OAIQSim: User's Guide

Nick Henscheid
University of Arizona Department of Medical Imaging

September 13, 2019

1 Introduction

This is the **user's guide** documentation for the Objective Assessment of Image Quality Simulation package (OAIQSim). The objective of this code package is to provide a user-friendly, *holistic* system simulation and evaluation package for emission imaging systems. In a nutshell, OAIQSim provides powerful but user-friendly computational methods for simulating the complete imaging chain:

$$\begin{tikzcd} \boxed{\begin{array}{c}\text{Object}\\ \boldsymbol{f}=\boldsymbol{f}(\boldsymbol{r},t)\end{array}} \arrow[r] & \boxed{\begin{array}{c}\text{Imaging data}\\ \boldsymbol{g}=\boldsymbol{\mathcal{H}}\boldsymbol{f}+\boldsymbol{n}\end{array}} \arrow[r] & \boxed{\begin{array}{c}\text{Decision}\\ \boldsymbol{d}=\boldsymbol{\mathcal{D}}(\boldsymbol{g})\end{array}} \end{tikzcd}$$

Figure 1: The Image Chain

This document will get the user started with included examples and the core Matlab objects. Note that large parts of OAIQSim are built on a `C++` and `CUDA` backend, with the interface to Matlab coming via the `mex` mechanism. Note that while a `CUDA`-capable GPU is not required, one is recommended to get the full usage of the package. This document will not discuss the backend code - see the full `oaiqsim_dev.pdf` for a more complete discussion of the package, image quality assessment theory, numerical methods and software implementation. Contact me at nph@email.arizona.edu for questions or concerns.

2 Compiling and Starting

To start using the package, follow these steps:

1. Some parts of the package require compilation using `mex`. If this is your first time using the package, navigate to the `OAIQSim` folder and type `compile` in the Matlab command window. The compilation will ask a few questions then compile the necessary backend functions. This process takes a few minutes. If the `compile` script results in an error, contact me and we'll try to debug the issue. **The most common errors result from incorrect `mex` configuration; I strongly recommend working through Matlab's `mex` tutorial available on their website if you have never used `mex`.**

2. If you have already compiled the package, navigate to the `OAIQSim` folder and type `start` in the Matlab command window. This adds the relevant folders to your Matlab path.

3. Open one of the example scripts found in `/examples`, e.g. `LumpyExample.m`. Inspect the default settings such as blah blah should write a 'master example' script.

4. Check out the other example scripts like blah blah

5. Modify or write your own script and enjoy!

The remainder of this document is split into three sections, corresponding to the three blocks of the image chain as displayed in Figure 1.

3 Object Models

Very briefly, an *object model* is intended to simulate some spatiotemporal property of an object that is going to be imaged. While a truly general imaging system simulation could, in theory, aim to simulate nearly any physical property of nearly any object, OAIQSim is intended to be used to simulate exclusively medical emission imaging systems, so the object models employed in OAIQSim are intended to reflect the properties of organisms that are relevant to this class of modalities. These properties mostly pertain to the behavior of drug concentrations in tissue (molecular diffusion) and the propagation of light in tissue (absorption and scattering at optical-to-gamma-ray energies i.e. approximately 1eV-1MeV). In this community, object models are more commonly called *computational phantoms*, and the phantoms required for emission imaging are usually *functionalized*; see `oaiqsim_dev.pdf` for more details.

Currently, the object models available in OAIQSim consist of a variety of random field models defined on basic 2D and 3D domains. In a later version of the package, we will investigate more anatomically and physiologically realistic computational phantoms.

For the remainder of this section, we denote an arbitrary random field by $\boldsymbol{f}(\omega) = f(\boldsymbol{\tau}, \omega)$, where $\boldsymbol{\tau}$ is a generic 2D, 3D or 4D spatial or spatiotemporal variable and $\omega \in \Omega$ is a 'stochastic' variable.

