Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

GraSPI: Extensible software for the graph-based quantification of morphology in organic electronics

Devyani Jivani [a], Jaroslaw Zola [b], Baskar Ganapathysubramanian [c], Olga Wodo [a,*]

[a] *Materials Design and Innovation, University at Buffalo, Buffalo, NY, USA*
[b] *Computer Science and Engineering, University at Buffalo, Buffalo, NY, USA*
[c] *Mechanical Engineering, Iowa State University, Ames, IA, USA*

A R T I C L E I N F O

Article history:
Received 2 April 2021
Received in revised form 18 October 2021
Accepted 20 December 2021

Keywords:
Graph theory
Materials informatics
Microstructure descriptors
Organic photovoltaics

A B S T R A C T

We describe GraSPI - extensible graph-based software implemented as a C/C++ package. GraSPI computes a large set of descriptors relevant to organic electronics given a segmented 2D or 3D microstructure. The package represents a microstructure as an equivalent graph and harnesses algorithms from graph theory to compute those descriptors efficiently. It also includes a suite of tools for converting data between various formats and post-processing the raw results from the graph analysis. Herein, we provide illustrative examples of GraSPI's capabilities in extracting microstructure descriptors and demonstrate the advantages that a graph-based approach affords via computational complexity analysis.

Code metadata

Current code version	v1.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00061
Legal Code License	BSD license
Code versioning system used	git
Software code languages, tools, and services used	C++, python, bash
Compilation requirements, operating environments	Boost library
Developer documentation/manual	https://owodolab.github.io/graspi/
Support email for questions	olgawodo@buffalo.edu

1. Motivation and significance

The aim of GraSPI, the package described in this paper, is to provide an efficient, extensible tool for microstructural data analytics. Such a tool can streamline the ability of material scientists to extract physics-informed features from a microstructure dataset. The availability of such a suite of features can greatly simplify and democratize capacity to map and explore structure–property (SP) relationships. Although the package's design is general, our primary focus of application is the active-layer morphology of organic electronic devices. Numerous emerging fields, including wearable electronics [1], multiplexed sensors [2], and bioelectronics [3], would significantly benefit from an improved understanding of those underlying SP relationships.

Extensible software frameworks such as ours are timely as more structure–property maps relying on the morphological descriptors are being established for organic electronics [4]. The organic electronic industry now has the potential to inexpensively and automatically characterize and image vast numbers of samples [5,6] produced by a combinatorial exploration of processing conditions. The ability to collect such data far outstrips the current capacity to understand and reason with those data [7]. Thus, software tools able to rapidly, efficiently, and automatically extract physically meaningful features from each image can pave the way for identifying meaningful process–structure–property (PSP) maps. Identifying PSP maps is rarely a trivial task due to the inherent mismatch between microstructural information that is observed (e.g., via microscopy or simulations) and the significantly fewer (i.e., limited) degrees of freedom needed to describe a PSP map. After all, whereas the goal of microstructural imaging is to provide detailed, high-resolution maps that produce

* Corresponding author.
E-mail address: olgawodo@buffalo.edu (Olga Wodo).

https://doi.org/10.1016/j.softx.2021.100969

high-dimensional datasets, the goal of establishing quantitative SP links is to derive a small set of features that can explain the most variability in the material properties. Further complicating that quandary is the fact that the desired set of features may not be directly measurable or even known *a priori*.

GraSPI computes a set of features[1] from a microstructure that can be used for subsequent data analytics and machine learning, especially in organic electronics. For input, the package uses a segmented micrograph with two or more phases to calculate a set of features that comprehensively captures descriptors of size, shape, and topology. GraSPI focuses on the microstructure quantification of organic photovoltaics (OPV) [8,9], modern versions of which have a so-called "bulk-heterojunction" (BHJ) architecture in which the photovoltaic active layer is a blend of two materials: electron donors and electron acceptors. The morphology of thin OPV films directly affects the physical processes within the cells and thus impacts the performance of the electronic devices. Therefore, to quantify, understand, and optimize the relationships between OPV morphology and performance, we have focused on the microstructural features that encode diffusion, transport, and interfacial characteristics.

Several research groups have used those features to establish SP maps [10,10–16] in OPV and leveraged those maps for microstructure-sensitive design [17]. In its current form, GraSPI handles the quantification of two-phase morphology, although all data structures are generalizable to handle multiphase morphologies [18]. Moreover, we have ensured extensibility by making the data structures agnostic to dimension as well as able to handle both structured (i.e., lattice) and unstructured (i.e., point cloud) data. Such capacities have been particularly useful in related work [19,20], in which data from molecular dynamics simulations and quantum chemical calculations were coupled to establish SP relationships across multiple scales using the same graph-based representation of the morphology.

2. Software description

GraSPI is built on the concept of the graph-based representation of a microstructure. The segmented, digitized morphology is represented as a labeled, weighted, undirected graph. Each pixel, or voxel in 3D, becomes a graph vertex with a label denoting its phase, and all vertices are connected with edges that encode information about distances. Graph construction for simple two- and three-phase morphologies are illustrated in Fig. 1. Once the morphology is represented as a graph, standard algorithms from graph theory are used to quantify information about shortest paths and connectivity. In the following paragraphs, we first introduce a few definitions of graphs and their features, followed by the formalization of graph construction from a digitized morphology.

2.1. Basic definitions

- An undirected graph $G = (V, E)$ is defined by a set of vertices, V, and a set of edges, E, where each edge in E is an unordered pair of vertices drawn from V.
- A weighted undirected graph $G = (V, (E, W))$ is an undirected graph (V, E) with an associated weight function, $W : E \rightarrow \mathbb{R}_+$, that assigns a non-negative real weight to each edge in E.
- A labeled weighted undirected graph $G = ((V, L), (E, W))$ is a weighted undirected graph $(V, (E, W))$ with an associated labeling function, L, that assigns a label to each vertex in V. In this work, we label each vertex with a color.
- A path between a source vertex, $s \in V$, and a target vertex, $t \in V$ is a sequence $p = [v_0, v_1, \dots v_i \dots v_k]$ of vertices such that $v_o = s$, $v_k = t$ and for each i from 0 to $i - 1$, vertices v_i and v_{i+1} are adjacent in G. The length of path p is $\sum_{i=0}^{k-1} w(e(v_i, v_{i+1}))$.
- A shortest path between a source vertex $s \in V$ and a target vertex $t \in V$ is a path between s and t that is of the shortest length among all paths between s and t in G. The distance between vertices s and t in G is the length of a shortest path between s and t in G. If no such path exists, the distance is defined as infinity. Note that the shortest path between a pair of vertices need not be unique, but the distance between them is unique.
- A subgraph of G is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. A vertex-induced subgraph on vertex set $V' \subseteq V$ is the maximal subgraph with the vertex set V'.
- A graph G is connected if there is a path between any pair of vertices in G. A connected component C in G is a maximal connected subgraph of G.

