EMerge Cheat sheet V1.1

Step 1. Setup

```python
import emerge as em
model = em.Simulation('Model Name')
```

Step 2. Modelling

Geometric

Axis
`ax = em.Axis([0,1,2])`
`em.XAX, em.YAX, em.ZAX`

Plane
`plane = em.Plane(ax1, ax2)`
`plane = ax1.pair(ax2)`
`plane = ax1 * ax2`

CoordinateSystem
`cs = em.CoordinateSystem(axx, axy, axz, pos)`
`cs = em.CS(...)`
`cs = ax.construct_cs(x0,y0,z0)`
`cs = ax1 * ax2 * ax3`
`cs = plane.cs(x0,y0,z0)`
`cs = em.cs('yzx',(x0, y0, z0))`

Construction

3D

Box `em.geo.Box(width, depth, height, pos)`

Sphere `em.geo.Sphere(rad, pos)`

Half Sphere `em.geo.HalfSphere(rad, pos, direction)`

Cylinder `em.geo.Cylinder(rad, length, cs)`

CoaxCylinder `em.geo.CoaxCylinder(ri, ro, length, cs)`

Horn `em.geo.Horn((w1,h1),(w2,h2),length, cs)`

Cone `em.geo.Cone((x0,y0,z0), (dx,dy,dz), r1, r2)`

2D

XYPlate `em.geo.XYPlate(width, height, pos)`

Plate `em.geo.Plate((x0,y0,z0),(ux, uy, uz),(vx, vy, vz))`

Disc `em.geo.Disc(origin, radius, axis)`

XYPolygon

Draw 2D polygons and embed in 3D

```python
poly = em.XYPolygon([x1, x2, ..., x3],[y1, y2, ..., y3])
```

`.extend` `poly.extend(xs, ys, reverse=False)`
Extend with extra coordinates

`.parametric` `poly.parametric(fx, fy, tolerance, ...)`
Extend with a parametric equation.

`.geo` `poly.geo(cs)`
Embed in 3D space in coordinate system

`.extrude` `poly.extrude(length, cs)`
Extrude along Z-axis

`.revolve` `poly.revolve(cs, origin, axis, angle)`
Revolve around axis

`.incs` Define a coordinate system without embedding

`.connect` Connect to another XYPolygon (requires calling `.incs()` on both)

Curves

```python
curve = em.geo.Curve(xpts, ypts, zpts, ctype='spline')
curve.pipe(crosssection=geo or XYPolygon, **options)
```

Operations

add `em.geo.add(obj1, obj2)`

remove `em.geo.remove(obj1, obj2)`

intersect `em.geo.intersect(obj1, obj2)`

unite `em.geo.unite(obj1, obj2, ...)`

mirror `em.geo.mirror(obj, origin, direction)`

rotate `em.geo.rotate(obj, origin, axis, angle)`

translate `em.geo.translate(obj, dx, dy, dz)`

change CS `em.geo.change_coordinate_system(obj, new_cs, old_cs)`

expand `em.geo.expand_surface(surf_obj, distance)`
Works on 2D surfaces only

embed `em.geo.embed(vol_obj, surf_obj)`
Embeds a surface into a volume.

stretch `em.geo.stretch(obj, fx, fy, fz)`

Final

```python
model.commit_geometry()
```

Step 3. Materials

Materials can be created using

```python
material = em.Material(er=er, ur=ur, cond=cond, tand=tand, color=...)
```

Properties can be frequency, coordinate or both frequency and coordinate dependent. Coordinate dependent properties need to be assigned a max value for meshing. Create function objects using:

```python
prop_freq = em.FreqDependent(lambda f: ...)
prop_xyz = em.CoordDependent(maxval: float, lambda x,y,z: ...)
prop_fxyz = em.FreqCoordDependent(maxval: float, lambda f,x,y,z: ...)
```

Assignment

Materials can be assigned using:

```python
box.set_material(my_material)
box.material = my_material
```

Priority

Two volumes that take up the same space will have their material assigned based on two comparisons
- First, the highest priority score (all default to 10)
- If the same score, use the creation order

Priorities can be changed using:

```python
obj.prio_set(value)
obj.prio_up() # +1
obj.prio_down() # -1
obj.foreground() # highest prio + 10
obj.background() # highest prio - 10
```

Step 4. Meshing

Physics size Set your simulation frequency and start resolution.