3.1 Lumpy-type Random Fields: `LumpyBgnd`

A lumpy-type random field has realizations which take the form

$$f(\boldsymbol{\tau}, \omega) = A + \sum_{k=1}^{K} b_k \ell(\boldsymbol{\tau} - \boldsymbol{\tau}_k; \boldsymbol{\alpha}_k) \tag{1}$$

Any or all of $(A, K, b_k, \boldsymbol{\tau}_k, \boldsymbol{\alpha}_k)$ can be randomized to generate different behavior profiles. If the lump function $\ell(\boldsymbol{\tau}; \boldsymbol{\alpha})$ is non-negative, then (1) is guaranteed to produce non-negative realizations. We typically assume K is a Poisson random number and $\boldsymbol{\tau}_k$ are I.I.D. uniform in an extended domain $V' \supset V$ (the extended domain prevents boundary artifacts). The package currently assumes $\ell(\boldsymbol{\tau}; \boldsymbol{\alpha})$ is an un-normalized 2D or 3D Gaussian function with covariance matrix $\boldsymbol{\alpha} = \mathbf{K}$ i.e.

$$\ell(\boldsymbol{\tau}; \boldsymbol{\alpha}) = \ell(\boldsymbol{\tau}; \mathbf{K}) = \exp\left(-\frac{1}{2}\boldsymbol{\tau}^{\mathsf{T}}\mathbf{K}^{-1}\boldsymbol{\tau}\right) \tag{2}$$

A demonstration of a basic lumpy-type random field behavior is shown in Figure 2.

3.1.1 Package Implementation

In the OAIQSim package, the Matlab object `LumpyBgnd` implements a fairly general class of lumpy-type models of the type (1) with Gaussian lump function (2). **See the example scripts `Lumpy_Bgnd_2D.m` and `Lumpy_Bgnd_3D.m` to explore the possibilities.** The script `Lumpy_Bgnd_2D.m` will produce the figure seen in Figure 2.

Instantiating a `LumpyBgnd` object At the Matlab command window, typing

```
L = LumpyBgnd
```

will instantiate a Lumpy Background object with default settings. Typing

```
plot(L)
```

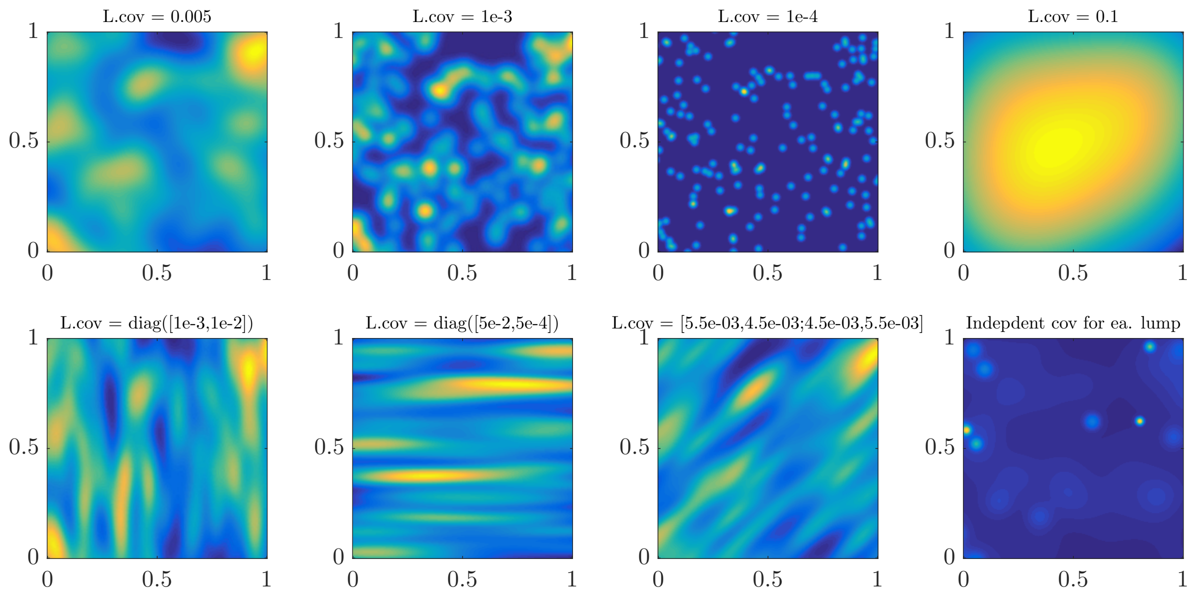

Figure 2: Example Gaussian lumpy backgrounds MAKE SURE SCRIPT IS AVAIL.

will run a plotting routine that evaluates the current realization of the field and displays a pseudocolor plot. The resulting default object (which is automatically randomized!) and a plot of the realized field are shown in Figure 3. The method `L.plot` begins by evaluating the field on a default rectilinear grid using the method `L.eval`, which we describe in more detail below.

We will begin by describing the properties of `LumpyBgnd`, then describe its methods.

Properties for the `LumpyBgnd` Object We now discuss each of the object properties, displayed in Figure 3.

