MIMO FOR MATLAB: A Toolbox for Simulating MIMO Communication Systems

Ian P. Roberts
https://mimoformatlab.com

***Abstract*—We present MIMO FOR MATLAB (MFM), a toolbox 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 sub-6 GHz, massive MIMO, millimeter wave, and terahertz communication. Out-of-the-box, MFM supplies users with widely used channel and path loss models from academic literature and wireless standards; if a particular channel or path loss model is not provided by MFM, users can create custom models by following a few simple rules. The complexity and overhead associated with simulating networks of multiple devices can be significantly reduced 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, aggregate 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.**

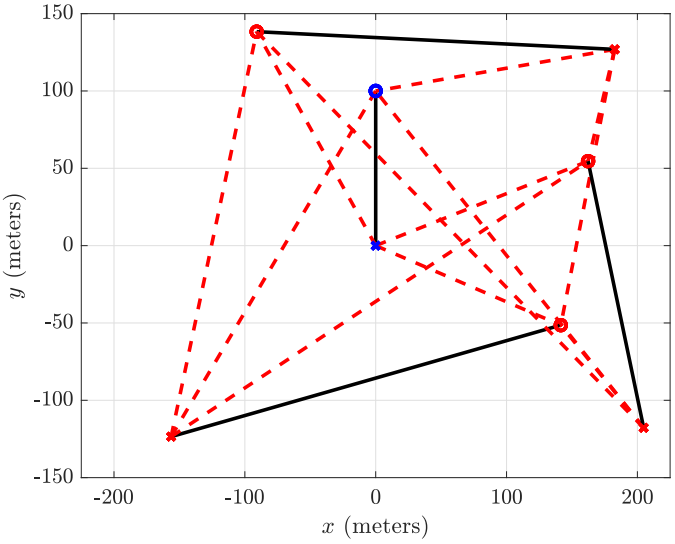

Fig. 1. A network of eight devices scattered in space simulated in MFM. Four transmit-receive pairs (shown as ×'s and ○'s, respectively) use the same time-frequency resource in their attempt to individually communicate.

I. OVERVIEW

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

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

sometimes termed *symbol-level* or *single-letter* formulations [1]. To communicate a symbol vector $\mathbf{s}$ over some channel matrix $\mathbf{H}$, a transmitter applies power P and a precoding matrix $\mathbf{F}$ while a receiver applies a combining matrix $\mathbf{W}$ to recover an estimate $\hat{\mathbf{s}}$ of the symbol vector $\mathbf{s}$. Along the way, path loss $1/G^2$ weakens the transmitted signal and additive noise $\mathbf{n}$ 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 a network grows to even a moderate size. The variety of channel models and precoding/combining strategies used in MIMO communication, along with enforcing common normalizations, can further complicate simulation and introduce the potential for mistakes and inconsistency.

This has motivated us to create MIMO FOR MATLAB (MFM), a toolbox for simulating MIMO communication systems [2]. MFM is written in an object-oriented fashion and comes with a collection physical layer tools including a variety of channel models, path loss models, transmitters, receivers, and antenna arrays that can be used for sub-6 GHz, millimeter wave (mmWave), terahertz (THz), and beyond. It supports both fully-digital and hybrid digital/analog transceivers with options for limited phase and amplitude control as well as fully- and partially-connected analog beamforming networks. In addition to its applications in research, MFM can act as an educational tool to help students understand, experiment, and visualize MIMO communication.

MFM is a physical layer toolbox that 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 or to draw channel realizations from a particular model. At its highest level, MFM can be used to simulate a network of many users, automatically aggregating interference inflicted at each device by one another. In between, simple point-to-point communication can be simulated, allowing users to develop, implement, and evaluate novel precoding and combining schemes.

MFM is freely available for use under the MIT license and can be setup in MATLAB in minutes. It has been completely documented in-line, which can be accessed using MATLAB's `help` function. In addition, the MFM website has extensive

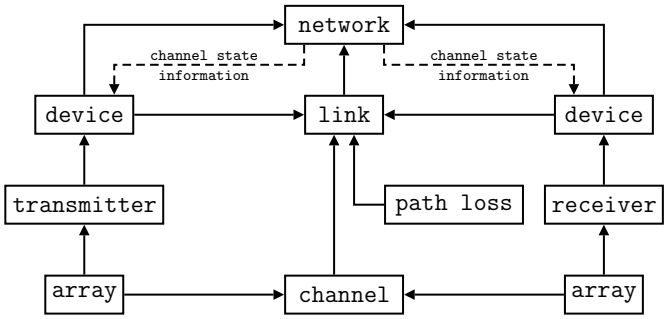

Fig. 2. Example architecture of an MFM script simulating point-to-point communication between a transmitting device and receiving device.

amounts of documentation, examples, and tutorials, which will continue to be updated. Along with the official MFM contents, users are encouraged to develop and share their own channel models, path loss models, etc. to extend MFM's capabilities and We hope MFM will act as a platform that facilitates consistency and reproducibility of MIMO research and accelerates its development. 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.

