**IEEE Robotics and Automation Letters (RAL) paper presented at the**
**2020 IEEE International Conference on Robotics and Automation (ICRA)**
**31 May - 31 August, 2020. Paris, France**

# Self-Reconfiguration in Response to Faults in Modular Aerial Systems

Neeraj Gandhi[1], David Saldaña[2], Vijay Kumar[3], Linh Thi Xuan Phan[1]

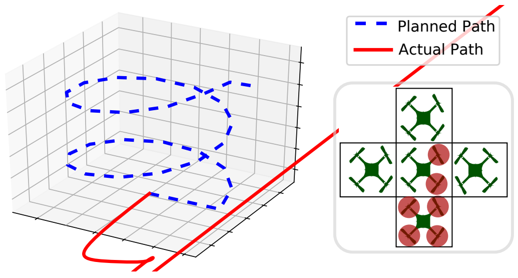
***Abstract*— We present a self-reconfiguration technique by which a modular flying platform can mitigate the impact of rotor failures. In this technique, the system adapts its configuration in response to rotor failures to be able to continue its mission while efficiently utilizing resources. A mixed integer linear program determines an optimal module-to-position allocation in the structure based on rotor faults and desired trajectories. We further propose an efficient dynamic programming algorithm that minimizes the number of disassembly and reassembly steps needed for reconfiguration. Evaluation results show that our technique can substantially increase the robustness of the system while utilizing resources efficiently, and that it can scale well with the number of modules.**

## I. Introduction

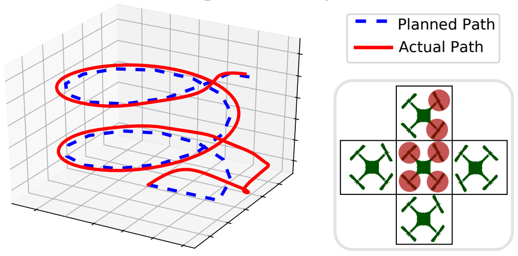
Termites, bees, and ants are prominent examples of many small units working together to accomplish large undertakings (e.g., a termite mound). Recently, there has been a trend towards replicating such swarm-like behavior in robotic systems to solve complex collective tasks such as exploration [21], construction [13], and transportation [4] [8]. For example, robot swarms have been used to create large structures that are impossible for a single agent to accomplish [1], [6], [19].

Using robot bodies as building units, individual agents in the swarm can attach to one another to form complex structures such as bridges or towers [12], [13], [16]. The versatility such systems can provide is vast – they can change their configuration to suit a wide range of tasks [17]. In the robotics literature, there exist aerial modular systems that can be manually assembled [3], [10], [20], [22], systems that self-assemble on the ground [11], or even ones that self-assemble in midair [13]. Despite having inherent redundancy in actuation, these systems are not fault tolerant by default: modules are often tightly coupled, and thus the failure of a small number of modules can substantially impact the overall system performance (see Fig. 1 for an example scenario).

Rotor failures have been investigated in multirotor systems. For instance, the work in [9] examines rotor failures for



(a) Non-optimal configuration.



(b) After self-reconfiguration.

Fig. 1: An aerial structure in a plus-shape following a helical trajectory. Illustrations of the structures are shown to the right of each plot. Translucent red disks indicate faulty rotors. (a) Six faulty rotors can be fatal for a structure driving it to undesired locations. (b) The structure can follow the trajectory after self-reconfiguration.

quadrotors. Since a quadrotor does not provide redundancy, it cannot follow a trajectory without undesirable rotations. The authors of [18] studied how thrust can be redistributed in an octorotor when a rotor fails such that the aircraft can continue flying. The approach described in [2] explores a fault-detection method inspired by the flashing lights of fireflies to detect and repair faults in a timely manner. However, existing work is limited to only single-module systems.

This paper takes advantage of the inherent redundancy modular aerial systems can provide to propose self-reconfiguration techniques for cooperative multi-agent systems to recover from rotor failures. It focuses on modular aerial robots, specifically ModQuad [13]; however, our approach is general and can be adapted to other robotic systems. ModQuad is a modular aerial system in which each module is a cuboid propelled by a quadrotor. Modules are bound together with permanent magnets at the corners of their frames to form a single rigid body. With this mechanism, a ModQuad swarm can rapidly (dis)assemble flying structures in midair, which is particularly useful in time-critical situations, such as in a disaster-relief missions.

[1]Neeraj Gandhi and Linh Thi Xuan Phan are with the PRECISE Center and Distributed Systems Laboratory, University of Pennsylvania, Philadelphia, PA, USA. {ngandhi3, linhphan}@seas.upenn.edu

[2]David Saldaña is with the Autonomous and Intelligent Robotics Laboratory (AIRLab), Lehigh University, Bethlehem, PA, USA. saldana@lehigh.edu

[3]Vijay Kumar is with the GRASP Laboratory, University of Pennsylvania, Philadelphia, PA, USA. kumar@seas.upenn.edu

The authors gratefully acknowledge the support of the NSF IIS-1426840, 1138847, CNS-1703936, CNS-1563873 and CNS-1750158, ONR N00014-15-1-2115, N00014-14-1-0510 and N00014-20-1-2744, ARL W911NF-08-2-0004, and DARPA HR00111520020, HR0011-16-C-0056 and HR0011-17-C-0047.

Previous work has shown that a system like ModQuad is physically feasible using a small quadrotor platform like Crazyflie 2.0 robots [14] [13] [4], or by using custom-built larger quadrotors [5]. While ModQuad might be far from being applied commercially, these initial papers show that the system is physically possible and has the potential to be applied in real-world settings.

Traditionally, multirotor vehicles handle faults by having more than four rotors. However, the maximum number of faults is fixed and limited by the number of rotors. For instance, a hexarotor might be able to handle two rotor failures, and perhaps stay aloft with four failures, but it cannot handle more. A multi-robot system such as ModQuad allows for the structure itself to expand, thereby increasing the number of faults that can be handled. If we wish for a system to withstand more faults, we can simply append an additional module to the structure.