- `L.b0` is the lump 'amplitude', written as b_k in (1). By default, this is a non-random constant, but in theory each lump can have its own amplitude. To employ this, one can directly modify `b0` by setting `L.b0 = b` where `b` is some array of size `L.K`-by-1. **Note:** you must make sure that the size of `L.b0` is correct, otherwise you'll get an error when trying to evaluate the field. An example of this feature is shown in Figure 4.

- `L.B0` is the field offset ('DC shift'), written as A in (1). This is typically taken to be either zero or some small number to keep the field strictly positive (which is necessary for some of the RDE coefficients). This parameter can be set manually via `L.B0 = B` where `B` is some scalar quantity.

- `L.cov` is the lump covariance matrix. The default is `L.cov = 0.005*eye(2)`. To modify the lump covariance, you can set `L.cov` using one of several methods. First, simply entering `L.cov = a` where `a` is any (positive) scalar quantity will set the covariance matrix to `L.cov = a*eye(2)`. Similarly, setting `L.cov = S` where `S` is a matrix will set the lump covariance matrix to `S`. You can also give each lump a different covariance matrix by setting `L.cov = S` where `S` is `L.dim`-by-`L.dim*L.K`.

  Warning: it's up to you to make sure the covariance matrix is valid (i.e. positive definite). The code does not check this and bizarre output may result if `L.cov` is invalid!

- `L.Kbar` is the mean number of lumps. In (1), the standard assumption is that the realized number of lumps K is a Poisson random number with mean `L.Kbar`.

```
L =
  LumpyBgnd with properties:
                    b0: 1
                    B0: 0
                   cov: [2x2 double]
                  Kbar: 100
               centers: [106x2 double]
                   gpu: 1
                     N: 128
        israndnumlumps: 1
                     L: [2x2 double]
                   dim: 2
                     K: 106
                     S: [1x1 RectSupport]
             padfactor: 3
          showwarnings: 1
```

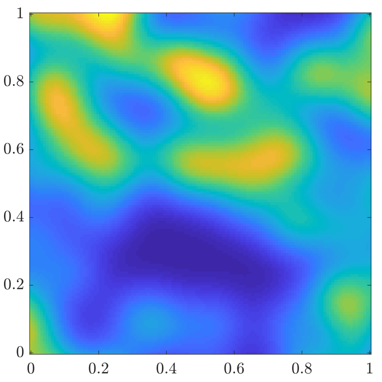

Figure 3: Default Lumpy Background Object

- `L.centers` is an `L.K`-by-`L.dim` matrix that stores the realized lump centers, written as $\boldsymbol{x}_k$ in (1). The first column is the x-coordinate, the second column the y-coordinate, and the third column (in 3D) is the z-coordinate. The lump centers can be modified manually by accessing `L.centers`. For instance, during a parameter estimation or Markov Chain Monte Carlo simulation, one may wish to 'manually' move the lump centers to fit some quantity. Suppose we wish to place the lumps on a uniform 8×8 grid - we can build the grid coordinates `X = [x,y]` using e.g. `meshgrid`, then set `L.centers = X`. The result is shown in Figure 5. Note: `L.K` and `L.dim` are tied to the dimension of `L.centers` (they are 'dependent' object properties).

- `L.gpu` is a boolean variable that determines whether the random field is evaluated on the Graphics Processing Unit (GPU). For small mesh sizes and in 2D, the time savings is somewhat marginal as the CPU-GPU memory transfer consumes more time than the actual evaluation. However, for dense meshes, (larger than say, 512×512 in 2D), very large number of lumps `L.K` (larger than 500, say) or for 3D evaluation, the GPU offers vast time savings. For example, on my current 2019 workstation (Intel Xeon with NVidia Quadro P4000), a background with `L.K = 4096`, `L.dim = 2` and `L.N = 1024` takes 45 seconds to evaluate on the CPU, and only 2 seconds to evaluate on the GPU. When `L.K = 64` and `L.N = 128`, the CPU is actually much faster: 0.027 seconds versus 0.57 seconds for the GPU. **Note: by default, `L.gpu` will be set to 0 (and cannot be changed) if your computer does not have a compatible GPU device, and set to 1 if it does (but can be changed to 0). You can check if your computer has a compatible device by calling `gpuDevice` in the Matlab window.**

- `L.N` is a grid size parameter for evaluating the field. By default, calling the method `L.eval` will first generate a rectilinear mesh using `meshgrid`, then evaluate the realized field on this grid. The parameter `L.N` sets the size of this default grid: it will be `L.N`-by-`L.N` (in 2D) or `L.N`-by-`L.N`-by-`L.N` in 3D. In previous iterations of the `LumpyBgnd` object, a full grid was stored as an object property, but this way is much more lightweight (the grid is only created when it's needed).