MFM executes physical layer signal processing at the symbol level, abstracting out the waveforms that carry those symbols as is commonly done in MIMO literature. The object-oriented structure of MFM can be summarized as in Fig. 2, though select object(s) can be useful on their own. At the lowest level of MFM are antenna arrays, channel models, and path loss models. Transmitters and receivers leverage antenna arrays to execute precoding and combining, respectively. Devices, which can have transmit and/or capability, are connected to one another via links, which capture propagation via channel and path loss models. The collection of devices and the links connecting them comprise a network. A network instance in MFM captures devices operating on the same time-frequency resource, meaning all transmitting devices in a network can inflict interference onto all receiving devices. As such, it is up to users of MFM to properly choose devices present in the network to capture time- or frequency-division.

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.

As mentioned, MFM can be used to varying degrees. For beamforming and array-related work, users may only need MFM's antenna array object. Those interested in using MFM to generate channel realizations can use the antenna array and channel objects. To use MFM to simulate point-to-point MIMO communication—perhaps to experiment with precoding/combining schemes—can use MFM at the link level. To capture the impacts of interference, users can use it at the network level. We imagine MFM could be used for a variety of applications beyond strictly MIMO research including stochastic geometry, joint communication and radar, underwater acoustic communication, sonar, machine learning in communications, and satellite communication. While MFM is currently strictly a physical layer toolbox, other areas of research, such as on scheduling, could leverage MFM to avoid the headache associated with implementing physical layer communication network-wide.

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 be easily shared and implemented across the research community. 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 additions to MFM are widely used, there are avenues for them to be incorporated into future versions of MFM.

II. Low Level Objects and Usage

At MFM's lowest level are its antenna arrays, channel models, and path loss models.

A. Antenna Arrays

From which the term MIMO takes its name, let us begin by outlining support for antenna arrays [3]. MFM's `array` object is used to represent antenna arrays, which can be constructed as uniform linear arrays (ULAs), uniform planar arrays (UPAs), or other arbitrary array the user wishes. Arrays can be constructed in a few ways:

- `a = array.create()` creates an empty array with no elements, after which elements can be added to the array.
- `a = array.create(N)` creates a half-wavelength ULA with `N` elements.
- `a = array.create(M,N)` creates a half-wavelength UPA with `M` rows of `N` elements.

Arrays can be rotated and translated as desired and individual array elements can be added or removed. To simulate a

typical MIMO communication system, creating an `array` using the ULA or UPA method will often suffice without significant modification, especially when the channel model used is independent of the array geometry (e.g., a Rayleigh-faded channel). Currently, MFM assumes isotropic elements, though support for more practical element patterns will likely be included in future versions.

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. In general, array elements can be placed arbitrarily in 3-D in units of carrier wavelengths λ, which makes the array behavior agnostic of the carrier frequency at which it operates. To handle arbitrary array construction, MFM computes the array response based on the relative positioning of the array elements.

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

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

where λ is the carrier wavelength and

$$\zeta(x,y,z,\theta,\phi) = x\sin\theta\cos\phi + y\cos\theta\cos\phi + z\sin\phi \quad (3)$$

The array response vector is constructed by collecting the relative phase shift seen by each of the array's N elements as

$$\mathbf{a}(\theta,\phi) = [a_1(\theta,\phi), a_2(\theta,\phi), \ldots, a_N(\theta,\phi)]^{\mathrm{T}} \quad (4)$$

To obtain the array response of an array `a` in a particular azimuth `theta` and elevation `phi` in MFM, one simply needs to call

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

MFM will automatically populate the array response vector `v` based on the array geometry.