**Motivating scenario:** Let us consider a fault scenario in a ModQuad structure consisting of 5 modules arranged in a plus-sign shape. The structure needs to travel along a helical trajectory, as shown in Fig. 1. Multiple rotor failures in critical locations can render the system to a nonfunctional state, as depicted in Fig. 1(a). Through optimal self-reconfiguration (using our proposed technique), the system is able to adapt to the faulty rotors and complete its task. As illustrated in Fig. 1(b), the actual trajectory of the system closely follows the planned trajectory despite having six faulty rotors. Self-reconfiguration can thus be effective in minimizing the impact of faults on performance, and it may even be necessary for mission completion.

**Challenges:** To make ModQuad adaptable to faults, we need to address two research questions: *i)* how should the modules be allocated to the structure to minimize the impact of the faulty rotors? and *ii)* how should a structure change its existing configuration to a new configuration such that faulty rotors are properly handled?

Finding an optimal new configuration for the modules is highly non-trivial, as it depends on not only the positions of the faulty rotors but also the current path of the structure. To illustrate this, consider the structure in Fig. 2, where each square represents a module, and Module 1 experiences a rotor failure (marked by a red disk). This module should be positioned differently in the new configuration based on the desired path to minimize path divergence. For instance, for Path 1, Module 1 should be moved to the current position of either Module 4 or Module 5. For Path 2, Module 1 can remain in its current position. Finally, for Path 3, it should be placed in the current position of either Module 2 or Module 7.

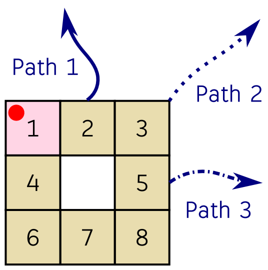


Fig. 2: Factors affecting optimal configurations.

Further, it is neither always feasible nor always desirable to move to the new configuration by disconnecting all modules from one another and then re-assembling. As illustrated in Fig. 3, for any given path, the impact of faults in Module 1 can be mitigated by moving Module 1 to the position of Module 3. However, achieving *Goal Structure 1* is infeasible, as it requires Module 1 to be disconnected from the rest (after which we will not be able to move it into a docking position). In contrast, *Goal Structure 2* is achievable because the breaks we introduce in the original structure do not isolate any module with faulty rotors.

Our technique addresses these challenges using three goals when computing new configurations and reconfiguration strategy: *i)* minimize the impact of the faulty rotors on the motion of the structure *over the entire trajectory*, *ii)* ensure all modules in the structure complete the trajectory by making sure faulty modules are never isolated in the migration between configurations, and *iii)* minimize the number of disassembly and reassembly steps during reconfiguration, as each step incurs non-negligible time and energy overhead. We assume that such computation is done offline and stored in a *configuration tree*. That is, given some sequence of faults that occur in the system, the robot can look through the configuration tree and find the specific new reconfiguration it should self-reconfigure to given the specific sequence of faults that occurred.

## II. Model

A ModQuad module has a cuboid shape and is propelled by a single quadrotor. Groups of modules can join their vertical faces ($yz$,$xz$ planes) to form a structure. A structure is represented as a graph $\mathcal{S} = (\mathcal{V}, \mathcal{E})$, where vertices are individual rotors and edges are connections between adjacent modules. A structure is formed when two individual modules or structures horizontally dock to one another. Similarly, a structure can be disassembled through an undocking action, in which two sides of the structure magnetically disconnect from one another [15]. For ease of discussion, we can describe the structure as being $m$ modules long and $n$ modules wide, though it is not necessary that every position in this $m \times n$ rectangular space is filled with a physical module.

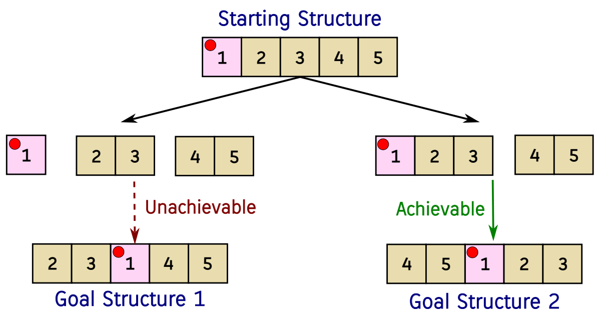


Fig. 3: A structure with two self-reconfiguration sequences.

We refer to a module by its index and denote the set of $N$ available modules by $\mathcal{M} = \{1,...,N\}$. Each module $i$ has four vertical rotors in a square configuration that generates vertical forces and moments

$$f_{ik} = k_F\, \omega_{ik}^2, \qquad M_{ik} = \pm k_M\, \omega_{ik}^2, \qquad \text{for } k = 1,...,4,$$

where $\omega_{ik}$ is the angular speed of the rotor, and $k_F$ and $k_M$ are motor constants that can be obtained experimentally. The inertial, or world, coordinate frame is denoted by $W$, the structure coordinate frame by $S$, and the module coordinate frame for the $i^{th}$ module position in $\mathcal{S}$ by $R_i$. We assume all modules are identically oriented, i.e. the $x$-axis of the structure coordinate frame is parallel to the $x$-axis of each individual module (as with the other axes).