GraSPI has three building blocks: graph construction, shortest path and connectivity calculations, and graph filtering to tailor graph queries that define the targeted descriptors. Several diffusion, interfacial, and transport descriptors are recast as queries on the constructed graphs. All descriptors defined in the package depend on those building blocks.

2.2. Graph construction

A segmented morphology is the input for the software. Without the loss of generality, a labeled weighted undirected graph $G = ((V, L), (E, W))$ for a two-phase, two-dimensional morphology can be constructed, which corresponds to the top row in Fig. 1. A vertex $v \in V$ corresponds to an individual pixel or voxel in the morphology. Each vertex $v \in V$ is assigned a label $L(v)$ as a "BLACK", "WHITE", or "GREY" index depending on the phase of the respective pixel. Vertices are connected via a set of edges E.

The inherent structure of the morphology (e.g., pixel locations on a uniform lattice) is used to construct the set of edges E. For each pixel in the digitized morphology, the local neighborhood is established; for example, a pixel can have eight neighbors in 2D, hence a vertex corresponding to a pixel can have up to eight neighbors in the graph. An edge between a pair of vertices corresponds to the neighboring pixels' positions. Each edge $e = (u, v) \in E$ is assigned a weight $W(e)$ equal to the Euclidean distance between the pixels corresponding to u and v in the morphology. First-order neighbors one lattice distance away have an edge weight of 1, whereas second-order neighbors have an edge weight of $\sqrt{2}$. For 3D systems, third-order neighbors are included as well.

The graph G can be constructed for several types of data. In Fig. 1, the graph construction is for three types of data, with structured two-phase morphology, structured three-phase morphology, and unstructured two-phase morphology, respectively. To address various types of data, GraSPI offers two options for inputting the data required to construct the graph. The first option allows reading the structured morphologies and can be used when the phases of the input morphology are discretized on a structured grid. In that case, the neighborhood is defined through the edges, and vertices are simultaneously created and added to the graph. The second option is for unstructured data. In that case, the data are read from the input file according to the internal format described in Appendix. That option allows handling

[1] We use feature and descriptor interchangeably to mean the same thing. Features are commonly used in the machine learning field, while descriptors are more commonly used in materials science literature.

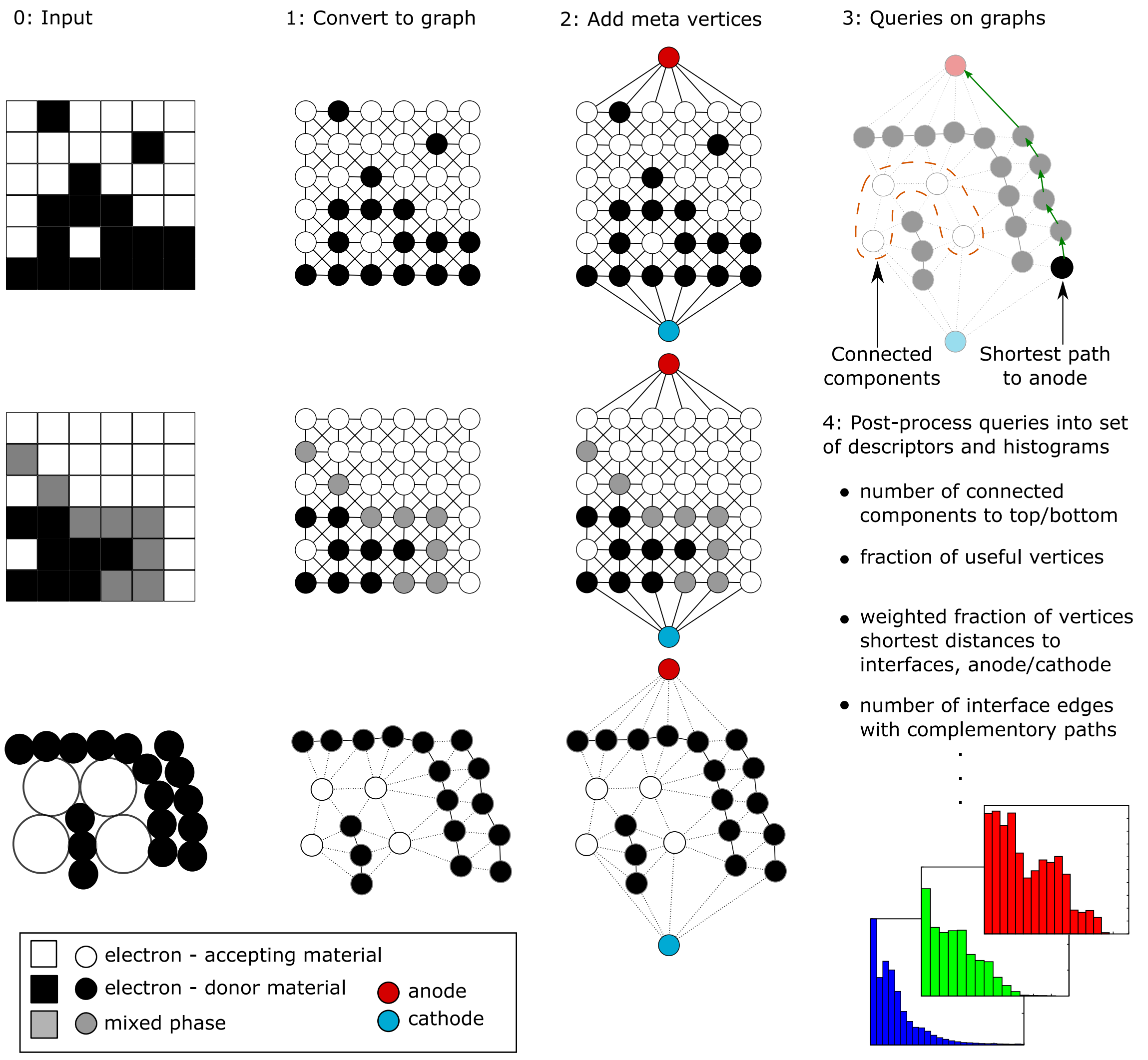

0: Input
1: Convert to graph
2: Add meta vertices
3: Queries on graphs
Connected components
Shortest path to anode
4: Post-process queries into set of descriptors and histograms
• number of connected components to top/bottom
• fraction of useful vertices
• weighted fraction of vertices shortest distances to interfaces, anode/cathode
• number of interface edges with complementory paths
electron - accepting material
electron - donor material
mixed phase
anode
cathode

Fig. 1. Graph construction for three types of morphology: structured two-phase morphology (the first row), structured three-phase morphology (the second row), and unstructured two-phase morphology (the third row). Stage 0 represents input with digitized morphology. In Stage 1, each unit element (e.g., pixel or atom) of morphology is represented as a vertex in the graph with a corresponding color assigned. Local neighborhood is used to connect vertices through edges. In Stage 2, meta-vertices are added to the graph that represent characteristic elements or landmarks—in our case, anodes and cathodes, with the anode vertex connected to all top vertices via additional edges and the cathode vertex connected to all bottom vertices. In Stage 3, graph queries are made on the filtered graph, whereas Stage 4 involves the post-processing of descriptors in terms of histograms. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

unstructured datasets where the neighborhood is specific to location and needs to be determined externally (e.g., using Voronoi diagrams or k-nearest neighbors). One can further expand the functionality of the package to account for multi-phase material systems fabricated for ternary and quaternary blends as these are promising candidates for high-performing devices [21,22]. With the current design of the package, the extension requires that the set of vertex labels and the set of meta-vertices are expanded accordingly. It also requires that the corresponding graph queries are redefined accordingly.

