.create('two-slope')`. Here, L_0 is the path loss at some reference distance d_0; within d_0, a path loss exponent of η_1 is used and beyond d_0, a path loss exponent of η_2 is used. To set these parameters, use
```
p.set_reference_distance(d0)
p.set_reference_path_loss(L0,'dB')
p.set_path_loss_exponents(ple_1,ple_2)
```

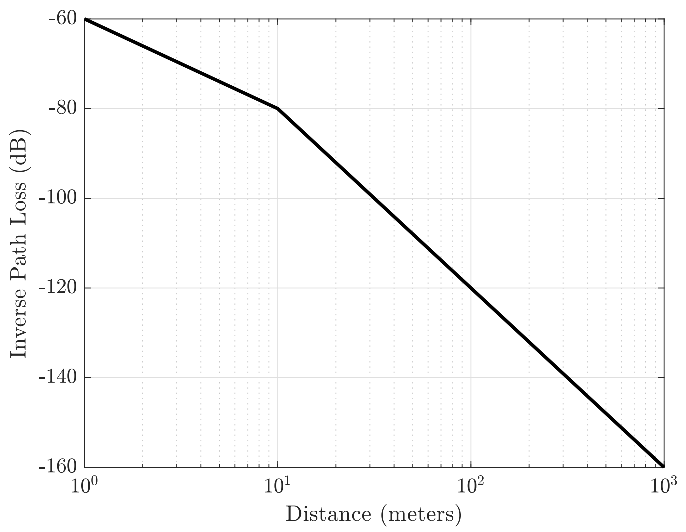

Fig. 9. Two-slope path loss where $d_0 = 10$ meters, $L_0 = 80$ dB, $\eta_1 = 2$, and $\eta_2 = 4$.

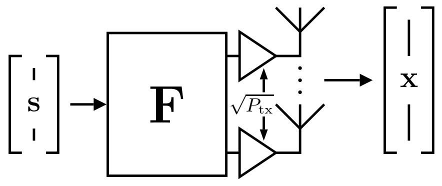

Fig. 10. A fully-digital transmitter applies a precoder $\mathbf{F}$ and a transmit gain $\sqrt{P_{\mathrm{tx}}}$ to a symbol vector $\mathbf{s}$ to transmit a vector $\mathbf{x}$.

C. *Invoking a Path Loss Realization*

Once a `path_loss` object `p` has been setup appropriately, the resulting path loss can be obtained via

```
L = p.realization()
```

where `L` is the linear power loss of the path. When `p` is a deterministic path loss model, `L` will not vary with each realization. When `p` is a stochastic path loss model, `L` will vary with each realization according to the underlying model.

VI. TRANSMITTER OBJECTS

A transmitter in MFM is captured by the `transmitter` object and its subclasses. A `transmitter` can be created via

```
tx = transmitter.create()
```

By default, a `transmitter` object employs fully-digital precoding. However, as we will discuss, MFM also supports hybrid digital/analog precoding. The symbol vector departing a fully-digital transmitter follows the form

$$\mathbf{x} = \sqrt{P_{\mathrm{tx}}} \cdot \mathbf{F}\mathbf{s} \tag{23}$$

where P_{tx} reflects the transmit power applied to a symbol vector $\mathbf{s}$ having undergone precoding by a matrix $\mathbf{F}$. The main properties of a `transmitter` include:

- antenna array
- transmit power (i.e., $P_{\mathrm{tx}} \cdot B$)
- precoding matrix (i.e., $\mathbf{F}$)
- precoding power budget
- transmit symbol (i.e., $\mathbf{s}$)
- channel state information
- symbol bandwidth (i.e., B)

A `transmitter`'s properties can be set using its various `set` commands.

A. *Setting the Array*

The antenna array at a `transmitter` object `tx` is set via

```
tx.set_array(a)
```

where `a` is an `array` object.

B. *Setting the Transmit Power*

The transmit power at a `transmitter` object `tx` is set via

```
tx.set_transmit_power(P)
```

where `P` is the transmit power in watts or

```
tx.set_transmit_power(P,'dBm')
```

where `P` is the transmit power in dBm. Note that this transmit power has units of energy per *time*, meaning this is not the transmitted "power" per *symbol*. In other words, this `P` $\neq P_{\mathrm{tx}}$. Rather, P_{tx} is the transmitted *energy* per symbol. Linking transmit power (energy per time) and energy per symbol is the symbol period (or bandwidth).

Let B be the symbol bandwidth and $T = 1/B$ be the symbol period. A transmit power of P watts (joules per second) is related to the transmit energy per symbol P_{tx} via

$$P_{\mathrm{tx}} = P \times T = P \times B^{-1} \tag{24}$$

To set the transmit energy per symbol directly, users can set the transmit power to P_{tx} and the symbol bandwidth to $B = 1$, for example.

C. *Setting the Symbol Bandwidth*

To set the symbol bandwidth at a `transmitter` object `tx`, use

```
tx.set_symbol_bandwidth(B)
```

This will automatically update the symbol period accordingly.

D. *Setting the Number of Streams*

The number of symbol streams transmitted by `tx` is set via

```
tx.set_num_streams(Ns)
```

where `Ns` is the number of streams per transmit symbol vector.