The location of the center of mass of the $i^{th}$ module in the desired structure is $\mathbf{x}_i^S = [x_i, y_i, z_i]^\top$. The location of the $k^{th}$ rotor of the $i^{th}$ module is $\mathbf{x}_{i,k}^S = [x_{i,k}, y_{i,k}, z_{i,k}]^\top$. The total thrust $F$ and total moments $\mathbf{M} = [M_x, M_y, M_z]^\top$ are a function of all individual rotor forces in the structure:

$$\begin{bmatrix} F \\ M_x \\ M_y \\ M_z \end{bmatrix} = \sum_i \begin{bmatrix} 1 & 1 & 1 & 1 \\ y_{i,1} & y_{i,2} & y_{i,3} & y_{i,4} \\ -x_{i,1} & -x_{i,2} & -x_{i,3} & -x_{i,4} \\ \frac{k_M}{k_F} & -\frac{k_M}{k_F} & \frac{k_M}{k_F} & -\frac{k_M}{k_F} \end{bmatrix} \begin{bmatrix} f_{i,1} \\ f_{i,2} \\ f_{i,3} \\ f_{i,4} \end{bmatrix}.$$

The resultant force and moments generate translational and rotational accelerations for the entire structure, denoted $\ddot{\mathbf{x}}^S$ and $\dot{\Omega}$, respectively (see [13] for more details). Additional notation we use is provided in Table I.

## III. Finding Optimal Allocation

The optimality of module-to-position allocation in a structure is related to the planned trajectory. Given a set of modules $\mathcal{M}$, a desired structure $\mathcal{S}$, and a rotor state matrix $\Gamma$, we use a mixed integer linear program (MILP) to find an optimal module placement.

### A. MILP Constraints

Our MILP formulation aims to find an optimal allocation $\Pi^*$ that assigns modules to positions in the structure such that faulty-rotor impact on performance is minimized. Towards this, it makes use of the following constraints.

1) Each location can only contain a single module and each module can only be placed in a single location:

$$\Pi \mathbf{1} = \mathbf{1} \qquad \text{and} \qquad \Pi^\top \mathbf{1} = \mathbf{1}.$$

2) The values of the adjacency matrix $\mathbf{A}$ must satisfy the condition that $A_{j,j',d} = 1$ if and only if $j$ and $j'$ are actually adjacent in the generated allocation:

$$A_{j,j',d} = \max_{i,i' \in \mathcal{S}} \left\{ P_{i,i',d} \left( \Pi_{i,j} + \Pi_{i',j'} - 1 \right), 0 \right\}$$
$$\forall j, j' \in \mathcal{M}, d \in \{0,1\}.$$

If modules $j, j'$ are assigned to positions $i, i'$, respectively, then $A_{j,j',d}$ acquires value 1 if and only if the $i^{th}$ position in the structure is adjacent in direction $d$ to the $i'^{th}$ position in the structure.

3) Optionally, we can also constrain specific modules to be placed next to one another. This is useful when a module has a faulty rotor and thus cannot be undocked from all other modules (c.f. Fig. 3). For each tuple $(\mathcal{S}^p, \Pi^p)$ passed as an input to the MILP, where $\mathcal{S}^p$ represents a structure and $\Pi^p$ is the module-to-position allocation matrix of $\mathcal{S}^p$, we constrain the adjacency matrix of the MILP, $\mathbf{A}$, to contain all the adjacencies between modules in in $(\mathcal{S}^p, \Pi^p)$. That is, we require $A_{j,j',d} = 1$ if, for any tuple $(\mathcal{S}^p, \Pi^p)$ passed in, $adj((\mathcal{S}^p, \Pi^p), j, j', d) = 1$. Here, the function *adj* specifies whether the modules $j, j'$ are adjacent to each other in direction $d$ in $(\mathcal{S}^p, \Pi^p)$.

TABLE I: Symbols used in our approach.

| Symbol | Description |
|---|---|
| $\ell$ | The side length of a cuboid module. |
| $\mathcal{R}$ | The set of rotors in a single module. $\mathcal{R} = \{1,2,3,4\}$ for quadrotors, where 1 indicates the top-right rotor and indices increase in a clockwise manner. |
| $\mathsf{m}_i$ | Mass of module $i$. |
| $\mathsf{m}_\mathcal{S}$ | Mass of the structure computed as $\sum_i^N \mathsf{m}_i$. |
| $\Gamma$ | $\Gamma \in [0,1]^{N \times 4}$ represents the state of each rotor of each module in $\mathcal{M}$. Each individual value $\Gamma_{j,k}$ indicates the functional degree of the $k^{th}$ rotor of module $j$ in $\mathcal{M}$ with 1 for fully-functional and 0 for non-functional. Values strictly between 0 and 1 indicate partially functional rotors. |
| $\Pi$ | The allocation matrix $\Pi \in \{0,1\}^{N \times N}$. $\Pi_{i,j} \in \{0,1\}$ defines whether the physical module $j$ is assigned to the $i^{th}$ module position. |
| $\mathbf{P}$ | $\mathbf{P} \in \{0,1\}^{N \times N \times 2}$ indicates adjacencies between positions in the structure, where $P_{i,i',d} = 1$ indicates that the position $\mathbf{x}_i^\mathcal{S}$ is adjacent to $\mathbf{x}_{i'}^\mathcal{S}$ in the direction $d$. A rightward adjacency is $d = 0$, and a downward adjacency is $d = 1$ (the other two directions are redundant). |
| $\mathbf{A}$ | $\mathbf{A} \in \{0,1\}^{N \times N \times 2}$ indicates adjacencies between physical modules. We can write the adjacency for two modules $j, j'$ as $A_{j,j',d}$, where the direction $d$ is the same as for $\mathbf{P}$. Unlike $\mathbf{P}$, this matrix establishes if a pair of physical modules (e.g., Module 1 and Module 2) are adjacent in direction $d$; it is agnostic to the actual positions the two modules are assigned to. |

### B. Objective Function

We plan minimum snap trajectories for structures to follow as described in [7]. To optimize the allocation of rotors to positions, we need a metric for optimality. We maximize the moment produced about a weighted average of the $x$ and $y$ axes based on the trajectory.