In the second stage, depicted in the third column in Fig. 1, more meta-vertices are added to the graph. For OPV morphologies, two types of meta-vertices are added. The first type facilitates the extraction of information with respect to the electrodes: anodes and cathodes, respectively on the red and blue vertices in the figure. The second type extracts information about the interface (i.e., GREEN vertex). For two-phase morphologies, only one type of interface exists between BLACK and WHITE vertices. That interface is tracked, and the edges that connect a BLACK and a WHITE vertex are deleted and subsequently connected via an added meta-vertex (i.e., GREEN vertex).[2] Once edges are added to the meta-vertices, weights are assigned to them. Edges of weight $W = 1$ are added between the anode (or cathode) and the vertices $v \in V$ that are placed physically adjacent to the anode (or cathode). Meanwhile, the edges of weight 0.5 are added to represent the connections between the interface vertex and the BLACK or WHITE vertices, $v \in V$. The anode, cathode, and interface vertices are labeled "anode", "cathode", and "interface", respectively. The added vertices allow for a straightforward estimation of graph distances from any location in the domain of the morphology to the electrodes, as well as of any distance from any point of the domain to the interface.

Once the graph is constructed, its quantification becomes independent of the original dimensions (i.e., 2D or 3D) and type (i.e., structured or unstructured). The quantification of morphology is recast as a graph query (i.e., Stage 3 in Fig. 1), which

2 Interfaces in three-phase morphologies are stored with two meta vertices: dark green and light green. They extract the connections between BLACK and GREY phases, and WHITE and GREY phases.

Listing 1

```
class edge_same_color_predicate {
public:
    edge_same_color_predicate() : G_(0), color_(0) { }
    edge_same_color_predicate(const gt::graph_t& G, const std::vector<COLOR>& color)
        : G_(&G), color_(&color) { }
    bool operator()(const gt::edge_t& e) const {
        return ((*color_)[boost::source(e, *G_)] == (*color_)[boost::target(e, *G_)]);
    }
private:
    const gt::graph_t* G_;
    const std::vector<COLOR>* color_;
};
```

relies on constructs from graph theory (e.g., all-pairs shortest paths, connected components (CC), and breadth- and depth-first searches) described in the next subsection.

2.3. Three basic operations on graphs

For a targeted application, characterization can be posed as a graph query operation using two graph-based algorithms: finding CCs and computing the shortest paths. Those algorithms are defined in Sections 2.3.1 and 2.3.2 and are used to construct application-specific queries. We provide the intuition behind the example presented and explain the details of the execution. In so doing, we pay special attention to the graph-filtering operation that enables the query definitions to capture some aspects of the underlying physics.

2.3.1. Identifying connected components of a graph

We start by elucidating the link between quantifying the number of individual domains and the graph queries. Here, our aim is to identify the subdomains of the morphology, which are surrounded by subdomains of other color(s). We accomplish that end by assigning an index of their CC to each vertex (Fig. 2). That process requires two steps. First, we define the filtered graph by masking the edges connecting vertices of different labels or colors. In other words, we ensure that only the edges connecting vertices of the same label are considered. In the second step, we invoke the CC algorithm on the filtered graph.

Translating that process into code requires only few lines of code, see Listing 1. We use the data structure and functions from the `boost` library. First, the code defines the predicate to facilitate graph filtering via class `edge_same_color_predicate`. That class has an operator, `operator()`, that checks whether a given edge satisfies the filtering condition and returns a `true` or a `false` value if the condition is satisfied or not. In that case, we check the condition of whether the colors of the two vertices that form the edge `e` (`source` and `target`) are the same. To retrieve the labels of vertices constituting the edge, the class additionally stores pointers to the graph `G_` and the vector of the vertices' labels (e.g., `color_`) .

Once the graph filtering is defined, it is used to filter the original graph. The function to determine the CCs in the graph is included in Listing 2. It consists of three lines: the declaration of the object `p` of type defined above, the declaration of the filtered graph `FG` of the type of `filtered_graph` defined in `boost::graph` library, and the call of function `connected_components` from `boost` library that determine connected components in the filtered graph. The outcome of that procedure is stored in the vector `components` with integer values that correspond to the index of the CCs.

Fig. 2 illustrates a simple example with the indices of CCs marked. In the figure, each vertex in the graph has an assigned index representing the CC that it belongs to. All vertices, included meta-vertices, have been assigned indices. With the CCs determined, a detailed quantification can be performed. For instance, a simple operation is to query the graph on the total number of CCs. In our example, the total number of CCs is seven, with three CCs being black (i.e., Indices 1–3), two being white (i.e., Indices 4 and 5), and two corresponding to two electrodes (i.e., Indices 0 and 6). In the next round of graph queries, we perform an additional operation that aims to determine the CCs adjacent to the bottom boundary. We first find all nearest neighbors of the blue meta-vertex, after which we create the corresponding set of CC indices in that neighborhood. In that case, the set consists of one CC index: Index 1. That CC is black, and no `WHITE` CC is directly adjacent to the `BLUE` vertex. We repeat the query for the red meta-vertex and learn that the set of CC consists of two indices: Indices 3 and 4. The CC with Index 3 is `BLACK`, whereas the CC with Index 4 is `WHITE`. For our targeted application, `BLACK` CC can be considered useful if connected to an anode (i.e., `RED` meta-vertex) and `WHITE` CC if connected to a cathode (i.e., `BLUE` meta-vertex). In that morphology, only one CC is considered to be useful: CC with Index 4. The gradual evolution of the graph query exemplifies the translation of queries into physically meaningful descriptors.

2.3.2. The shortest path in the graph

The computation of shortest paths is useful to estimate the properties related to the transport of charged and uncharged species within a BHJ morphology. In particular, such computation can be used to estimate the path lengths of excitons as they diffuse toward the donor–acceptor interface and both electrons and holes (i.e., with positive and negative charges) as they traverse the tortuous domains to a specific electrode. For those calculations, the shortest paths are first determined from meta-vertex to other vertices of interest, followed by post-processing to determine histograms of path lengths.