- `israndnumlumps` is a boolean variable that determines whether or not the number of lumps (`L.K`) is randomized when the method `L.randomize` is called. Occasionally, it is convenient to fix the number

```
L =
  LumpyBgnd with properties:

                    b0: [106x1 double]
                    B0: 0
                   cov: [2x2 double]
                  Kbar: 100
               centers: [106x2 double]
                   gpu: 1
                     N: 128
         israndnumlumps: 1
                     L: [2x2 double]
                   dim: 2
                     K: 106
                     S: [1x1 RectSupport]
             padfactor: 3
          showwarnings: 1
```

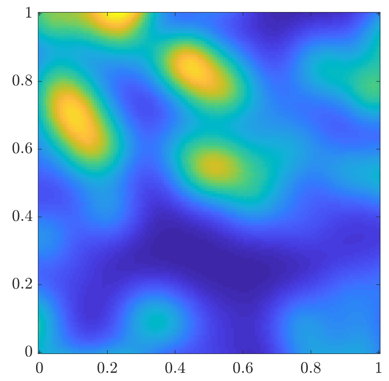

Figure 4: Lumpy Background Object with random lump amplitudes, generated using `L.b0 = rand(L.K,1)`. Note that nothing else was changed from the figure shown in Figure 3!

of lumps at a non-random number. To set this property, call `L.israndnumlumps = a` where `a` is either 0 or 1.

- `L.L` is an `L.dim`-by-2 matrix that defines the support of the random field, in the format `L.L = [a,b;c,d]` (in 2D). This is a 'dependent' property that is fixed by a corresponding matrix in the property `L.S`; see below.
- `L.dim` is the dimension of the random field. This is a 'dependent' property that is set automatically to be the number of columns in `L.centers`, and cannot be set by the user.
- `L.K` is the number of lumps in the current field realization. This is also a 'dependent' property, fixed by the number of rows in `L.centers`, and cannot be set by the user.
- `L.support` is a support set object that defines the support of the random field. Currently, only rectangles are supported via the `RectSupport` object, and currently **the support set can only be set when the `lumpybgnd` object is first instantiated**. To modify `L.S`, instantiate the `LumpyBgnd` object with the optional `'S'` argument, for instance

  ```
  L = LumpyBgnd('S',RectSupport([a,b;c,d]))
  ```

  will instantiate a `LumpyBgnd` with rectangular support set $[a, b] \times [c, d]$.
- `L.padfactor` is a padding factor to reduce boundary artifacts when generating realizations of the field. This has the effect of expanding the support set to a larger $V' \supset V$, then randomizing the lump centers on V' but only evaluating on V. By default, the support will be made `L.padfactor*sqrt(max(L.cov(:)))` larger. For instance, with `L.padfactor = 3`, we are expanding the domain by 'three sigma', where 'sigma' is the standard deviation of the lumps. To change `L.padfactor`, use the method `L.setPadFactor(p)` where `p` is some (non-negative) scalar. To see what happens when the pad factor is set to zero, see Figure 6.

```
L =
  LumpyBgnd with properties:

                  b0: 1
                  B0: 0
                 cov: [2x2 double]
                Kbar: 100
             centers: [64x2 double]
                 gpu: 1
                   N: 128
      israndnumlumps: 1
                   L: [2x2 double]
                 dim: 2
                   K: 64
                   S: [1x1 RectSupport]
           padfactor: 3
        showwarnings: 1
```

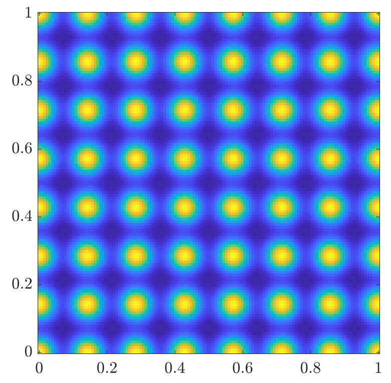

Figure 5: Lumpy Background Object with manually fixed lump centers, set using `L.centers = X` where `X` was constructed using `meshgrid`.

- `L.showwarnings` is a boolean that determines whether to show various warning messages. For instance, if the user sets `L.cov`, a warning message appears that suggests re-randomizing to avoid the boundary effect shown in Figure 6 (if the centers were selected according to a domain padding determined by a small lump size, switching to a larger lump may introduce boundary effects). This property can be set by the methods `L.turnOffWarnings` and `L.turnOnWarnings`.