E. *Setting the Transmit Symbol*

The transmit symbol vector $\mathbf{s}$ at a `transmitter` object `tx` is set via

```
tx.set_transmit_symbol(s)
```

where `s` is an $N_{\mathrm{s}} \times 1$ symbol vector.

F. Setting the Transmit Symbol Covariance

The transmit symbol covariance matrix defined as

$$\mathbf{R_s} = \mathbb{E}\left[\mathbf{s}\mathbf{s}^*\right] \tag{25}$$

is automatically set based on the number of transmit streams N_{s} as

$$\mathbf{R_s} = \frac{1}{N_{\mathrm{s}}} \cdot \mathbf{I} \tag{26}$$

The transmit symbol covariance matrix can be set manually using

```
tx.set_transmit_symbol_covariance(Rs)
```

where `Rs` is the covariance matrix of appropriate size ($N_{\mathrm{s}} \times N_{\mathrm{s}}$). It should be known, however, that the transmit symbol covariance is not tied to the actual transmit symbols in any way in MFM. The transmit symbol covariance is merely used for computing quantities such as mutual information. It is up to the user to ensure that the symbols they transmit follow whatever symbol covariance they intend.

G. Setting a Precoding Power Budget

To limit the power associated with precoding, MFM supports a precoding power budget, which takes on the form

$$\left\|\mathbf{F}\right\|_{\mathrm{F}}^2 \leq E \tag{27}$$

where E is the precoding power budget. While it is common to take $E = 1$ or $E = N_{\mathrm{s}}$ along with other normalizations as appropriate to ensure the maximum average energy transmitted per symbol is in fact no more than P_{tx}, MFM supports customizing E as desired. By default, MFM sets the precoding power budget to $E = N_{\mathrm{s}}$ to ensure that the set transmit power holds when $\mathbb{E}\left[\mathbf{s}\mathbf{s}^*\right] = 1/N_{\mathrm{s}} \cdot \mathbf{I}$.

H. Setting the Precoder

Setting the precoder of a transmitter `tx` can be achieved in a few ways in MFM. The most straightforward is to explicitly set the precoding matrix $\mathbf{F}$ using

```
tx.set_precoder(F)
```

where `F` is the $N_{\mathrm{t}} \times N_{\mathrm{s}}$ precoding matrix. If the matrix `F` does not satisfy the allotted precoding power budget, it will be normalized to meet it.

I. Setting Channel State Information at the Transmitter

Channel state information can be provided to the transmitter via

```
tx.set_channel_state_information(csi)
```

where `csi` is a cell of channel state information structs. While there is no specific definition or format for channel state information in MFM, the `link` and `network_mfm` objects will set them in particular way that is described later.

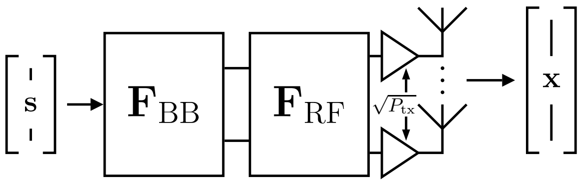

Fig. 11. A hybrid digital/analog transmitter applies a digital precoder $\mathbf{F}_{\mathrm{BB}}$, an analog precoder $\mathbf{F}_{\mathrm{RF}}$, and a transmit gain $\sqrt{P_{\mathrm{tx}}}$ to a symbol vector $\mathbf{s}$ to transmit a vector $\mathbf{x}$.

J. Turning off the Transmitter

A `transmitter` can be "turned off" via

```
tx.turn_off()
```

which sets its precoder to a matrix of zeros.

K. Hybrid Digital/Analog Transmitter

MFM supports hybrid digital/analog precoding via its `transmitter_hybrid` object, which is a subclass of the `transmitter` object, meaning it inherits all of the properties and functions discussed so far. The symbol vector departing a hybrid transmitter follows the form

$$\mathbf{x} = \sqrt{P_{\mathrm{tx}}}\mathbf{F}_{\mathrm{RF}}\mathbf{F}_{\mathrm{BB}}\mathbf{s} \tag{28}$$

where digital precoding followed by analog precoding are applied to symbol vector $\mathbf{s}$ as described by Fig. 11.

A hybrid digital/analog transmitter can be created by including the `'hybrid'` specifier when creating a transmitter.

```
tx = transmitter.create('hybrid')
```

L. Setting the Number of RF Chains

The number of radio frequency (RF) chains present in a hybrid transmitter `tx` can be set using

```
tx.set_num_rf_chains(Lt)
```

where `Lt` is the number of RF chains.

M. Setting the Connected-ness

Fully-connected hybrid precoding architectures are those that connect each RF chain to each antenna element, meaning the *structure* of an analog precoding matrix $\mathbf{F}_{\mathrm{RF}}$ is unconstrained. In partially-connected hybrid architectures—such as sub-array architectures—analog precoding is limited by the physical connections present in an analog beamforming network. Typically, partially-connected architectures offer simplicity and cost-savings as compared to fully-connected architectures. MFM can handle both, partially-connected and fully-connected hybrid architectures, by allowing users to specify the connections that are present in their particular system. This is achieved via

```
tx.set_precoder_hybrid_connections(M)
```

where `M` is an $N_{\mathrm{t}} \times L_{\mathrm{t}}$ matrix whose (i,j)-th entry is a boolean indicating if the j-th RF chain contributes to the i-th antenna. To capture a sub-array architecture, for instance, `M` would take on a block-diagonal form, whereas fully-connected architectures are captured by a matrix `M` of all ones.