We can directly control angular acceleration in our system model, which is proportional to the linear snap (forth derivative of position). The trajectory planner gives us several parameters, among which are the snap in the $x$ and $y$ dimensions, denoted $x^{(4)}$ and $y^{(4)}$. We find the constants

$$b_x = \int_0^{t_f} |x^{(4)}(t)|dt \quad \text{and} \quad b_y = \int_0^{t_f} |y^{(4)}(t)|dt,$$

where $t_f$ is the planned time to complete the trajectory.

Next, we define the moments about the $x$ and $y$ axes of the structure as a function of the allocation of modules to positions $\Pi$:

$$\overline{M}_x(\Pi,\Gamma) = \mathsf{f}_{\mathsf{max}} \sum_{\substack{i\in\mathcal{S}, j\in\mathcal{M},\\ k\in\mathcal{K}}} \Pi_{i,j}\Gamma_{j,k} \left|x^{\mathcal{S}}_{i,k}\right|, \text{ and}$$

$$\overline{M}_y(\Pi,\Gamma) = \mathsf{f}_{\mathsf{max}} \sum_{\substack{i\in\mathcal{S}, j\in\mathcal{M},\\ k\in\mathcal{K}}} \Pi_{i,j}\Gamma_{j,k} \left|y^{\mathcal{S}}_{i,k}\right|.$$

We multiply the rotor positions by $\Pi_{i,j}$ to ensure that the moment is nonzero only when the module $j$ is actually assigned to the $i^{th}$ module position, and we also multiply by $\Gamma_{j,k}$ to ensure that the maximum moment is capped based on how faulty each rotor is. Finally, we multiply the entire expression by $\mathsf{f}_{\mathsf{max}}$ to compute the moment as a function of the module-to-position allocation.

The two moments are combined to produce an objective function that uses the module allocation $\Pi$ to maximize the average moment in the $x$– and $y$–axes of the trajectory

$$\phi = \max_{\Pi} \frac{b_x}{b_x+b_y}\overline{M}_x + \frac{b_y}{b_x+b_y}\overline{M}_y.$$

The MILP returns an optimal allocation $\Pi^*$ for a given structure. If the structure is already in some arbitrary configuration $\Pi$, the system must know the sequence of steps to self-reconfigure from $\Pi$ to $\Pi^*$ such that reconfiguring is physically feasible.

## IV. Self-Reconfiguration

This section examines the problem of determining the intermediate configuration(s) a structure must acquire to reach its new formation. We need to determine *a)* whether the migration is feasible, and *b)* the steps the system needs to take to migrate from one configuration to another. A step for a single structure consists of disassembly along a straight line. From the set of all possible disassembly (or "break") lines, we need to decide which one is the best option. This is illustrated in Fig. 4, which shows all possible disassembly lines for a doughnut structure (in dotted red lines) and two examples of splitting.

### *A. Self-disassembly*

We know that if a module has a faulty rotor, then it cannot control its attitude properly [9] and will be unable to perform docking actions. The minimum number of co-planar rotors needed for a quadrotor to fly without undesired rotations is four. Each rotor must be in a different quadrant of the $xy$–plane of the structure frame. Thus, we must ensure that when we break a structure to rearrange the quadrotors, no module with failed rotors becomes isolated.

*1) Method 1: Full self-disassembly:* One solution is to completely disassemble the structure into as many substructures as possible, to the point of individual modules if possible. This approach was studied in [15] for structures without faulty rotors. The fault-tolerant version of this approach would need to account for the fact that modules with faulty rotors cannot be isolated at any point during the process.

One way to disassemble the structure is to enumerate all possible links to break and then choose the "path of breaking links" that is best for disassembly. However, this strategy is highly inefficient. For an $m\times n$ rectangular structure $\mathcal{S}$, there will be $(m-1)+(n-1)$ possible undocking actions to consider when we perform the first disassembly. For each sub-structure generated, we have $O(m+n)$ disassembly options to consider. A naïve approach would be checking all the possible combinations to find the best self-disassembly sequence. However, this would result in exponential complexity.

We observe that self-disassembly will generate many non-unique substructures that we do not need to find disassembly steps for more than once (given a unique fault configuration). This optimal sub-problem structure makes it appropriate to use dynamic programming, thereby reducing the exponential complexity to polynomial complexity. We present our dynamic programming approach in Algorithm 1.

In Algorithm 1, $\mathcal{C}$ maps a structure and known faults to a particular disassembly ("break") location and its associated cost. genSplits$(\mathcal{S},\Gamma,\mathcal{C})$ is the main recursive call. It takes a structure, the states of rotors in that structure, and the cost map as an input. It will recursively find the best disassembly sequence and store the results in the map $\mathcal{C}$. The function splittable$(\mathcal{S},\Gamma,b)$ checks whether the structure $\mathcal{S}$ can be split along the break line $b$ such that both generated substructures can stay aloft on their own given the current $\Gamma$. The function getBreaklines$(\mathcal{S})$ returns a list of all possible lines of the structure $\mathcal{S}$ along which we can break it in two.

**Algorithm 1** Dynamic programming algorithm to find disassembly sequence for a structure $\mathcal{S}$.

```
Require: $\mathcal{S},\Gamma$
 1: $\mathcal{C} \leftarrow \{\}$                                  ▷ $Path : map\langle(\mathcal{S},\Gamma),(breakline, cost)\rangle$
 2: genSplits($\mathcal{S},\Gamma,\mathcal{C}$)
 3: function genSplits($\mathcal{S},\Gamma,\mathcal{C}$)
 4:     if size($\mathcal{S}$) = 1 then
 5:         return 1
 6:     if exists($\mathcal{C}[(\mathcal{S},\Gamma)]$) then
 7:         return $\mathcal{C}[(\mathcal{S},\Gamma)].cost$
 8:     $\mathcal{B} \leftarrow$ getBreaklines($\mathcal{S}$)
 9:     $\beta \leftarrow \infty$                                 ▷ cost of breakline of min cost
10:     for $b \in \mathcal{B}$ do
11:         if splittable($\mathcal{S},\Gamma,b$) then
12:             $[\mathcal{S}_1,\Gamma_1,\mathcal{S}_2,\Gamma_2] \leftarrow$ split($\mathcal{S},b$)
13:             $\sigma \leftarrow$ genSplits($\mathcal{S}_1,\Gamma_1,\mathcal{C}$) + genSplits($\mathcal{S}_2,\Gamma_2,\mathcal{C}$)
14:             if $\sigma < \beta$ then
15:                 $\beta \leftarrow \sigma$
16:                 $\mathcal{C}[(\mathcal{S},\Gamma)] \leftarrow (b,\sigma)$
17:         else                                           ▷ Unsplittable due to faults
18:             return 1                                   ▷ Splitting here would cause a crash
```