Methods for the `LumpyBgnd` object

- `L.plot` plots the current realization using `imagesc` in 2D to display the resulting field, and `isosurface` to display the field in 3D. You can make your own specialized plots by first evaluating the field using `L.eval` (described below).

- `L.eval` evaluates the current realization on either a default grid (specified by the support `L.S` and the grid size `L.N`), or a user-supplied set of points. To call this method for the default grid, set `L.N` and then call `u = L.eval;`. The result is an `L.N`-by-`L.N` array with values $f(\boldsymbol{x}_{ij}, \omega)$. The evaluation is done in a `mex` file, either on the CPU or GPU depending on if `L.gpu` is 0 or 1. To supply your own set of points, you can call `u = L.eval(X)` where `X` is an array of size `M`-by-`L.dim`, where `M` is the number of evaluation points. In other words, `X = [x1,y1;x2,y2;x3,y3;...;xM,yM]`. **Note that you must pass the array in this format - `L.eval` currently does not work with the '`meshgrid`' format, for instance**. In this case, the returned array will be `M`-by-1, so you will need to use `reshape` to make it the desired size. One possible reason for evaluating the field on a non-standard grid is if you are working with an unstructured grid (e.g. for finite elements), and thus would need to evaluate the field on some arbitrary collection of (x, y) points. Another reason is if the field is being used as a(n unnormalized) probability density, in which case we may want to evaluate $f(\boldsymbol{x})$ at an arbitrary point.

- `L.randomize` randomizes the field according to the settings specified in the object properties. Specifically, if `L.israndnumlumps=1`, a new number of lumps `K` is chosen using `K = poissrnd(L.Kbar)`.

```
L =
  LumpyBgnd with properties:

                    b0: 1
                    B0: 0
                   cov: [2x2 double]
                  Kbar: 1000
               centers: [1031x2 double]
                   gpu: 1
                     N: 128
         israndnumlumps: 1
                     L: [2x2 double]
                   dim: 2
                     K: 1031
                     S: [1x1 RectSupport]
             padfactor: 0
          showwarnings: 1
```

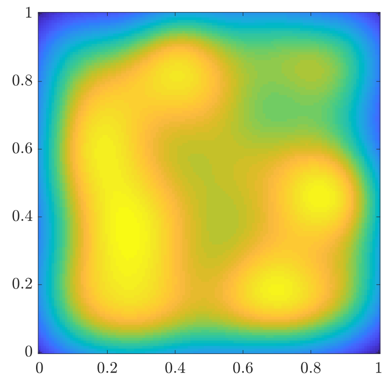

Figure 6: Lumpy Background Object with `L.padfactor=0`, demonstrating the boundary artifact resulting from not ‘padding’ the domain.

Then, the centers are randomized using `L.centers = rand(K,L.dim)` (assuming uniformly distributed centers on $[0,1]^2$ - `rand()` is replaced with an appropriate RNG for other domains/extended domains).

- `L.sample` produces a cell array of `N` random field samples, evaluated on the default grid, by calling `U = L.sample(N)`. For each sample, `L.randomize` is called followed by `L.eval`, and the resulting field is stored in `U{i}`.
- `L.setrandnumlumps`, called with a boolean argument, sets the property `L.israndnumlumps`.
- `L.turnOffWarnings` and `L.turnOnWarnings` set the property `L.showwarnings`.

Advanced Initialization If you do not wish to modify the settings of the random field after initialization, I have implemented a `varargin` feature that allows for comma-separated ‘setting’,‘value’ initialization. For instance, to initialize a `LumpyBgnd` with `Kbar = 500`, `K = 0.1*eye(2)`,

Warnings and Notes

- The `LumpyBgnd` object is a *handle class*, which means you must be careful when copying an instance. For instance, `L1 = LumpyBgnd` and `L2 = L1` means that any changes made to `L1` will automatically propagate to `L2`. Read the Matlab documentation for more information.
- There are several ways to mess up the parameters in the `LumpyBgnd` object. Safeguards are in place to prevent any severe bugs (like segmentation faults) when the `mex` file is called, but I haven't tried to think of everything. When modifying the properties of a `LumpyBgnd` object, be careful and refer to this document to determine the correct way to do it.

3.2 Gaussian and Lognormal Random Fields

4 Imaging System Simulation

5 Task Performance and Model Observers