If an analog precoding matrix is set that does not comply with the connections present in the hybrid transmitter, it will be forced to by applying the mask `M`.

N. Setting the Resolution of Phase Shifters and Attenuators

It is common for the entries of the analog precoding matrix, which is implemented as a physical network of digitally-controlled phase shifters and possibly attenuators, to be constrained to some phase and amplitude resolutions. MFM accounts for this very important practical constraint by letting the user specify the number of bits used at each phase shifter and each attenuator via

```
tx.set_precoder_analog_phase_resolution
 ..._bits(bits_phase)
tx.set_precoder_analog_amplitude_resolution
 ..._bits(bits_amplitude)
```

MFM assumes that the phase shifter resolution is uniformly spread over 2π radians. For amplitude resolution, MFM supports both linear and logarthmic uniform amplitude control to account for log-stepped digitally controlled attenuators.

To eliminate the presence of amplitude control, users can set `bits_amplitude = 0`. To remove the resolution constraints associated with digitally-controlled phase shifters and/or attenuators, users can set `bits_phase = Inf` and `bits_amplitude = Inf`, respectively.

O. Setting a Digital Precoding Power Budget

Like the fully-digital precoder $\mathbf{F}$, MFM supports a power budget placed on the digital precoder $\mathbf{F}_{\mathrm{BB}}$ of the form

$$\|\mathbf{F}_{\mathrm{BB}}\|_{\mathrm{F}}^2 \leq E \tag{29}$$

where E is the maximum power the digital precoder can exhibit. To set this, use

```
tx.set_precoder_digital_power_budget(E)
```

P. Setting the Digital and Analog Precoders

There are multiple ways to set the digital and analog precoders of a hybrid transmitter. The most straightforward way is to explicitly set them using

```
tx.set_precoder_digital(F_BB)
tx.set_precoder_analog(F_RF)
```

where `F_BB` is the $L_{\mathrm{t}} \times N_{\mathrm{s}}$ digital precoding matrix $\mathbf{F}_{\mathrm{BB}}$ and `F_RF` is the $N_{\mathrm{t}} \times L_{\mathrm{t}}$ analog precoding matrix $\mathbf{F}_{\mathrm{RF}}$. When setting each, their respective constraints (e.g., digital precoding power budget, phase shifter and attenuator resolution, connected-ness) will be enforced. With the digital and analog precoders set, the effective precoder of the transmitter is then

$$\mathbf{F} = \mathbf{F}_{\mathrm{RF}}\mathbf{F}_{\mathrm{BB}} \tag{30}$$

which will be reflected in the hybrid transmitter's inherited property `tx.precoder`.

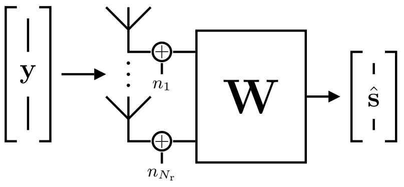

Fig. 12. A fully-digital receiver.

Q. Another Way to Set the Precoder(s)

Methods to explicitly set the precoding matrices have been discussed. However, this is not always a very attractive approach, especially since it can severely undermine the advantages of MFM's object-oriented design. This motivates setting a transmitter's precoder(s) via

```
tx.configure_transmitter(strategy)
```

where `strategy` is a string specifying the strategy/method to use when designing the transmitter's precoder(s). Based on `strategy`, the transmitter will refer to its `configure_transmitter_<strategy>()` functions. For example, for eigen-based precoding,

```
tx.configure_transmitter('eigen')
```

will refer to

```
tx.configure_transmitter_eigen()
```

which will use the transmitter's channel state information to design its precoder.

This string-based way to specify a transmit strategy is particularly useful since it keeps main simulation scripts free of the linear algebra involved in precoder design, allows users to easily switch between strategies, and makes setting precoders network-wide much more manageable. Users can add custom transmit strategies with a few simple steps. Please refer to the MFM website for more information on doing so.

VII. Receiver Objects

A receiver in MFM is captured by the `receiver` object and its subclasses. A `receiver` can be created via

```
rx = receiver.create()
```

Like the transmitter, a `receiver` object employs fully-digital precoding by default, though hybrid digital/analog receivers are supported. The estimated symbol vector output by a fully-digital receiver follows the form

$$\hat{\mathbf{s}} = \mathbf{W}^*(\mathbf{y} + \mathbf{n}) \tag{31}$$

where a combining matrix $\mathbf{W}$ is applied to the signal vector $\mathbf{y}$ impinging the receive array plus noise $\mathbf{n}$. The main properties of a `receiver` include:

- antenna array
- combining matrix (i.e., $\mathbf{W}$)
- receive symbol (i.e., $\hat{\mathbf{s}}$)
- noise power spectral density (i.e., N_0 or σ_{n}^2)
- channel state information
- symbol bandwidth (i.e., B)

Like the `transmitter`, a `receiver` object's properties can be set using its various `set` commands.