This function does not take the feasibility of performing such a disassembly into account. The base case returns value 1 because we wish to find the recursive depth, and the minimum recursive depth is 1 for a structure that can never be disassembled; the algorithm at minimum, still needs to determine that the given structure cannot be disassembled in the first call to GENSPLITS.

Given an $m \times n$ structure, there are at most $mn$ sizes of sub-structures. For instance, a $4 \times 3$ structure breaks down into sub-structures of sizes: $\{1 \times 1, 1 \times 2, 1 \times 3, 2 \times 1, 2 \times 2, 2 \times 3, 3 \times 1, 3 \times 2, 3 \times 3, 4 \times 1, 4 \times 2, 4 \times 3\}$. If we do not have to consider faults, then the final complexity of the problem would be $O((m+n)mn)$, where the $(m+n)$ factor is because it takes linear time to find the best disassembly location for a given structure. However, structures with the same $\mathcal{S}$ and different $\Gamma$ are not equal. A $3 \times 1$ sub-structure with a fault in the top does not have the same disassembly options as a $3 \times 1$ sub-structure with a fault in the bottom. The complexity thus depends on how many unique $3 \times 1$ sub-structures might be generated over the course of the algorithm. It similarly depends on this computation for each sub-structure size possible given the original structure.

There is only one possible $m \times n$ sub-structure that can be generated for the original $m \times n$ structure (i.e., the root of the recursion). There are two ways to generate a $m \times (n-1)$, three ways to generate a $m \times (n-2)$, and so on. If we fix the value $m$, there are $\sum_{j=1}^{n} j$ sub-structures that can be generated. Since the number of rows in a sub-structure can be any number in the set $\{1, 2, \ldots, m\}$, the worst-case number of sub-structures to consider is

$$\sum_{i=1}^{m} \sum_{j=1}^{n} ij = \frac{m(m+1)n(n+1)}{4} = O(m^2 n^2).$$

For each sub-structure, it takes linear time to find the most optimal disassembly location, since we need to examine each disassembly operation once to see if it is the one that incurs the least cost. Thus, the time complexity of the algorithm is $O((m+n)m^2n^2)$.

*2) Method 2: Partial self-disassembly:* Our partial self-disassembly method adapts ideas from Algorithm 1 but improves efficiency by eliminating unnecessary disassembly operations.

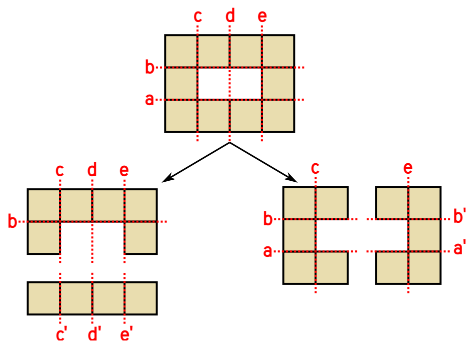


Fig. 4: Self-disassembly options. Undocking can be performed along multiple break-lines. For instance, break-line *a* (bottom left) and break-line *d* (bottom right).

Observe that Algorithm 1 always fully disassembles the structure except for cases in which the disassembly would cause a structure with faulty rotors to become isolated. However, there are scenarios in which it is unnecessary to further break down a structure (even if it is possible). For instance, if a particular $2 \times 1$ sub-structure appears in both the original configuration and the new configuration, then it does not need to be broken down further, even if neither module contains a faulty rotor. Avoiding such unnecessary disassembly actions is useful in two ways: *i)* It saves energy and time in performing the reconfiguration, and *ii)* it reduces the computation the algorithm must perform to find the best possible intermediate stages for reconfiguration. This technique can also result in fewer intermediate stages than the full disassembly algorithm does.