An example code to define a class with a predicate in order to filter the original graph is presented in Listing 3. The predicate allows for the filtration of edges that connect the `RED` vertex to all `BLACK` vertices as well as `BLACK` to `BLACK` vertices. There are several ways of defining the conditions to filter the edges. Our implementation involves checking for three conditions. For each edge defined by source and target vertices, we check two combinations of colors: `RED` and `BLACK` as well as `BLACK` and `RED`.[3]

Once the predicate to filter the graph is defined, we use it in the function that determines the shortest paths in the graph. The code of that function, included in Listing 4, begins by declaring the predicate and the corresponding filtered graph. Next, we

[3] The function also preserves all edges that connect vertices of the same color. The predicate ensures that we preserve the edges that connect `WHITE` and `WHITE` vertices. Those edges are ignored while computing the shortest paths, because the edges connecting `WHITE` and `BLACK` vertices are ignored.

Listing 2

```
void DetermineConnectedComponents(gt::graph_t* G, const std::vector<COLOR>& color,
        std::vector<int>& components){
    edge_same_color_predicate p( *G, color);
    boost::filtered_graph<gt::graph_t, edge_same_color_predicate> FG(*G, p);
    boost::connected_components(FG, &components[0]);
}
```

Listing 3

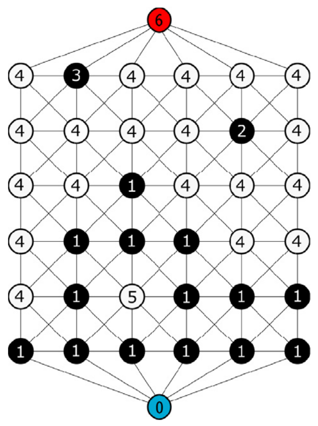
```
class edge_color_BLACK_or_RED_predicate {
public:
        edge_color_BLACK_or_RED_predicate() : G_(0), color_(0) { }
        edge_color_BLACK_or_RED_predicate(const gt::graph_t& G,
                                          const std::vector<COLOR>& color)
 : G_(&G), color_(&color) { }
 bool operator()(const gt::edge_t& e) const {
 if ( ( (*color_)[boost::source(e, *G_)] == RED )
     && ( (*color_)[boost::target(e, *G_)] == BLACK ) )
         return true;
 if ( ( (*color_)[boost::target(e, *G_)] == RED )
     && ( (*color_)[boost::source(e, *G_)] == BLACK ) )
         return true;
 return ((*color_)[boost::source(e, *G_)] == (*color_)[boost::target(e, *G_)]);
 }
private:
        const gt::graph_t* G_;
        const std::vector<COLOR>* color_;
};
```

define two vectors to store the outcome from the shortest-path calculations: p and d. Vector d stores the lengths of all single-sourced shortest paths, whereas vector p stores the indices to the parents along the shortest path. Both vectors are initialized to be used in the last line of the code snippet, the execution of Dijkstra's algorithm, used to solve the single-source shortest-path problem [23,24]. Here, we determine the shortest paths from the RED vertex to all vertices in the filtered graph. As the outcome, for each vertex we can look up the distance to the RED meta-vertex and reconstruct the shortest path by tracing the generation of all parental vertices using vector p.

The structure of the code is similar to the code in Section 2.3.1. It not only involves graph filtering via the predicate and executing the graph algorithm but also has a generic function in choosing the source in the single-source shortest-path problem. In GraSPI, that function is used for several descriptors—for instance, to find the shortest path from the GREEN vertex to all BLACK vertices, as well as from the BLUE vertex to all WHITE vertices.

We now explain how the graph query's outcome is post-processed to provide qualitative and quantitative means to compare morphologies. In Fig. 3, we depict three morphologies along with the histograms of the shortest paths from all black pixel to the interface. Those morphologies are selected due to the dramatic variation in feature size and distance to the interface. In the second row, we depict the histogram of the length of all shortest paths from the graph query. That histogram provides insight into the number of paths of varying length. From the histograms, information about the fraction of domains that are at an exceptionally short distance from the interface can be easily extracted. In the case of the first morphology, 10% of black vertices are within a 0.5-pixel distance from the interface. The other morphologies have 18.5% and 20%, respectively.

In our application in OPVs, the described query is translated to quantify the fraction of the donor (BLACK) within d distance from the interface. That information can be more seamlessly extracted from the cumulative histogram (Fig. 3, third row), which



Number of BLACK CCs : 3 (Indices: 1, 2, 3)
Number of WHITE CCs : 2 (Indices: 4 ,5)
Number of BLACK CCs conn to RED : 1 (Index: 4)
Number of WHITE CCs conn to BLUE : 0 (-)

Fig. 2. Example of a simple graph with indices of the connected components marked. Number of BLACK components (i.e., three) and WHITE components (i.e., two), number of BLACK components connected to top (one), white components connected to bottom (i.e., none).

Listing 4

```
void FindDistancesToVertexN( gt::graph_t* G, const std::vector<COLOR>& color, gt::vertex_t n ){

    edge_color_BLACK_or_RED_predicate pred( *G, color);
    boost::filtered_graph<gt::graph_t, edge_color_BLACK_or_RED_predicate> FG(*G, pred);

    std::vector<double> d(boost::num_vertices(*G));
    std::vector<gt::vertex_t> p(boost::num_vertices(*G));
    std::fill(d.begin(), d.end(), 0.0);
    for (unsigned int i = 0; i < n; ++i) p[i] = i;

    boost::dijkstra_shortest_paths(FG, top_electrode, boost::predecessor_map(&p[0]).distance_map(&d[0]));

}
```