A. Setting the Array, Symbol Bandwidth, and Number of Streams

The antenna array, symbol bandwidth, and number of streams can all be set at a `receiver` object `rx` in the same fashion as for the `transmitter` object.

```
rx.set_array(a)
rx.set_symbol_bandwidth(B)
rx.set_num_streams(Ns)
```

B. Setting the Noise Level

MFM models noise as being additive, i.i.d. Gaussian across receive antennas as shown in Fig. 12. The noise vector $\mathbf{n}$ is drawn from the complex Gaussian distribution as

$$\mathbf{n} \sim \mathcal{N}_{\mathbb{C}}\left(\mathbf{0}, \sigma_{\mathrm{n}}^2 \cdot \mathbf{I}\right) \tag{32}$$

where σ_{n}^2 is the average noise energy per symbol (joules). The noise energy per symbol is equal to the noise power per Hertz of bandwidth given our symbol period is the inverse of our symbol bandwidth. To set the noise energy per symbol, then we can use

```
rx.set_noise_power_per_Hz(psd)
```

where `psd` is the noise power spectral density (in joules per Hz) or the more convenient

```
rx.set_noise_power_per_Hz(psd,'dBm_Hz')
```

where `psd` is the noise power spectral density is in dBm/Hz (e.g., `psd = -174`). The effective noise power is then the product of noise power spectral density and symbol bandwidth as $\sigma_{\mathrm{n}}^2 \cdot B$, which is not used to generate noise but is convenient for interpreting link budgets and making the connection from theory to practice.

C. Setting/Realizing Noise

The noise vector $\mathbf{n}$ can be realized using

```
rx.set_noise()
```

which will update the property `rx.noise` with the realized noise vector. The noise vector can be set manually using

```
rx.set_noise(n)
```

where `n` is an $N_{\mathrm{r}} \times 1$ noise vector.

D. Setting the Received Signal Vector

To set the $N_{\mathrm{r}} \times 1$ received signal vector $\mathbf{y}$ that a combiner observes at its antennas, use

```
rx.set_received_signal(y)
```

E. Setting the Combiner

A receiver's combiner can be set via

```
rx.set_combiner(W)
```

where `W` is an $N_{\mathrm{r}} \times N_{\mathrm{s}}$ combining matrix.

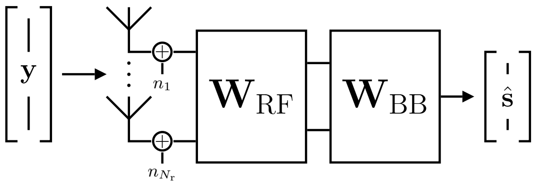

Fig. 13. A hybrid digital/analog receiver.

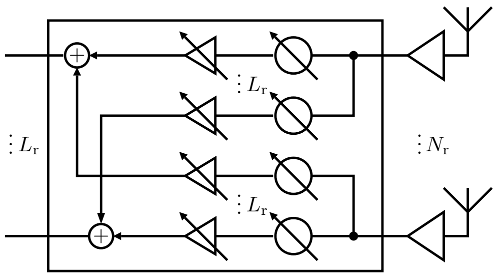

Fig. 14. A fully-connected analog combining network, where each antenna feeds each RF chain via a phase shifter and attenuator.

F. Getting the Receive Symbol

The $N_{\mathrm{s}} \times 1$ receive symbol $\hat{\mathbf{s}}$ can be retrieved via

```
s = rx.get_receive_symbol()
```

which will return the estimated symbol vector based on the current received signal, combiner, and noise.

G. Hybrid Digital/Analog Receiver

Like with transmission, MFM supports hybrid digital/analog receivers via its `receiver_hybrid` object, which is a subclass of the `receiver` object, meaning it inherits all of the `receiver` properties and functions discussed so far. The receive symbol output by a hybrid receiver takes the form

$$\hat{\mathbf{s}} = \mathbf{W}_{\mathrm{BB}}^{*} \mathbf{W}_{\mathrm{RF}}^{*} (\mathbf{y} + \mathbf{n}) \tag{33}$$

where an analog combining matrix followed by a digital combining matrix are applied to a received signal vector $\mathbf{y}$ plus noise $\mathbf{n}$.

A hybrid receiver can be created by including the `'hybrid'` specifier when creating a receiver.

```
rx = receiver.create('hybrid')
```

H. Setting Hybrid Receiver-Specific Parameters

The settings pertinent to a hybrid digital/analog receiver can be set with the same functions of a hybrid digital/analog transmitter. The number of RF chains, connected-ness of the hybrid receiver, and constraints of analog combining can all be configured in the same fashion as with the hybrid transmitter. Please refer to the hybrid transmitter's documentation and the MFM website for detailed information.