To weight the N elements of an antenna array `a`, one can use `a.set_weights(w)`, where `w` is a vector of N complex weights. 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

$$g(\theta,\phi) = \mathbf{w}^{\mathrm{T}}\mathbf{a}(\theta,\phi) \quad (5)$$

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)
```

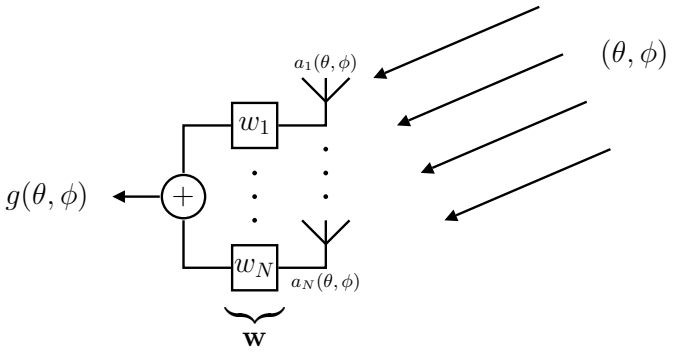

Fig. 3. The gain of a weighted array in a direction (θ, ϕ).

```
a.set_weights(w)
```

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

```
g = 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.

B. Channel Models

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. MFM supports a variety of channel models including the Rayleigh-faded channel, line-of-sight (LOS) channel, ray/cluster channel (extended Saleh-Valenzuela model [4]), and spherical-wave channel [5].

To create a Rayleigh-faded channel object `c`, for instance, one can simply use

```
c = channel.create('Rayleigh')
```

Other channels can be created similarly. Setup of a channel `c` begins by declaring the propagation velocity (e.g., 3×10^8) and carrier frequency using

```
c.set_propagation_velocity(vel)
c.set_carrier_frequency(fc)
```

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. Note that setting the carrier frequency will automatically update the channel's carrier wavelength according to its propagation velocity.

The next necessary step is informing the channel of the transmit and receive arrays between which it lives, which can be accomplished by

```
c.set_arrays(atx,arx)
```

where `atx` and `arx` are the transmit and receive array objects, respectively. Informing `channel` objects of the transmit and receive arrays dictates the size of the to-be-realized channel matrix $\mathbf{H}$ and also provides geometric channel models—such as the LOS channel and ray/cluster channel—with access to the array responses. Each channel model will have unique setup steps before it can becomes useful.

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. Users can create custom channel models that are compatible with MFM by following a few simple rules; more information can be found on the MFM website.

C. Path Loss Models

While channel models capture the small-scale mixing between transmit and receive antennas, path loss models capture the large-scale gain G between a transmitter and receiver (e.g., due to propagation loss, shadowing, blockage, etc.). Path loss models can be used on their own to realize values of G directly or in the point-to-point and network settings. A number of path loss models exist in MFM including free-space path loss (with and without log-normal shadowing) and two-slope path loss. More path loss models are continuing to be added to MFM. Creating a free-space path loss object `p` can be achieved via

```
p = path_loss.create('FSPL')
```

After setting its propagation velocity and carrier frequency, the path loss exponent `ple` can be set as

```
p.set_path_loss_exponent(ple)
```

The distance of the path `d` can be set using

```
p.set_distance(d)
```

From there, the path loss can be realized using

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

which will return a power loss `L` according to the free-space formula

$$L^{-1} = G^2 = \left(\frac{\lambda}{4\pi}\right)^2 \times \left(\frac{1}{d}\right)^{\eta} \tag{6}$$

where η is the path loss exponent and d is the distance of the path. More complicated path loss models may require additional setup, and those involving stochastics (e.g., with shadowing) may realize a random path loss on each realization.

III. Link-Level Usage

A. Transmitters

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, but MFM also supports hybrid digital/analog precoding. The symbol vector departing a fully-digital transmitter follows the form

$$\mathbf{x} = \sqrt{P} \cdot \mathbf{F}\mathbf{s} \tag{7}$$

where P 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 its antenna array, transmit power, precoding matrix (i.e., $\mathbf{F}$), precoding power budget, transmit symbol (i.e., $\mathbf{s}$), channel state information, and symbol bandwidth (i.e., B). A `transmitter`'s properties can be set using various `set` commands. For example, to set the transmit power of a transmitter `tx`, one can simply use

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

where `P` is the transmit power in dBm.

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

$$\|\mathbf{F}\|_{\mathrm{F}}^2 \leq E \tag{8}$$

where E is the precoding power budget. By default, MFM sets the precoding power budget to $E = N_{\mathrm{s}}$.