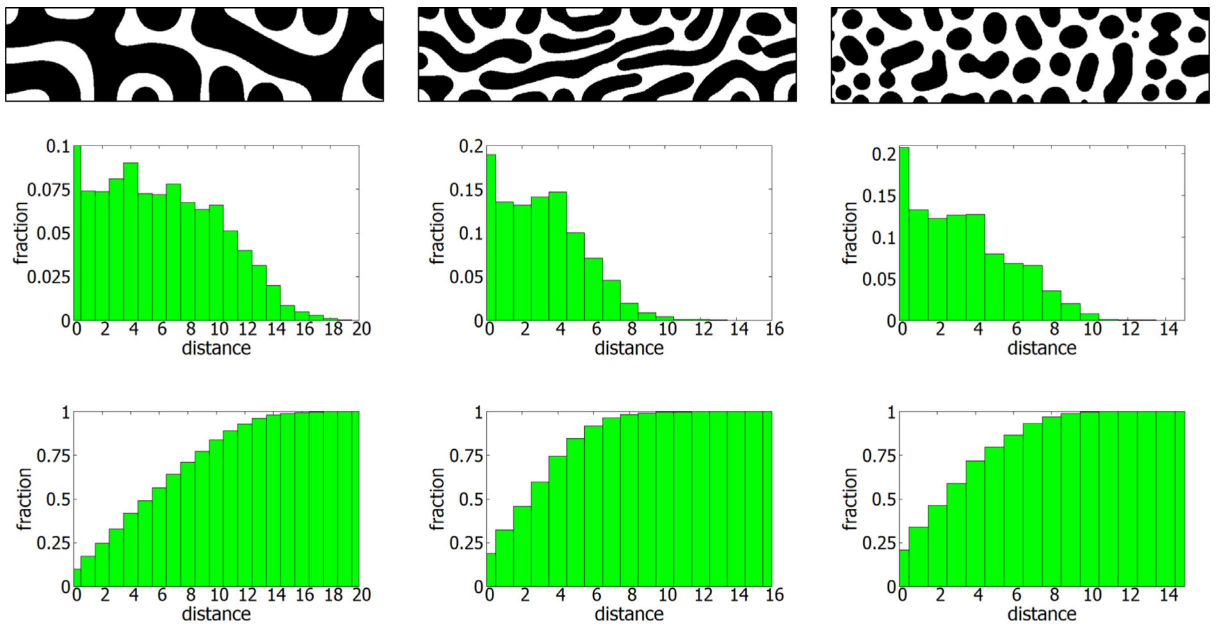


Fig. 3. Various types of information extracted from three example morphologies (top). Depicted histograms provide a detailed characterization of morphologies with respect to the distance from any vertex representing donor material (BLACK) to the interface. We plot histograms illustrating the probability mass function $P(distance = x)$ (second row) and the cumulative probability mass function $P(distance < x)$ (last row).

facilitates a physically meaningful understanding of the total fraction of electron donor material within 10 nm distance from the interface ($P(distance = x)$). Among the analyzed morphologies, that fraction is the highest for the third morphology (100%) and the lowest for the first (82%). We can also vary the distance, d, and extract the fraction of pixels within that distance from a histogram (i.e., the probability mass function) of the path lengths (i.e., distances). That method is an example of post-processing the histograms to extract physically meaningful descriptors that capture certain aspects of the targeted application.

2.4. Software architecture

GraSPI is a command line tool with settings configurable in the command line. Details about the available functionality appear in Section 2.5. GraSPI is written in C/C++ using the boost library for graph-based operation [25]. GraSPI accepts two input formats for structured and unstructured data (see Appendix for details on those formats) and a set of text files for output. The set of descriptors is outputted to the standard stream and can be used in the sequence of more than one command (i.e., in a pipeline). For example, the standard output from GraSPI can be redirected to the log file. GraSPI is supplemented by a set of tools including format converters that convert the plt format to the row-major format of the input data and post-processing tools that can be used to generate histograms of distances and generate a one-page summary of all descriptors:

- format converters: from plt format to the row-major format of the input data;
- post-processing tools that can be used to: generate histograms of distances, and to generate one page summary of all descriptors.

2.5. Software functionalities

To learn the set of GraSPI's available settings, the user is encouraged to execute GraSPI in the command line without any arguments. The usage message provides the list of parameters that can be used:

```
./graspi
GraSPI accepts input data in two formats: graph, and array.
For more information check documentation
./graspi -g <INPUT_FILE.graphe>
./graspi -a <INPUT_FILE.txt> (row-major order) -s <pixelSize> (default 1) -p <0,1> (default 0-false) -n <2,3> (default 2-D,A) -r path where store results (default ./)
```

Table 1
The list of descriptors along with the relation to the graph algorithms, the OPV performance measure, and the three steps of the photovoltaic process: light absorption (ABS), exciton dissociation (DISS), and charge transport (CT). When no direct relation with OPV performance exists, then the descriptor is labeled as statistics (STAT). The naming is consistent with the log file outputted by GraSPI. The third column lists the relation to the graph algorithms. CC = connected component algorithm, DA = Dijkstra's algorithm, GF = graph filtering.

	Descriptors	Relation to graph	Relation to OPV performance
ABS_wf_D	Weighted fraction of BLACK vertices	cardinality of V	light absorption
STAT_CC_D	Number of BLACK CCs:	CC	–
STAT_CC_A	Number of WHITE CCs	CC	–
STAT_CC_D_An	Number of BLACK CCs conn to RED	CC + GF	
STAT_CC_A_Ca	Number of WHITE CCs conn to BLUE	CC+GF	
STAT_n	Number of vertices:	cardinality of V	
STAT_n_D	Number of BLACK vertices	GF+cardinality of V	
STAT_n_A	Number of WHITE vertices	GF+cardinality of V	
ABS_f_D	Fraction of BLACK vertices	GF+cardinality of V	light absorption
CT_f_conn	Fraction of useful vertices - w/o islands:	CC+GF	charge transport
CT_f_conn_D_An	Fraction of BLACK vertices conn to RED	CC+GF	charge transport
CT_f_conn_A_Ca	Fraction of WHITE vertices conn to BLUE	CC+GF	charge transport
DISS_wf10_D	Weighted fraction of BLACK vertices in 10 distance to GREEN	GF+DA	exciton diffusion
DISS_f10_D	Fraction of BLACK vertices in 10 distance to GREEN	GF+DA	exciton diffusion
STAT_e	Number of GREEN 1st order edges	cardinality of E	exciton diffusion
CT_e_conn	Number of int edges with complementary paths	GF+CC	–
CT_f_e_conn	Fraction of interface with complementary paths to BLUE and RED	GF+CC	charge transport
CT_e_D_An	Number of BLACK int vertices with path to RED	GF+CC	–
CT_e_A_Ca	Number of WHITE int vertices with path to BLUE	GF+CC	–
CT_f_D_tort1	Fraction of BLACK vertices with straight rising paths (t=1)	GF+DA	charge transport
CT_f_A_tort1	Fraction of WHITE vertices with straight rising paths (t=1)	GF+DA	charge transport
CT_n_D_adj_An	Number of BLACK vertices in direct contact with RED vertex	GF+DA	charge transport
CT_n_A_adj_Ca	Number of WHITE vertices in direct contact with BLUE vertex	GF+DA	charge transport

`GraSPI` requires one mandatory input parameter: the name of the input file with the corresonding argument, (-a or -g and <INPUT_FILE>). The remaining parameters are optional and have the default value if the parameter is not explicitly provided.