VIII. Device Object

The `device` object, as its name suggests, represents a wireless communications terminal such as a user equipment (UE), base station, and the like. A `device` object having only transmit *or* receive capability will contain a `transmitter` *or* `receiver`, respectively. A `device` object that has both transmit *and* receive capability—i.e., a transceiver—will be comprised of both a `transmitter` and `receiver`.

An empty `device` can be created via

```
d = device.create(type)
```

where `type` is either `'transmitter'`, `'receiver'`, or `'transceiver'` (default).

A `device` object's `transmitter` and/or `receiver` can either be digital or hybrid digital/analog. A `device` object with fully-digital transmit and receive capability can be created via

```
d = device.create('transceiver','digital')
```

whereas a `device` with a hybrid digital/analog transmitter and receiver can be created via

```
d = device.create('transceiver','hybrid')
```

A. Setting a Device's Location

A `device` object `d` can be placed in 3-D space by setting its coordinate via

```
d.set_coordinate(x,y,z)
```

where `x`, `y`, and `z` are Cartesian coordinates in meters. The location of the `device` will be essential for geometry-dependent path loss and channel models in addition to visualization.

B. Setting the Transmitter and/or Receiver

The transmitter and/or receiver of a `device` object `d` can be set manually if desired via

```
d.set_transmitter(tx)
d.set_receiver(rx)
```

C. Setting the Transmit and Receive Arrays

The arrays at a `device` object `d`'s transmitter and receiver can be set using

```
d.set_arrays(array_transmit,array_receive)
```

which will set the arrays respectively.

D. Setting the Number of Streams

The number of streams at a `device` object `d`'s transmitter and receiver can be set using

```
d.set_num_streams(Ns)
```

which will set the same number of streams at the transmitter and receiver.

E. Setting the Number of RF Chains

The number of RF chains in a `device` object `d`'s hybrid transmitter and hybrid receiver can be set using

```
d.set_num_rf_chains(Lt,Lr)
```

which will set them respectively.

F. Setting the Symbol Bandwidth

The symbol bandwidth at a `device` object `d` can be set using

```
d.set_symbol_bandwidth(B)
```

which will set the symbol bandwidth at the device's transmitter and receiver to `B`, also.

G. Interfacing with a Device's Transmitter and/or Receiver

In many ways, the `device` object acts as a proxy for configuring and interfacing with its transmitter and/or receiver. Thus, the `device` is supplied with a number of *passthrough* functions that make directly interfacing with its transmitter and/or receiver simpler. Some of the passthrough functions that exist for the transmitter:

- `d.set_transmit_symbol_bandwidth(B)`
- `d.set_transmit_num_streams(Ns)`
- `d.set_transmit_symbol(s)`
- `d.set_transmit_array(a)`
- `d.set_transmit_num_rf_chains(Lt)`
- `d.set_transmit_power(P,unit)`
- `d.set_transmit_symbol_covariance(Rs)`

And for the receiver:

- `d.set_receive_symbol_bandwidth(B)`
- `d.set_receive_num_streams(Ns)`
- `d.set_received_signal(y)`
- `d.set_receive_array(a)`
- `d.set_receive_num_rf_chains(Lr)`
- `d.set_noise(n)`
- `d.set_noise_power_per_Hz(psd,unit)`

H. Setting the Source and Destination

Declaring which other `device` a given `device d` should transmit to (i.e., the destination device) or receive from (i.e., source device) can be accomplished via

```
d.set_destination(dev_1)
d.set_source(dev_2)
```

where `dev_1` is a `device` with receive capability and `dev_2` is a `device` with transmit capability. This source-destination concept is only pertinent to particular use-cases of MFM, particularly at its `link`-level and `network_mfm`-level, which will be discussed shortly, though also can be used in scenarios outside of such.

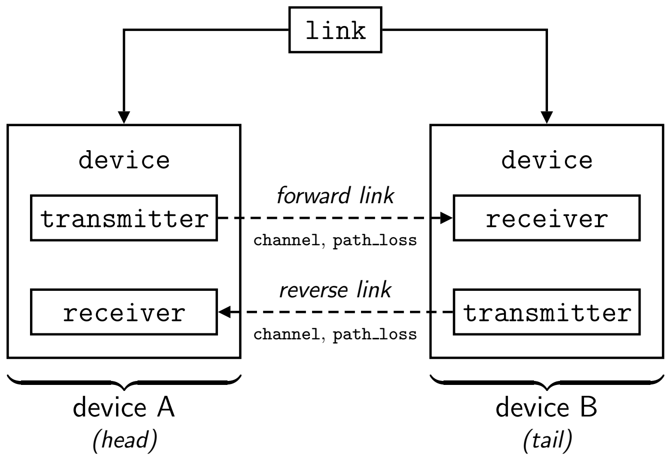

Fig. 15. A link established between two transceivers.

IX. Link Object

Between any two devices sharing the same radio resources exists a channel matrix and path loss connecting them. MFM employs exactly that in its `link` object used to connect a pair of `device` objects, with the caveat that one of the devices must have transmit capability and the other have receive capability; otherwise the physical connection (or *link*) between the two devices would be immaterial.