Resolution R is defined as $ds_{max} = R\lambda\Re\left(\sqrt{\varepsilon_r\mu_r}\right)$.

```python
model.mw.set_frequency_range(f1, f2, nF)
model.mw.set_resolution(0.3)
```

Maximum domain size Several options

```python
box.max_meshsize = 3*mm
model.mesher.set_domain_size(mybox, 3*mm)
model.mesher.set_face_size(mysurf, 3*mm)
```

Boundary size
- Boundary of volume = surface
- boundary of surface = edge etc.

```python
model.mesher.set_boundary_size(obj, 3*mm)
```

Generation

```python
model.generate_mesh()
```

Step 5. Selection

The following ways exist to select objects and faces

Convert object to selection of it

```python
my_object.selection
```

Bundling multiple objects in a selection

```python
em.select(my_obj1, my_obj2, ...)
```

Faces of objects

```python
my_box.face('front') # left, right, front, back, top bottom
my_cyl.face('front') # front/left/bottom and back/right/top
my_box.boundary(exclude=('front','right')) #all but the front and right
```

Faces of inherited objects

After boolean operations, the resultant object does not have the original face definitions. To still access them use the optional tool argument:

```python
my_box = em.geo.Box()
new_obj = em.geo.subtact(my_box, other_obj)
face = new_obj.face('front', tool=my_box)
```

Near a coordinate

```python
my_model.select.face.near(x,y,z)`
```

Faces in a plane

```python
my_model.select.in_plane(x,y,z,nx,ny,nz)
```

Faces in a layer

```python
my_model.select.in_layer(x,y,z,vector)
```

Step 6. Boundary Conditions

PEC `em.mw.bc.PEC(faces)`

PMC `em.mw.bc.PMC(faces)`

Lumped Port `em.mw.bc.LumpedPort(faces, number, width, height, dir, Z0)`

Wave Port `em.mw.bc.ModalPort(faces, number, TEM=False/True)`
Executes eigenmode analysis for wave mode.

Rectangular waveguide port `em.mw.bc.RectangularWaveguide(faces, number, mode)`

Floquet port `em.mw.bc.floquet_port(faces, number)`

User defined port `em.mw.bc.UserDefinedPort(faces, number, fEx, ...)`

Surface Impedance `em.mw.bc.SurfaceImpedance(faces, material)`

Absorbing Boundary `em.mw.bc.AbsorbingBoundary(faces)`

Lumped Impedance `em.mw.bc.LumpedImpedance(faces, value)`

PML

PML is technically not a boundary condition but a fictitious material property. Right now, the only way to create PML's effectively and easily is using the PML box:

```python
mainbox, *pml_regions = em.geo.pmlbox(dims, ...)
```

Periodic Boundary Conditions

To facilitate periodic cells, use the `em.RectCell()` and `em.HexCell()` functions.

```python
m = em.Simulation(...)
cell = em.RectCell(...)
m.set_periodic_cell(cell)
volume = cell.volume(z1, z2) # create periodic volume
fp = m.mw.bc.floquet_port(...)
cell.set_scan_angle(theta, phi)
```

Step 7. Simulation

EMerge can run either a frequency domain sweep or an eigenmode study.

Frequency Domain

```python
data = model.mw.run_sweep()
```

For parallel processing:
- `parallel: bool = True`: Use parallel processing
- `n_workers: int = 3`: Number of parallel workers
- `frequency_groups: int = 6`: The number of matrices to pre-compute before solving. Whole multiple of njobs is ideal smaller number means less RAM usage.
- `multi_processing: bool = True`: If Python's Multi-processing should be used instead of multithreading (required for UMFPACK). Requires entrypoint protection

Entry-point protection for multi-processing

```python
def main():
    # simulation setup
    data = model.mw.run_sweep(..., multi_processing=True)

if __name__=="__main__":
    main()
```

Eigenmode analysis

For more details on how the eigenmode analysis works and can be optimized, consult the manual.

```python
data = model.mw.eigenmode(search_freq, n_modes, ...)
```

Setting solvers

To set a specific solver use:

```python
model.mw.solveroutine.set_solver(em.EMSolver.UMFPACK)
```

available solvers:
- `em.EMSOLVER.SUPERLU`
- `em.EMSOLVER.PARDISO` (x86 Only)
- `em.EMSOLVER.UMFAPCK` (custom install)
- `em.EMSOLVER.CUDSS` (requires modern NVidia GPU)
- `em.EMSOLVER.LAPACK` (direct eigenmode solver)
- `em.EMSOLVER.ARPACK` (iterative eigenmode solver)
- `em.EMSOLVER.SmartARPACK` (ARPACK + search algorithm)

Disabling a solver from use:

```python
model.mw.solveroutine.disable(em.EMSolver.UMFPACK)
```

Parameter Sweep

To do a parameter sweep, define the parameter series as iterable.

```python
param1 = np.linspace(...)
param2 = np.linspace(...)

model = em.Simulation(...)

for p1, p2 in model.parameter_sweep(clear_mesh=True, param1=param2, param2=param2):
  # setup Simulation
  data = model.mw.run_sweep()
```

- `clear_mesh: bool = True`: If a parameter sweep changes the geometry, set `clear_mesh` to True. If you change boundary condition values and can keep the mesh, set it to false.
- Full outer-product of parameters is sweeped. Parameters are assigned by keyword name in alphabetical order. (See the manual for detail).