- `-a` `<INPUT_FILE.txt>` (row-major order) is the option to input information about structured data. With that assumption, the neighborhood of each voxel or pixel can be determined as the graph is constructed.
- `-g` `<INPUT_FILE.graphe>` is the option to input information about unstructured data. The input file has to provide all information about the graph, meaning that the neighborhood of each vertex in the graph needs to be determined externally. Meta-vertices and the associated edges need to be defined in the input file. When this option is used, `GraSPI` reads the text file and initializes the set of vertices and edges from the input file, which need to be in agreement with those defined in the package for a given usage case.
- `-s` `<pixelSize>` (default = 1) is the option that sets the size of the pixel to compute the length of the shortest paths. If unspecified, then all results will be outputted in terms of the number of pixels and need to be rescaled for dimensional analysis.
- `-p` `<0,1>` (default = 0, meaning false) is the option that specifies whether periodicity on the side faces is to be applied, which is valid only for morphology inputted as the array option (i.e., `-a`).
- `-n` `<2,3>` (default = 2, i.e., black and white, electron-donor and electron accepting material) is the option for specifying the number of phases. For three-phase morphology (option `-n` 3), black, white, and grey vertices are read, which correspond to electron donor, electron acceptor, and mixed phase material, respectively.
- `-r` `path` is the option that specifies where results are stored (default ./)—that is, where text files with results will be saved.

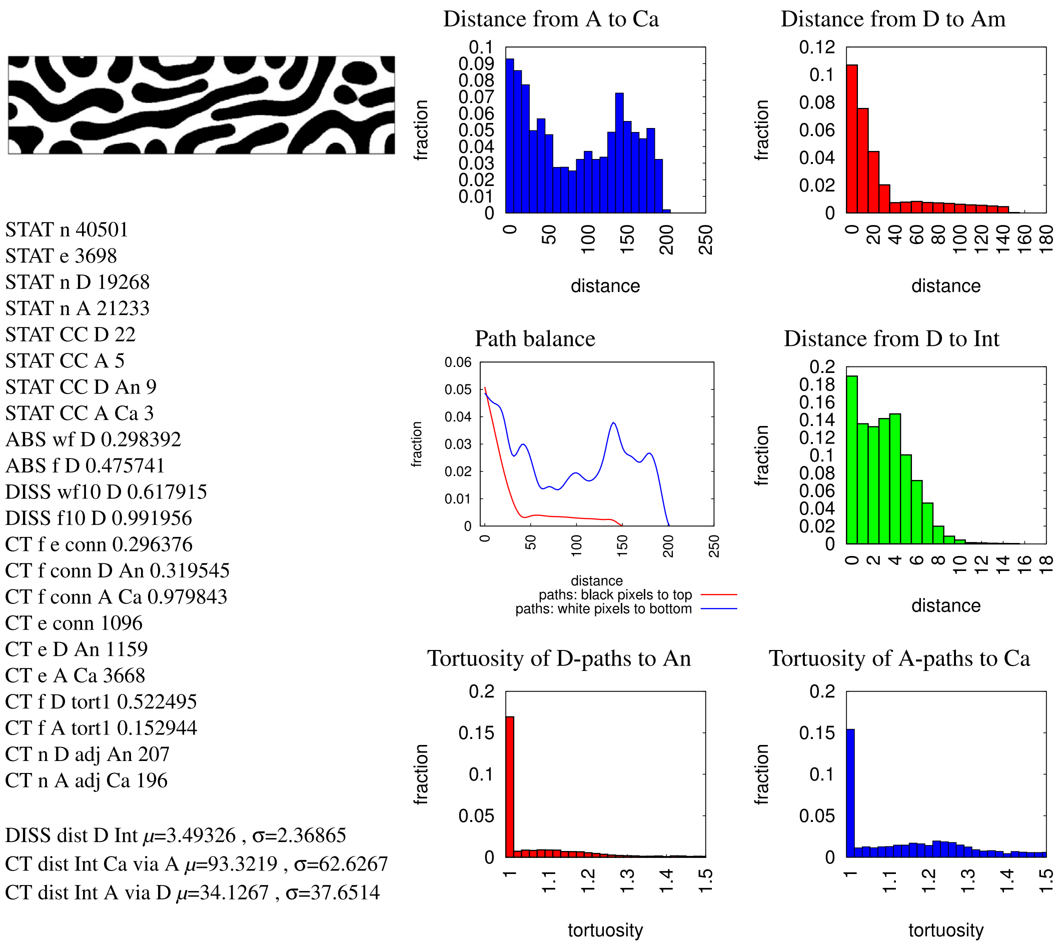
`GraSPI` computes two types of descriptors: scalar descriptors and array descriptors. Scalar descriptors are directed to the standard output,[4] whereas array descriptors are directed to the corresponding file. The list of scalar descriptors along with their graph relationships is summarized in Table 1. The array descriptors correspond to the shortest distances and are saved in the files DistancesGreenToRedViaBlack.txt, DistancesGreenToBlueViaWhite.txt, DistancesBlackToRed.txt, DistancesWhiteToBlue.txt, DistancesBlackToGreen.txt, TortuosityBlackToRed.txt, and TortuosityWhiteToBlue.txt. The name of the file indicates the conditions used to filter the graph. For example, the file DistancesBlackToGreen.txt stores all of the shortest distances between any donor, black, and 0 voxel and the green interface. If there is no direct connection between the source vertex and the target vertex, then the distance is set to infinity. This is how `boost` library is initializing the distance vector. The distances are saved in the order of the labels from the input file, and the infinity distances are maintained in order to maintain the capability to map back the distances to the input voxels.

3. Illustrative examples of morphology annotation through graph-queries

In this section, we present an example morphology annotation with the set of descriptors in the form of a one-page summary (Fig. 4) generated using `GraSPI` with `bash`, a command line tool for processing text (e.g., `grep` and `sed`); `gnuplot`, which allows rapid visualization; and `latex`, which allows combining various data sources. The scripts are included in the folder with examples.

The left column in the summary includes the figure with the morphology, 22 scalar descriptors computed in `GraSPI`, and three additional descriptors post-processed from the distance array. The right column in the summary depicts six histograms of distances for the quick visual display of the characteristics of the pathways. For one type of pathway, A-path to Ca (i.e., acceptor pathways toward the cathode), three types of information are displayed: the histogram of the path lengths, the tortuosity of the pathways, and the pathway balance (i.e., between donor and acceptor pathways).

[4] Standard output `std::cout`.



```
STAT n 40501
STAT e 3698
STAT n D 19268
STAT n A 21233
STAT CC D 22
STAT CC A 5
STAT CC D An 9
STAT CC A Ca 3
ABS wf D 0.298392
ABS f D 0.475741
DISS wf10 D 0.617915
DISS f10 D 0.991956
CT f e conn 0.296376
CT f conn D An 0.319545
CT f conn A Ca 0.979843
CT e conn 1096
CT e D An 1159
CT e A Ca 3668
CT f D tort1 0.522495
CT f A tort1 0.152944
CT n D adj An 207
CT n A adj Ca 196

DISS dist D Int μ=3.49326 , σ=2.36865
CT dist Int Ca via A μ=93.3219 , σ=62.6267
CT dist Int A via D μ=34.1267 , σ=37.6514
```

Distance from A to Ca
Distance from D to Am
Path balance
Distance from D to Int
Tortuosity of D-paths to An
Tortuosity of A-paths to Ca
fraction
distance
tortuosity
paths: black pixels to top
paths: white pixels to bottom

Fig. 4. Example one page summary of descriptors and histograms generated by `GraSPI`.