The precoding matrix can be set using

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

where `F` is an $N_{\mathrm{t}} \times N_{\mathrm{s}}$ precoding matrix, or using other methods as we will discuss shortly.

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

$$\mathbf{x} = \sqrt{P} \cdot \mathbf{F}_{\mathrm{RF}}\mathbf{F}_{\mathrm{BB}}\mathbf{s} \tag{9}$$

where a digital precoding matrix $\mathbf{F}_{\mathrm{BB}}$ followed by an analog precoding matrix $\mathbf{F}_{\mathrm{RF}}$ are applied to symbol vector $\mathbf{s}$. A

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

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

The number of radio frequency (RF) chains, phase and amplitude resolution of analog beamforming, and connected-ness of the analog beamforming network can all be configured as desired.

Once setup, the hybrid transmitter's digital and analog precoders can be set via

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

where `F_BB` is an $L_{\mathrm{t}} \times N_{\mathrm{s}}$ digital precoding matrix and `F_RF` is an $N_{\mathrm{t}} \times L_{\mathrm{t}}$ analog precoding matrix.

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). For example, for eigen-based precoding, one can use

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

which will automatically 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.

B. Receivers

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

$$\hat{\mathbf{s}} = \mathbf{W}^{*}(\mathbf{y} + \mathbf{n}) \tag{10}$$

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 its 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, and symbol bandwidth (i.e., B). Like the `transmitter`, a `receiver` object's properties can be set using its various `set` commands.

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

$$\mathbf{n} \sim \mathcal{N}_{\mathbb{C}}\left(\mathbf{0}, \sigma_{\mathrm{n}}^2 \cdot \mathbf{I}\right) \tag{11}$$

where σ_{n}^2 is the average noise energy per symbol (joules) (i.e., the noise power spectral density). To set the noise energy per symbol, we can use

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

where `psd` is the noise power spectral density is in dBm/Hz.

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. Like the transmitter, receiver combining strategies can be specified using strings like `'eigen'` or `'mmse'` with `rx.configure_receiver(strategy)` rather than setting the combining matrix explicitly.

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

$$\hat{\mathbf{s}} = \mathbf{W}_{\mathrm{BB}}^{*}\mathbf{W}_{\mathrm{RF}}^{*}(\mathbf{y} + \mathbf{n}) \tag{12}$$

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')
```

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.

C. Devices

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 `device` can be created via

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

where `type` is either `'transmitter'`, `'receiver'`, or `'transceiver'` (default). It can be placed in 3-D space by setting its coordinate via

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

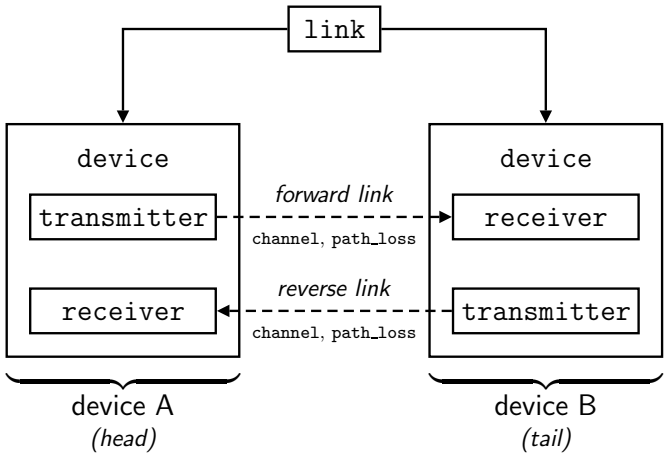

Fig. 4. A link established between two transceivers.

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. Other device parameters can be set straightforwardly using a variety of `set` commands.

In many ways, the `device` object acts as a proxy for configuring and interfacing with its transmitter and/or receiver. As such, the `device` is supplied with a number of *passthrough* functions that make directly interfacing with its transmitter and/or receiver simpler. For instance, the passthrough function

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

will set the transmit power of the device's transmitter, rather than the user needing to do

```
d.transmitter.set_transmit_power(P,'dBm')
```

Declaring which other device `dev` a given device `d` should transmit to (i.e., the *destination* device) is accomplished via

```
d.set_destination(dev)
```

where `dev` is a device with receive capability. Likewise, declaring the device it should receive from (i.e., *source* device) is accomplished via

```
d.set_source(dev)
```

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

D. Links

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.

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. 4. 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).

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