Examining our familiar MIMO formulation, we can see that MFM uses a `link` to capture the channel matrix $\mathbf{H}$ and large-scale gain G due to path loss.

$$\hat{\mathbf{s}} = \mathbf{W}^* \left(\sqrt{P_{\mathrm{tx}}} \cdot \underbrace{G \cdot \mathbf{H}}_{\texttt{link}} \mathbf{F}\mathbf{s} + \mathbf{n} \right) \tag{34}$$

Suppose there exist two devices `d1` and `d2`, where `d1` has transmit capability and `d2` has receive capability. Note that one or both devices could be transceivers. A `link` connecting these two devices is created via

```
lnk = link.create(d1,d2);
```

By convention, the first device (`d1` in this case) is called the *head* while the second device (`d2`) is called the *tail*. The head always has transmit capability and the tail always has receive capability.

Since both `d1` and `d2` could be transceivers, a link may exist between the two `device` objects in both directions, as illustrated by Fig. 15. We refer to the link from the head to the tail as the *forward link* and from the tail to the head as the *reverse link*. To handle cases when the forward and reverse links are *symmetric* (or reciprocal), a single `link` object contains both the forward and reverse links. A `link` object, therefore, has two `channel` objects (a forward channel and reverse channel) and two `path_loss` objects (forward path loss and reverse path loss).

A. Setting the Forward and Reverse Channel Models

The channel models used on the forward and reverse links of a `link` object `lnk` can be set respectively using

```
lnk.set_channel(chan_fwd,chan_rev)
```

where `chan_fwd` and `chan_rev` are `channel` objects. To use the same channel model on the forward and reverse links (but independent instances of such), use

```
lnk.set_channel(chan)
```

Note that these `channel` objects are deep copied when used on a particular `link` meaning the same `channel` objects used when configuring multiple `link` objects if desired.

B. Setting the Forward and Reverse Path Loss Models

The path loss models used on the forward and reverse links of a `link` object `lnk` can be set respectively using

```
lnk.set_path_loss(path_fwd,path_rev)
```

where `path_fwd` and `path_rev` are `path_loss` objects. To use the same path loss model on the forward and reverse links (but independent instances of such), use

```
lnk.set_path_loss(path)
```

Note that these `channel` objects are deep copied when used on a particular `link` meaning the same `channel` objects used when configuring multiple `link` objects if desired.

C. Setting the Forward and Reverse SNR Manually

Following MIMO literature, the so-called *large-scale* SNR for a given transmit-receive pair is defined as

$$\mathrm{SNR} = \frac{P_{\mathrm{tx}} \cdot G^2}{\sigma_{\mathrm{n}}^2} \tag{35}$$

The SNR of the forward and reverse links—which need not be equal—are computed by the `link` object based on the realized large-scale gains (i.e., G), transmit power, and noise level and are stored in the properties

```
lnk.snr_forward
lnk.snr_reverse
```

where we have

$$\mathrm{SNR}_{\mathrm{fwd}} = \frac{P_{\mathrm{tx,head}} \cdot G_{\mathrm{fwd}}^2}{\sigma_{\mathrm{n,tail}}^2} \tag{36}$$

$$\mathrm{SNR}_{\mathrm{rev}} = \frac{P_{\mathrm{tx,tail}} \cdot G_{\mathrm{rev}}^2}{\sigma_{\mathrm{n,head}}^2} \tag{37}$$

Often times, researchers are interested in examining performance as a function of SNR. Since MFM uses path loss to determine the large-scale gain G, it is impractical to expect researchers to tailor either the path loss model or their devices location to achieve a desired SNR. Therefore, the `link` object supplies users with the functions

```
lnk.set_snr(snr_fwd,snr_rev,unit)
```

to automatically adjust the large-scale gain G on each link to achieve an SNR on the forward link of `snr_fwd` and on the reverse link of `snr_rev`. The `unit` argument can be neglected when `snr_fwd` and `snr_rev` are in linear units or can be `'dB'` when they are in units of dB.

D. Setting Forward and Reverse Channel Symmetry

In some cases, it may be desired that the forward and reverse channels are symmetric where the reverse channel matrix is the conjugate transpose of the forward channel matrix as

$$\mathbf{H}_{\mathrm{rev}} = \mathbf{H}_{\mathrm{fwd}}^{*} \tag{38}$$

Currently, MFM only supports this case when the number of transmit antennas and receive antennas are equal at a given transceiver; this captures most practical cases since channel symmetry is more likely to hold when the transmit array and receive array are in reality the same array (as opposed to separate transmit and receive arrays that therefore may see different channels).

To set the forward and reverse channels to be symmetric, simply use

```
lnk.set_channel_symmetric(true)
```

which will use the conjugate transpose of realizations from the forward channel model to set the reverse channel matrix.

E. Setting Forward and Reverse Path Loss Symmetry

In some cases, it may be desired that the forward and reverse path losses (and therefore large-scale gains G_{fwd} and G_{rev}) be symmetric where

$$G_{\mathrm{rev}} = G_{\mathrm{fwd}} \tag{39}$$

To enforce this in MFM, use

```
lnk.set_path_loss_symmetric(true)
```

Note that when the same deterministic path loss model is used on the forward and reverse links, the forward and reverse path losses (and thus large-scale gains) will always inherently be symmetric (e.g., since the separation between the head and tail devices is equal in both directions). Enforcing path loss symmetry, therefore, is particularly useful with stochastic path loss models.