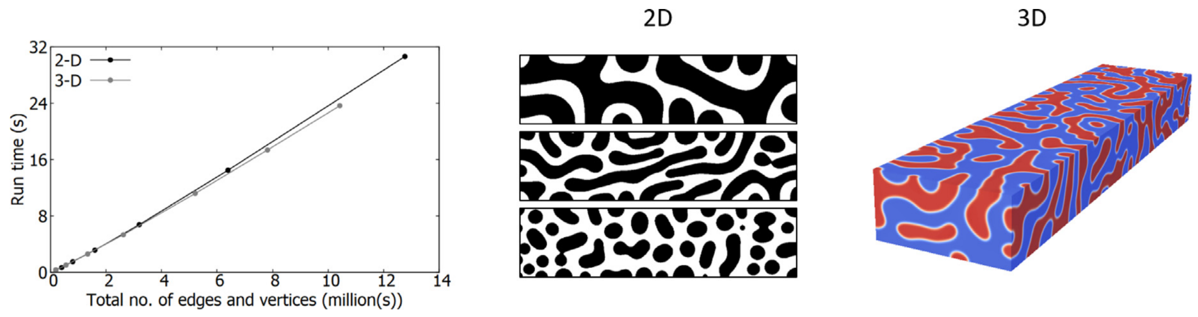


Fig. 5. The runtime of `GraSPI` for 2D and 3D morphologies with increasingly large graphs in terms of the number of vertices and edges. For both 2D and 3D morphologies, the runtime is in seconds. The largest morphologies analyzed consisted of 2.5 million vertices, or 13 million edges and vertices, on a desktop computer with a 2.3-GHz dual-core Intel Core i5 processor and 8 GB of RAM.

4. Computational complexity

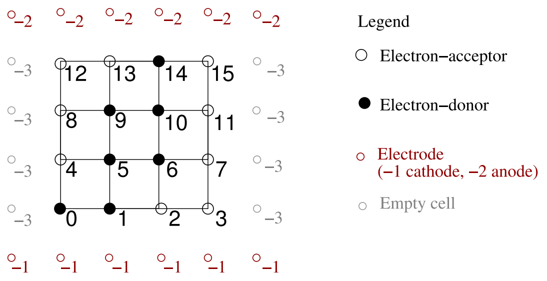
The complete framework discussed thus far is implemented in C++ using the `boost` library for graph-based operations [25]. We performed a set of experiments to demonstrate the computational cost associated with the descriptor calculations. Fig. 5 reports the total runtime (in seconds) for all of the operations involved in descriptor calculations. The reported runtime includes the reading of data, graph filtration, the determination of the CCs and shortest distances, and the computation of the descriptors (Table 1).

The tests were executed on two types of data of increasing size. We used 2D and 3D periodic structures which were generated using Cahn–Hillard equations for thin-film geometries [26]. The representative morphologies of thin films are included in the second and third columns of Fig. 5. To gradually increase the size of the morphologies, the basic morphology was replicated gradually: vertically in 2D and horizontally in 3D.[5]

For 2D analysis, three morphologies of varying domain sizes and geometries were considered. We selected those structures to test the performance with different types of morphologies for a range of CCs and shortest distances. The quantities are plotted for the topmost morphology in the second column in Fig. 5.[6]

[5] The basic morphologies are periodic, meaning that the replication ensures the continuity of the morphologies.

[6] We also replicated the morphologies horizontally (not reported here) but did not notice any significant difference.



```
4 4 1
1 1 0 0
0 1 1 0
0 1 1 0
0 0 1 0
```

```
16
0 1 4 1 f 5 √2 s 1 1 f -1 1 f
1 1 5 1 f 6 √2 s 2 1 f -1 1 f 0 1 f 4 √2 s
2 0 6 1 f 7 √2 s 3 1 f -1 1 f 1 1 f 5 √2 s
...
14 1 -2 1 f 15 1 f 11 √2 s 10 1 f 9 √2 s 13 1 f
15 0 -2 1 f 11 1 f 10 √2 s 14 1 f
16 0 0 s 1 0 s 2 0 s 3 0 s
17 15 0 s 14 0 s 13 0 s 12 0 s
```

Fig. A.6. Simple example of structured phase information and the two formats: array and graph-based.

The smallest 2D or 3D sample explored was a 100×400 micrograph that corresponds to a total of $\approx 40,000$ vertices.[7] The sample size was doubled for every test, with the largest sample sizes exhibiting approximately 2.56 million vertices in 2D and 1.6 million in 3D, which correspond to a total of approximately 12 million vertices and edges combined. The upper range of the problem size is limited due to RAM limitations. For samples with more than 3 million vertices, the maximum RAM (8 GB) limit was reached on the machine used.

In Fig. 5, we plot the time taken to evaluate those two types of morphology. We plot data for the 2D and 3D structures as a function of problem size (i.e., to the total of edges and vertices) using black and grey lines and points, respectively. For 2D, the runtime increases from 2 s for the smallest sample and to 32 s for the largest. Moreover, the runtimes for 3D structures with sizes similar to the graphs of the 2D structures are comparable.

For the range of the morphology sizes examined, the runtime is observed to be nearly linear. That agrees with complexity theory, which holds that Dijkstra's algorithm and the CCs have a time complexity of $\mathcal{O}(V \log(V))$ and $\mathcal{O}(V + E)$, respectively, in which V is the number of vertices and E is the number of edges of the input graph. The computational complexities of the graph-based algorithms depend on the size of the data (i.e., the number of vertices and edges). The analysis was repeated five times for each sample size, and the values were within 2% of the mean runtime value (not shown here).

In summary, our analysis demonstrates that in the problem range (i.e., of up to 12 million $(V + E)$, 2.5 million vertices in 2D, and 10 million $(V + E)$ that correspond to 1.6 million vertices in 3D), the total execution time is less than 1 min, which creates opportunities to run real-time analysis and high-throughput exploration. Moreover, the execution time is insensitive to the dimensionality (i.e., 2D vs. 3D). That quality is valuable for using graph-based algorithms to evaluate 3D structures, because the complexities depend only on the size of the graph.

5. Impact

`GraSPI` has already been instrumental in advancing current understandings of PSP maps in OPVs. Furthermore, it can be used to democratize the ability of non-computer-savvy domain scientists to rapidly and nearly in real time extract a wide suite of microstructure descriptors. `GraSPI` has previously been used to build a surrogate model of performance in OPVs [11] and microstructure optimization for OPVs [17,27], as well as to create a corpus of data for analysis [10–16]. We have applied the approach to quantify two- and three-phase morphologies and extended it to point cloud data for the analysis of molecular dynamics simulations [20]. We also expanded it into a multiscale approach for SP mapping [19]. We anticipate that GraSPI can contribute to various fields of research, including biomaterials and ceramics at both the molecular and continuum scales.