**Algorithm 2** Find disassembly sequence for a structure $\mathcal{S}$.

```
Input: $\mathcal{Q}, \mathcal{Q}^*$                    ▷ The input $\mathcal{Q}$ denotes a triple $(\mathcal{S}, \Pi, \Gamma)$.
Output: $\mathcal{C}$          ▷ $\mathcal{C}$ maps a unique $\mathcal{Q}$ to optimum (disassembly line, cost)
$\mathcal{C} \leftarrow \{\}$          ▷ Pass by reference–$Path: map\langle \mathcal{Q}, (breakline, cost) \rangle$
$\beta \leftarrow$ RECONFIGURE($\mathcal{Q}, \mathcal{Q}^*, \mathcal{C}$)
function RECONFIGURE($\mathcal{Q}, \mathcal{Q}^*, \mathcal{C}$)
    if SIZE($\mathcal{Q}.\Pi$) $= 1$ then return 1
    if EXISTS($\mathcal{C}[\mathcal{Q}]$) then return $\mathcal{C}[\mathcal{Q}].cost$
    $\beta \leftarrow \infty$                    ▷ cost of breakline of min cost
    for each possible splitting location do
        for each substructure pair generated by split: $\mathcal{Q}_1, \mathcal{Q}_2$ do
            if at least one substructure is invalid then
                continue
            Perform checks in CHECK_PAIR($\mathcal{Q}_1, \mathcal{Q}_2, b, \beta$)
    return $\beta$
function CHECK_PAIR($\mathcal{Q}_1, \mathcal{Q}_2, breakline, \beta$)
    $\sigma \leftarrow \max\{\mathcal{C}[\mathcal{Q}_1].cost, \mathcal{C}[\mathcal{Q}_2].cost\}$
    $\mathcal{T} \leftarrow \mathcal{Q} \forall \mathcal{Q} \in \{\mathcal{Q}_1, \mathcal{Q}_2\}$ if $\mathcal{Q} \notin \mathcal{C}$
    if RECONFIGURABLE($\mathcal{T}, \mathcal{Q}^*$) then
        if $\sigma + 1 < \beta$ then
            $\beta \leftarrow \sigma + 1$
            $\mathcal{C}[\mathcal{Q}] \leftarrow (breakline, \beta)$
    else
        $\psi_i \leftarrow$ RECONFIGURE($\mathcal{Q}_i, \mathcal{Q}^*, \mathcal{C}$)    $\forall \mathcal{Q}_i \in \mathcal{T}$
        if $\max_{i \in \{1,2,\ldots|\mathcal{T}|\}} \psi_i$ then $\sigma \leftarrow \max_{i \in \{1,2,\ldots|\mathcal{T}|\}} \psi_i + 1$
        if $\sigma < \beta$ then
            $\beta \leftarrow \sigma$
            $\mathcal{C}[\mathcal{Q}] \leftarrow (breakline, \beta)$
    return $\beta$
function RECONFIGURABLE($\mathcal{T}, \mathcal{Q}^*$)
    for $\mathcal{Q} \in \mathcal{T}$ do
        if not all modules in $\mathcal{Q}$ can be placed in correct spot in $\mathcal{Q}^*$ then
            return False
    return True
```

Algorithm 2 presents our partial-disassembly method that optimizes Algorithm 1 based on this insight. The base cases in this algorithm are the same as in Algorithm 1. As before, we generate all possible splitting locations. For each possible splitting location, the algorithm must consider the split and determine whether the generated sub-structures would be valid (i.e., stay aloft on their own). If so, that pair of sub-structures is passed to CHECK_PAIR$(\mathcal{Q}_1, \mathcal{Q}_2, breakline, \beta)$. CHECK_PAIR determines whether the two sub-structures $\mathcal{Q}_1$ and $\mathcal{Q}_2$ can be directly placed into the new desired structure. That is, each module in $\mathcal{S}_1$ would be able to achieve its desired position in the final structure without losing its relative position in $\mathcal{Q}_1$. We require each module to be placed in the exact position that the MILP returns; future work will relax this constraint. Further, the sub-structure cannot "enclose" another, i.e., we cannot force another substructure to dock to two or more locations on the current sub-structure simultaneously.

If both conditions are met, then no further recursion is needed on such a structure. If not, then the structure will be broken down further in recursion. The migration cost is computed in the same manner as Algorithm 1, i.e., it is the maximum depth of recursion needed such that each module from the original configuration can be feasibly placed in its new position in the new configuration.

It is important to consider the case where the presence of faulty rotors prevents the dynamic programming algorithm from generating sufficiently small substructures that can be placed in the exact positions the MILP computed. For instance, let's say that we have a structure $[1,2,3,4]$ in a line configuration and we desire the structure $[4,3,2,1]$. If a rotor of module 1 is faulty, then we cannot disconnect module 1 from module 2, making it impossible to exactly create the structure $[4,3,2,1]$.

*3) Infinite cost disassembly:* The migration cost may be infinite if there is no method by which it is feasible to migrate from the current structure into the new one due to where the modules containing faults exist. In this case, we introduce additional constraints to bind modules with faulty rotors to adjacent modules and rerun the MILP (this is what the final constraint in Section III-A was for). Since we have bound all modules that need extra support to other modules, we only need to rerun the MILP once to obtain a solution we can feasibly reconfigure to.

## B. Self-assembly

Once the best feasible self-disassembly operation is completed, the structure proceeds to re-assemble.

*1) Finding the minimal-cost self-assembly sequence:* Parallelizing the docking actions is desirable so as to minimize assembly time. Our approach is summarized in Algorithm 3. Here, $\mathcal{L}$ is used to denote a set of substructures and $\mathcal{A}$ denotes a mapping of $\mathcal{L}$ to a dividing line that we will assemble the two sides on. We then recurse over the lists of structures on the two sides of this line to generate corresponding assembly lines. The algorithm is designed to minimize the number of assembly layers needed to construct the whole structure.

**Algorithm 3** Find a self-assembly sequence for a set of substructures $\mathcal{L}$.

```
    Input: $\mathcal{A}, \mathcal{L}, \mathcal{Q}^*$          ▷ The input $\mathcal{Q}$ denotes a triple $(\mathcal{S}, \Pi, \Gamma)$
    Output: $\mathcal{A}$
 1: $\mathcal{A} \leftarrow \{\}$          ▷ Pass by reference–$Path : map\langle \mathcal{L}, assemble\_line \rangle$
 2: ASSEMBLE($\mathcal{A}, \mathcal{L}, \mathcal{Q}^*$)
 3: function ASSEMBLE($\mathcal{A}, \mathcal{L}, \mathcal{Q}^*$)
 4:     $c_{best} \leftarrow \infty$          ▷ cost to assemble partition
 5:     for $b \in$ GET_BREAKLINES($\mathcal{L}$) do          ▷ assembly lines same as breaklines
 6:         $c_b \leftarrow 0$
 7:         if ASSEMBLABLE($\mathcal{L}, b$) then
 8:             $\mathcal{H} \leftarrow$PARTITION($\mathcal{L}, \mathcal{Q}^*, b$)
 9:             for $h \in \mathcal{H}$ do
10:                 $c_{b,h} \leftarrow$ ASSEMBLE($\mathcal{A}, h, \mathcal{Q}^*$)
11:                 if $c_{b,h} > c_b$ then
12:                     $c_b \leftarrow c_{b,h}$
13:         else
14:             $c_b \leftarrow \infty$
15:         if $c_b < c_{best}$ then
16:             $\mathcal{A}[\mathcal{L}] \leftarrow b$
17:             $c_{best} \leftarrow c_b$
18:     return $c_{best}$
```