F. Invoking a Link Realization

To invoke a realization of the channel and path loss models of a `link` object `lnk`, simply use

```
lnk.realization()
```

To invoke a realization of the channel and path loss models separately, use

```
lnk.realization_channel()
lnk.realization_path_loss()
```

G. Getting the Realized Channel Matrix and Large-Scale Gain

Following the realization of a `link` object `lnk`, the resulting channel matrix $\mathbf{H}$ and large-scale gain G of the forward and reverse links can be retrieved via

```
H_fwd = lnk.channel_matrix_forward()
H_rev = lnk.channel_matrix_reverse()
G_fwd = lnk.large_scale_gain_forward()
G_rev = lnk.large_scale_gain_reverse()
```

H. Computing the Received Signal

Once a `link` object has been realized (and the devices have been configured), it is capable of automatically computing the received signal on the forward and reverse links via

```
lnk.compute_received_signal()
```

which will populate the received signal vector $\mathbf{y}$ at the tail and head devices (if applicable).

I. Computing the Link Budget

Once a `link` has been realized, users can call

```
[fwd,rev] = lnk.compute_link_budget()
```

to retrieve two structs containing link budget values (in log-scale for convenience) for the forward link and reverse link (if applicable).

J. Computing Channel State Information

Users can fetch channel state information of a realized link using

```
[fwd,rev] = lnk.compute_channel_state...
            _information()
```

which returns two structs containing channel state information for the forward link and reverse link (if applicable).

K. Computing Signal Covariance

Users can fetch the covariance matrices of the received desired term and of received noise on the forward and reverse links (if applicable) using

```
[Ry_fwd,Rn_rev] = lnk.compute_covariance...
                  _forward()
[Ry_rev,Rn_rev] = lnk.compute_covariance...
                  _reverse()
```

where the first return argument of each is the covariance matrix of the form

$$\mathbb{E}\left[\mathbf{W}^{*}\mathbf{y}\mathbf{y}^{*}\mathbf{W}\right] \tag{40}$$

and the second return argument of each is the covariance matrix of the form

$$\mathbb{E}\left[\mathbf{W}^{*}\mathbf{n}\mathbf{n}^{*}\mathbf{W}\right] \tag{41}$$

Both of these expectations are computed based on the current precoder-combiner configurations and the covariance matrix settings at the transmitter and receiver.

L. Reporting the Mutual Information

MFM can automatically compute and report the mutual information (under Gaussian signaling) based on the current link state using

```
mi_fwd = lnk.report_mutual_information...
         _forward()
mi_rev = lnk.report_mutual_information...
         _reverse()
```

where the returned value of each is the mutual information in bits/sec/Hz.

M. Reporting the Symbol Estimation Error

MFM can automatically compute and report the symbol estimation error based on the current link state using

```
[ef,nf] = lnk.report_symbol...
          _estimation_error_forward()
[er,nr] = lnk.report_symbol...
          _estimation_error_reverse()
```

where the first return value of each is the absolute symbol estimation error

$$\|\hat{\mathbf{s}} - \mathbf{s}\|_2^2 \tag{42}$$

and the second return value is the symbol estimation error normalized to the transmit symbol energy defined as

$$\frac{\|\hat{\mathbf{s}} - \mathbf{s}\|_2^2}{\|\mathbf{s}\|_2^2} \tag{43}$$

X. Network Object

At the highest level of MFM's object-oriented structure is the `network_mfm` object, which houses `device` objects and the `link` objects connecting them. Recall that the `link` object represents a *physical* connection between two devices rather than a *communication* link. A `network_mfm` object can be created simply via

```
net = network_mfm.create()
```

Currently, the `network_mfm` object in MFM captures scenarios where all devices present in the network share the same radio resources (i.e., same time and frequency resources), meaning some degree of interference will be inflicted onto each receiver in the network by each transmitter (except for those that are turned off).

A. Adding Devices to the Network

Adding a particular device `dev` to the network can be achieved via

```
net.add_device(dev)
```

B. Adding Source-Destination Pairs to the Network

Suppose we have two `device` objects `dtx` and `drx`, where `dtx` is a transmitting device and `drx` is a receiving device, both of which have already been set up as necessary. To inform the network that `dtx` should transmit to `drx` and that `drx` should receive from `dtx`, the following command is used

```
net.add_source_destination(dtx,drx);
```

which adds `dtx` and `drx` as a *source-destination pair*, `dtx` being the source and `drx` being the destination. There are multiple ways to add devices to a network; this is merely the most common.

C. Linking Devices in the Network

To impose a physical connection (i.e., channel and path loss) between devices, links within the network can be manually added via

```
net.add_link(lnk)
```

where `lnk` is a `link` object. With many devices—and therefore likely many links—this can be cumbersome. Fortunately, MFM comes with a more convenient way of automatically populating links between pairs of devices via

```
net.populate_links_from_source_destination()
```

which will populate all links from each source to each destination. Recall that since all devices in an MFM network share the same radio resources, each transmitting device will impose interference on those it does not intend to transmit to.