[7] To match that dimension in 3D, we took 2D morphology with thickness of 1 voxel in the third dimension.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by National Science Foundation, USA (1906344 and 1906194, in part by OAC-1845840). BG acknowledges support from the ONR, USA MURI program. OW acknowledges the support provided by the Center for Computational Research at the University at Buffalo, USA.

Appendix. Two input formats

`GraSPI` accepts two input formats for structured and unstructured data. The structured data assumes that the size of discrete points in each dimension is known. The dimensionality is provided in the first line of the input file as n_x, n_y, n_z. The reminder of the input file contains the array (i.e., row-wise representation of the corresponding matrix) with the corresponding phases from the set of available labels. Once the data are structured, the local neighborhood can be constructed as the data are read by `GraSPI`.

In the second format, corresponding to unstructured data, information about phases needs to be provided along with the neighborhood. Here, we use the adjacency matrix to inform `GraSPI` about the neighborhood. The input file starts with the total number of vertices in the dataset. Next, each line provides the information about each vortex: the index of the vertex, followed by the color of the vertex. The remainder of the line consists of triplets with information about the neighbors. Each triplet consists of the index of the neighbor, the distance from the neighbor, and the type of neighbor. In our example, three types of neighbors are considered: first-, second-, and third-order neighbors, demarcated as "f", "s", and "t". The set of types may be redefined to encode additional information (see Fig. A.6).

References

[1] Cima MJ. Next-generation wearable electronics. Nature Biotechnol 2014;32(7):642–3.
[2] He X-P, Hu X-L, James TD, Yoon J, Tian H. Multiplexed photoluminescent sensors: towards improved disease diagnostics. Chem Soc Rev 2017;46(22):6687–96.
[3] Malliaras GG. Organic bioelectronics: a new era for organic electronics. Biochim Biophys Acta 2013;1830(9):4286–7.
[4] Peng Z, Jiang K, Qin Y, Li M, Balar N, O'Connor BT, et al. Modulation of morphological, mechanical, and photovoltaic properties of ternary organic photovoltaic blends for optimum operation. Adv Energy Mater 2021;11(8):2003506.

[5] Spurgeon SR, Ophus C, Jones L, Petford-Long A, Kalinin SV, Olszta MJ, et al. Towards data-driven next-generation transmission electron microscopy. Nature Mater 2020;1–6.

[6] de Pablo JJ, Jackson NE, Webb MA, Chen L-Q, Moore JE, Morgan D, et al. New frontiers for the materials genome initiative. Npj Comput Mater 2019;5(1):1–23.

[7] Wodo O, Ganapathysubramanian B. How do evaporating thin films evolve? Unravelling phase-separation mechanisms during solvent-based fabrication of polymer blends. Appl Phys Lett 2014;105(15):153104.

[8] Hoppe H, Sariciftci N. Organic solar cells: An overview. J Mater Res 2004;19:1924–45.

[9] Brabec C, Scherf U, Dyakonov V. Organic photovoltaics: materials, device physics, and manufacturing technologies. John Wiley & Sons; 2014.

[10] Kipp D, Wodo O, Ganapathysubramanian B, Ganesan V. Utilizing morphological correlators for device performance to optimize ternary blend organic solar cells based on block copolymer additives. Sol Energy Mater Sol Cells 2017;161:206–18.

[11] Wodo O, Zola J, Pokuri BSS, Du P, Ganapathysubramanian B. Automated, high throughput exploration of process–structure–property relationships using the mapreduce paradigm. Mater Discov 2015;1:21–8.

[12] Gebhardt RS, Du P, Wodo O, Ganapathysubramanian B. A data-driven identification of morphological features influencing the fill factor and efficiency of organic photovoltaic devices. Comput Mater Sci 2017;129:220–5.

[13] Pokuri BSS, Sit J, Wodo O, Baran D, Ameri T, Brabec CJ, et al. Nanoscale morphology of doctor bladed versus spin-coated organic photovoltaic films. Adv Energy Mater 2017;7(22):1701269.

[14] Pfeifer S, Wodo O, Ganapathysubramanian B. An optimization approach to identify processing pathways for achieving tailored thin film morphologies. Comput Mater Sci 2018;143:486–96.

[15] Hickey RT, Jedlicka E, Pokuri BSS, Colbert AE, Bedolla-Valdez ZI, Ganapathysubramanian B, et al. Morphological consequences of ligand exchange in quantum dot-polymer solar cells. Org Electron 2018;54:119–25.

[16] Pfeifer S, Pokuri BSS, Du P, Ganapathysubramanian B. Process optimization for microstructure-dependent properties in thin film organic electronics. Mater Discov 2018;11:6–13.

[17] Du P, Zebrowski A, Zola J, Ganapathysubramanian B, Wodo O. Microstructure design using graphs. Npj Comput Mater 2018;4(1): 1–7.

[18] Wodo O, Roehling JD, Moulé AJ, Ganapathysubramanian B. Quantifying organic solar cell morphology: a computational study of three-dimensional maps. Energy Environ Sci 2013;6(10):3060–70.

[19] Van E, Jones M, Jankowski E, Wodo O. Using graphs to quantify energetic and structural order in semicrystalline oligothiophene thin films. Mol Syst Des Eng 2018;3(5):853–67.

[20] Lee C-K, Wodo O, Ganapathysubramanian B, Pao C-W. Electrode materials, thermal annealing sequences, and lateral/vertical phase separation of polymer solar cells from multiscale molecular simulations. ACS Appl Mater Interfaces 2014;6(23):20612–24.

[21] Langner S, Häse F, Perea JD, Stubhan T, Hauch J, Roch LM, et al. Beyond ternary OPV: high-throughput experimentation and self-driving laboratories optimize multicomponent systems. Adv Mater 2020;32(14):1907801.

[22] Bi Z, Zhu Q, Xu X, Naveed HB, Sui X, Xin J, et al. Efficient quaternary organic solar cells with parallel-alloy morphology. Adv Funct Mater 2019;29(9):1806804.

[23] Cormen T, Leiserson C, Rivest R, Stein C. Book: introduction to algorithms. MIT Press; 2009.

[24] Dijkstra EW, et al. A note on two problems in connexion with graphs. Numer Math 1959;1(1):269–71.

[25] Siek J, Lee L-Q, Lumsdaine A. Boost library. 2000, http://www.boost.org/libs/graph/.

[26] Wodo O, Ganapathysubramanian B. Computationally efficient solution to the cahn–hilliard equation: Adaptive implicit time schemes, mesh sensitivity analysis and the 3D isoperimetric problem. J Comput Phys 2011;230(15):6037–60.

[27] Lee XY, Waite JR, Yang C-H, Pokuri BSS, Joshi A, Balu A, et al. Fast inverse design of microstructures via generative invariance networks. Nat Comput Sci 2021;1(3):229–38.