The function ASSEMBLE$(\mathcal{A}, \mathcal{L}, \mathcal{Q}^*)$ is the main recursive call that generates the assembly sequence. As before, GET_BREAKLINES$(\mathcal{L})$ is used to find lines along which structures can be assembled, though the argument passed in is adapted for assembly. ASSEMBLABLE$(\mathcal{L}, b)$ checks that no sub-structure in $\mathcal{L}$ crosses the assembly line $b$, as if it does $b$ is not a valid assembly location. Note that because of the way the disassembly algorithm was designed, we are guaranteed that there is at least one sequence of assembly lines such that the full structure can be recreated in the desired configuration.

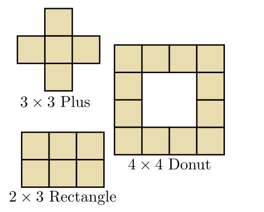
*2) Path Planning for self-assembly:* The path planning is relatively straightforward. Once we have the mappings from Algorithm 3, we move every pair of substructures that will be combined in a particular assembly step to be on an isolated $z$-height. Then, we keep one substructure stationary while moving the other into position to dock in the correct position. We do the same for the next layer of assembly, and again for the following layer, until the entire structure is assembled.

## V. Evaluation

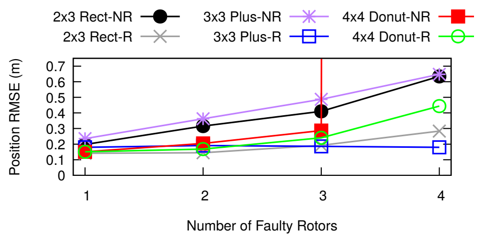
To evaluate the applicability and performance of our technique, we performed a series of simulations. We evaluate the optimality of computed configurations through simulations in ROS (adapting the simulator available from `https://github.com/dsaldana/modquad-simulator`) with several ModQuad structures and fault scenarios. The simulator solves the Newton-Euler equations for multiple flying structures, taking into account the impact of the faulty rotors in their dynamical model. It also combines/separates rigid bodies after docking/undocking actions. Our main objective is to show *i)* how effective self-adaptation is in minimizing the impact of failures on performance in terms of staying true to task path and energy efficiency, *ii)* the complexity and scalability of self-reconfiguration, and *iii)* the end-to-end integration of the MILP solving and the disassembly and assembly sequence at run time.

## A. Benefits of self-reconfiguration

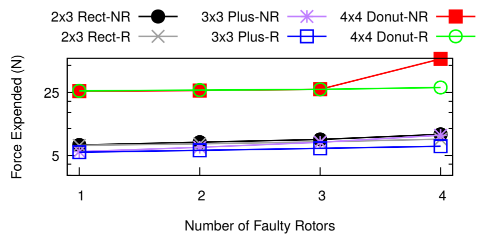
We compared structures traveling along trajectories with faulty rotors under two settings: *i)* the default setting, where the structures remain unchanged, and *ii)* the self-adaptation setting, where the structures self-reconfigure using our proposed technique. For this, we performed simulations in ROS using the structures shown in Fig. 5a, with $1-4$ faults. For each structure and each desirable trajectory, we measured the root-mean-square-error (RMSE) in position and the integral of force expended over the course of the



(a) Structures in evaluation.

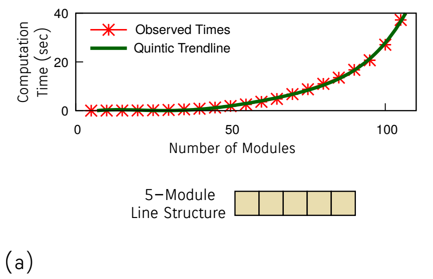


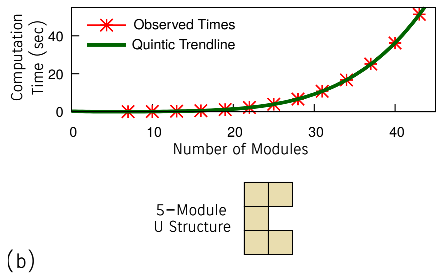
(b) RMSE in meters.

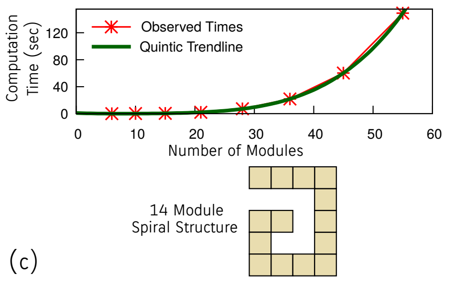


(c) Force applied by the structure.

Fig. 5: Effectiveness of self-adaptation. '-NR' indicates not reconfigured and '-R' indicates reconfigured.

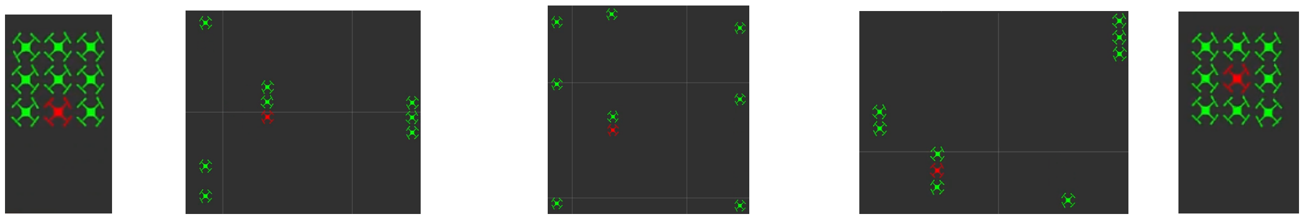






(a) (b) (c)

Fig. 6: Times to invert module allocations in structures. Representative structures showing each shape of a specific size are shown below the corresponding plot.