D. Removing Source-Destination Pairs from the Network

To remove a specific source-destination pair from the network, use

```
net.remove_source_destination_pair(s,d)
```

where `s` and `d` are `device` objects to remove. Note that the *pair* of devices is removed from the network, not the individual devices, meaning if either `s` or `d` exists in another source-destination pair, it will remain in the network.

To remove all source-destination pairs from the network, use

```
net.remove_all_source_destination()
```

which does not remove any links from the network.

E. Removing a Device from the Network

Removing a particular device `dev` from the network can be achieved via

```
net.remove_device(dev)
```

which also removes any links and source-destination pairs associated with `dev`.

F. Setting the Channel and Path Loss Models Network-Wide

To specify the channel and path loss models used on all links in the network, we can invoke

```
net.set_channel(c)
net.set_path_loss(p)
```

where `c` and `p` are `channel` and `path_loss` objects, respectively.

G. Other Network-Wide Settings

MFM offers the user the convenience of setting various system parameters network-wide instead of setting them at each device one-by-one.

```
net.set_symbol_bandwidth(B);
net.set_propagation_velocity(prop_vel);
net.set_carrier_frequency(fc);
net.set_num_streams(num_streams);
net.set_transmit_power(P,'dBm');
net.set_transmit_symbol(s);
net.set_noise_power_per_Hz(psd,'dBm_Hz');
```

H. Viewing the Network

To view a network `net` in 2-D or 3-D, use

```
net.show_2d()
net.show_3d()
```

I. Invoking a Network Realization

Invoking a realization of an entire network `net` is achieved with a single line

```
net.realization()
```

which realizes all channels and path loss models in the network.

J. Channel State Information

To collect and distribute channel state information across a network `net`, use

```
net.compute_channel_state_information()
net.supply_channel_state_information()
```

K. Configuring Transmitters and Receivers Network-Wide

To configure the transmitters and receivers across the network using string-specified transmit and receive strategies, use, for example,

```
net.configure_transmitter('eigen')
net.configure_receiver('mmse')
```

which transmits using eigen-beamforming and receives in an minimum mean square error (MMSE) fashion.

L. Computing the Receiving Signals Network-Wide

With a network realized and its devices configured, the received signals can be computed via

```
net.compute_received_signals()
```

which will automatically incorporate interference caused by other transmitting devices in the network.

M. Reporting the Mutual Information

To report the mutual information (under Gaussian signaling) achieved between a particular pair of devices `dev_1` and `dev_2`, use

```
mi = net.report_mutual...
     _information(dev_1,dev_2)
```

which will automatically incorporate interference caused by other transmitting devices in the network.

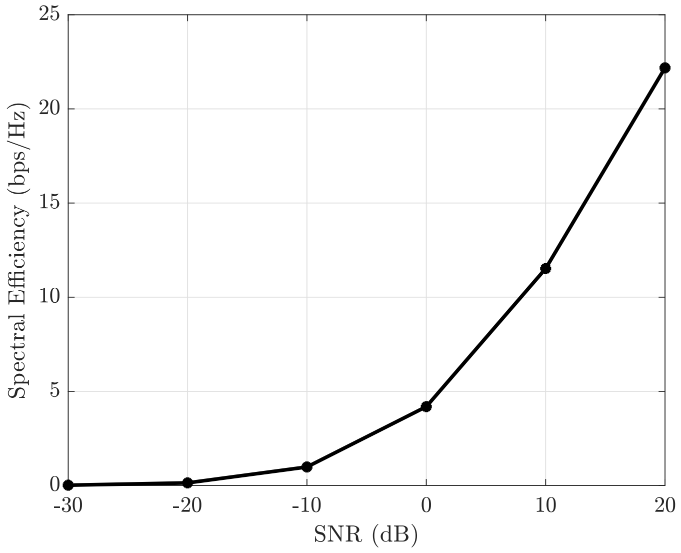

Fig. 16. The spectral efficiency of a point-to-point Rayleigh-faded network as a function of SNR simulated using MFM.

N. Reporting the Symbol Estimation Error

To report the symbol estimation error achieved between a particular pair of devices `dev_1` and `dev_2`, use

```
[err,nerr] = net.report_symbol...
             _estimation_error(dev_1,dev_2)
```

where the first return value `err` is the absolute symbol estimation error

$$\|\hat{\mathbf{s}} - \mathbf{s}\|_2^2 \tag{44}$$

and the second return value `nerr` is the symbol estimation error normalized to the transmit symbol energy defined as

$$\frac{\|\hat{\mathbf{s}} - \mathbf{s}\|_2^2}{\|\mathbf{s}\|_2^2} \tag{45}$$

XI. Conclusion

We have presented MFM, a MATLAB framework that can facilitate accuracy and reproducibility in wireless research. MFM supplies users with widely used channel and path loss models out-of-the-box and supports fully-digital and hybrid digital/analog beamforming. MFM can be used flexibly in a variety of ways: from low-level uses like antenna array design and drawing channel and path loss realizations to higher-level system simulation. For complete documentation, examples, and guides on using MFM, please visit `https://mimoformatlab.com`.