```
lnk.set_channel(chan_fwd,chan_rev)
lnk.set_path_loss(path_fwd,path_rev)
```

where `chan_fwd` and `chan_rev` are `channel` objects and `path_fwd` and `path_rev` are `path_loss` objects.

IV. Network-Level Usage

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, a network in MFM captures scenarios where all devices present in the network share the same time-frequency resource, meaning some degree of interference will be inflicted onto each receiver in the network by each transmitter.

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 useful.

To establish a physical connection (i.e., channel and path loss) between *each* transmitting device and *each* receiving device in the network, links can be added manually, though this is very cumbersome even for a small networks. 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 device to *each* destination device. 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, meaning the number of links is equal to the number of source-destination pairs squared.

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. MFM offers the user the convenience of setting various system parameters network-wide instead of setting them at each device one-by-one, such as the carrier frequency, noise power, etc.

Once it has been properly setup, 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. To collect and distribute channel state information across a network `net`, use

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

which can be used by devices to automatically configure themselves based on their transmit and receive strategies. To configure *all* 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.

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

```
net.compute_received_signals()
```

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

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)
```

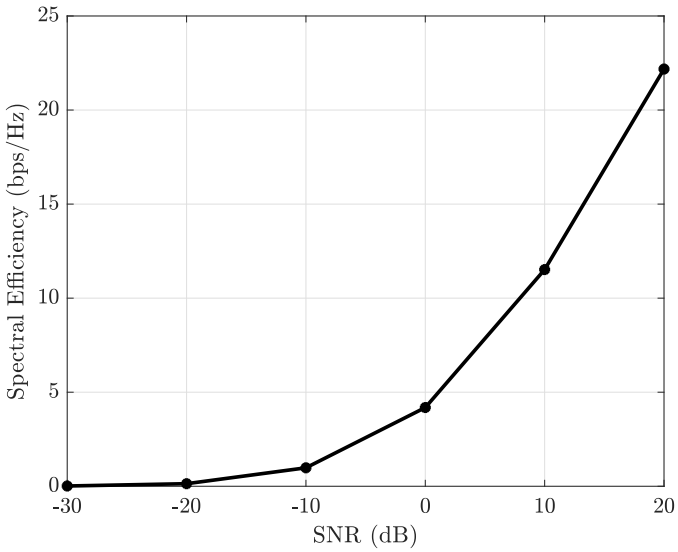

Fig. 5. The spectral efficiency of a point-to-point Rayleigh-faded network as a function of SNR simulated using MFM.

which will automatically account for interference caused by other transmitting devices in the network. 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 $\|\hat{\mathbf{s}}-\mathbf{s}\|_2^2$ and the second return value `nerr` is the symbol estimation error normalized to the transmit symbol energy defined as

$$\frac{\|\hat{\mathbf{s}}-\mathbf{s}\|_2^2}{\|\mathbf{s}\|_2^2} \tag{13}$$

V. Conclusion

MFM aims to improve the reproducibility of MIMO research by providing a common framework for researchers to use when implementing their work. Out-of-the-box, MFM is equipped with a variety of widely used channel and path loss models. Custom objects that integrate with MFM can be created independently by users by following a few simple rules. These third-party customizations to MFM can then be shared within the research community for others to use and potentially integrated into MFM itself. Beyond research, MFM's potential also lay in educating students on MIMO and general wireless, offering students and educators a tool to explore concepts numerically, mathematically, and algorithmically.

Acknowledgments

This work is 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.

REFERENCES

[1] R. W. Heath Jr. and A. Lozano, *Foundations of MIMO Communication*. Cambridge University Press, 2018.

[2] I. P. Roberts, "MIMO for MATLAB: A toolbox for simulating MIMO communication systems in MATLAB," http://mimoformatlab.com, 2021.

[3] C. Balanis, *Antenna Theory: Analysis and Design*. John Wiley & Sons, 2016.

[4] R. W. Heath, N. González-Prelcic, S. Rangan, W. Roh, and A. M. Sayeed, "An overview of signal processing techniques for millimeter wave MIMO systems," *IEEE J. Sel. Topics Signal Process.*, vol. 10, no. 3, pp. 436–453, Apr. 2016.

[5] J.-S. Jiang and M. A. Ingram, "Spherical-wave model for short-range MIMO," *IEEE Trans. Commun.*, vol. 53, no. 9, pp. 1534–1541, Sep. 2005.