Adaptive Mesh Refinement (Beta)

One can do an adaptive mesh refinement step before solving as following. First use the `SimulationBeta()` class

```python
sim = em.SimulationBeta(...)
#make geometry
sim.adaptive_mesh_refinement()
sim.mw.run_sweep()
```

Step 8. Post Processing

All data returned by frequency sweep is returned in a large dataset in the simulation model.

```python
dataset = model.data
```

Physics related simulation results are stored in a separate property

```python
mw_data = model.data.mw
```

This is also what is returned by the `.run_sweep()` method.

```python
mw_data = model.mw.run_sweep()
mw_data == model.data.mw #Same objects!
```

Data categories

There are two physics data categories:
- **Global constants** : `wmdata.scalar`
- **Field data** : `mwdata.field`

Finding data

During a frequency-sweep, each port excitation field for the same frequency is stored in the same entry.

Each simulation run is stored in an internal entry in order of simulation with the specific parameters for which they where run.

By Order

```python
mwdata = model.mw.run_sweep()
field_data = mwdata.field[0] # first freq/param combi
field_data = mwdata.field[1] # second freq/param combi
```

By Parameter

```python
mwdata = model.mw.run_sweep()
field_data = mwdata.field.select(freq=1.2e9) #exact match
field_data = mwdata.field.find(freq=1.2e9) #closest match
field_data = mwdata.field.find(freq=1.2e9, param1=3, param2=8) # for paramter sweeps
scalar_data = mwdata.scalar.find(freq=1.2e9) #same as field
s11_1p2GHz = mwdata.scalar.find(freq=1.2e9).S(1,1)
```

NDim global results

If a simulation is run with a frequency sweep including possibly a parameter sweep, the simulation results can be cast into a structured grid (N-dimensional array)

```python
mwdata = model.mw.run_sweep()
gritted_mwdata = model.scalar.grid
freq = gritted_data.freq #1D array for freq sweep
S11arry = gritted_data.S(1,1) #1D array for freq sweep
```

Vector Fitting

Vector fitting is a technique to interpolate S-parameters based on modelling poles and zeros.

```python
mwdata = model.mw.run_sweep()
f_dense = np.linspace(f1, f2, 1001)
S11_dense = mwdata.scalar.grid.model_S(1,1,f_dense)
```

Working with field data

A field dataset returned by `mwdata.field[0]` for example, is not yet defined in space. It is a complex data structure.

The user has to probe this dataset at certain coordinates.

This action returns a class instance that contains all field variables

```python
mwdata = model.mw.run_sweep()
field_data = mwdata.field.select(freq=1.2e9) #exact match

xs = np.linspace(...)
ys = np.linspace(...)
zs = np.linspace(...)
X,Y,Z = np.meshgrid(xs, ys, zs)

field = field_data.interpolate(X,Y,Z)
Ex = field.Ex
Ey = field.Ey
#etc.
```

Creating grids

Creating the sample coordinates together with interpolation can be done with handy macros:

```python
mwdata = model.mw.run_sweep()
field_data = mwdata.field.select(freq=1.2e9)

xyplane = field_data.cutplane(ds, z=5*mm) #XY data at z=5mm
xyzgrid = field_data.grid(ds) #structured XYZ grid
uvplane = field_data.cutplane_normal(origin, normal, Npts)
boundary = field_data.boundary(face_selection)
```

Far-field Data

Far-field data can be computed from the field-data computed from a solution.

```python
mwdata = model.mw.run_sweep()
field_data = mwdata.field.select(freq=1.2e9)

data_plane = field_data.farfield_2d(ref, normal, face_selection)
data_sphere = field_data.farfield_3d(face_selection, thetas, phis, origin)
```

Step 9. 3D Plots

3D plots can be made with the PyVista interface. To display the geometry of the simulation model use:

```python
model.view() #basic view
model.view(plot_mesh=True) #plot the 3D Mesh
model.view(plot_mesh=True, volume_mesh=False) #Show mesh only on surface.
model.view(labels=True) # Add name labels for all objects
model.view(selections=[sel1, sel2]) #highlights selections
```

One can also construct a custom view with the following commands.

```python
display = model.display #shorter
#then
display.add_object(my_obj, opacity=...)
display.add_objects(my_obj1, my_obj2)
display.add_objects(*sim.add_geos()) # All geometries
display.add_surf(X,Y,Z, field) #surface plot
display.add_boundary_field(face, field.boundary(face).scalar())
display.add_portmode(port_bc, k0=...) #k0 is needed for selecting the right mode
display.show()
```

Cutplanes can easily be viewed using the `.scalar()` method:

```python
data = model.mw.run_sweep()
model.display.add_surf(*data.field[0].cutplane(...).scalar('Ez','real'))
```

Objects can be hidden:

```python
box = em.geo.Box(...)
box.hide() # Will not be shown
```

Animations

Some views can be animated using:

```python
display.animate().add_surf(*data.field[0].cutplane(...).scalar('Ez','complex'))
display.show()
```

Use Q to quit the viewer as simly clicking away freezes the process.