(a) (b) (c) (d) (e)

Fig. 7: Simulation snapshots of self-reconfiguration. (a) Initial configuration (b) Disassembling (c) Reaching max disassembly needed *and* not isolating faulty modules (d) Enter assembly process (e) New configuration.

trajectory. The measured RMSE values provide an estimate of how effect of reconfiguration on ensuring task completion in the presence of failures. The measured force expended gives us an estimate of how much more (or less) efficient reconfiguring the structure is in utilizing available resources compared to the default case.

Fig. 5b shows the RMSE averaged across the three dimensions. The errors shown are for both non-reconfigured structures ('-NR') and reconfigured structures ('-R'). The results show that, across all structures, the position error under the self-adaptation setting is much smaller than that under the default setting. For the $4 \times 4$ Donut-NR case, the structure was unable to follow the trajectory with 4 faults in the default setting. This demonstrates that our self-adaptation technique can effectively minimize the impact of faults on the system's performance and mission completion.

Fig. 5c shows the force expended by structure in traveling trajectory. We observe that self-adaptation leads to a smaller force expended across all structures, and thus is more efficient in optimizing resources compared to the default setting.

## B. *Algorithm efficiency*

We evaluate the time the dynamic programming algorithm takes to compute the disassembly and reassembly sequence for migrating from a current configuration to the desired one, as the number of modules in the structures increases. Since the disassembly algorithm is more complex than the assembly algorithm, we focus our evaluation on the former (i.e., Algorithm 2).

For our evaluation, we used three different structure types to assess how the performance scales as a function of both the number of modules and the structure type. We considered structures for which we can define "inversion"; that is, a module $k$ modules away from the last module of the structure is moved to be $k$ modules away from the first module in the structure. For example, suppose the original configuration was $[1,2,3,4]$, then the desired inverted module-to-position assignment would be $[4,3,2,1]$. This also allows us to compare scaled versions of the problem more easily since the self-adaptation involves solving similar types of problems, whereas in other structures we cannot say the same. We performed such inversion tests for linear structures, U-shape structures, and for spiral structures. Note that we used spiral structures purely for stress-testing the self-disassembly algorithm. As per [13], such a structure would be unable to fly because of the symmetry assumptions the ModQuad controls rely on.

The results are shown in Fig. 6. We observe that a fitted fifth-order polynomial curve closely matches the measured time, confirming the validity of our analysis in practice. Further, the time to find disassembly steps scales well with the structure size, and it is able to solve complicated spiral structures with more than fifty modules within reasonable time (under two minutes).

### C. End-to-end integration

We tested the integration of the MILP with the disassembly and assembly algorithms in simulation. A video of reconfiguring a three-module line structure, five-module plus-shaped structure, and nine-module square structure is available at `https://youtu.be/S1vK6crfwIg`. Snapshots from a top-view recording of the simulation are shown in Fig. 7. The structure starts with the red module in a non-optimal location, breaks apart into as many pieces as necessary while never isolating the module with the faulty rotor, and re-assembles into a new structure.

## VI. Conclusion and Future Work

We presented an effective fault-tolerance technique for modular aerial systems that increases their robustness against rotor failures through self-reconfiguration. We designed a mixed integer linear program to determine an optimal module allocation in the structure based on rotor faults and desired trajectories. We proposed an efficient dynamic programming algorithm that minimizes the number of required steps for self-reconfiguration. Our results show that our technique substantially increases the robustness of the system while utilizing resources efficiently.

As future work, we plan to explore refinements of our algorithms to relax constraints imposed on the self-assembly, as well as distributed approaches for self-reconfiguration. Online fault-detection methods and custom controllers are also interesting extensions that could considerably improve the performance of the flying structure during a mission. Other interesting directions include: changing the shape of the structure, excluding modules with many faulty rotors from the reconfigured structure, and handling failures that occur during the self-reconfiguration process.

## References

[1] F. Augugliaro, S. Lupashin, M. Hamer, C. Male, M. Hehn, M. W. Mueller, J. S. Willmann, F. Gramazio, M. Kohler, and R. D'Andrea. The flight assembled architecture installation: Cooperative construction with flying machines. *IEEE Control Systems Magazine*, 34(4):46–64, 2014.

[2] A. L. Christensen, R. OGrady, and M. Dorigo. From fireflies to fault-tolerant swarms of robots. *IEEE Transactions on Evolutionary Computation*, 13(4):754–766, 2009.

[3] M. J. Duffy and T. C. Samaritano. The lift! project–modular, electric vertical lift system with ground power tether. In *33rd AIAA Applied Aerodynamics Conference*, page 3013, 2015.

[4] B. Gabrich, D. Saldaña, V. Kumar, and M. Yim. A flying gripper based on cuboid modular robots. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7024–7030. IEEE, 2018.

[5] G. Li, B. Gabrich, D. Saldaña, J. Das, V. Kumar, and M. Yim. ModQuad-Vi: A vision-based self-assembling modular quadrotor. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 346–352. IEEE, 2019.

[6] Q. Lindsey, D. Mellinger, and V. Kumar. Construction of cubic structures with quadrotor teams. *Proc. Robotics: Science & Systems VII*, 2011.

[7] D. Mellinger and V. Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525. IEEE, 2011.

[8] D. Mellinger, M. Shomin, N. Michael, and V. Kumar. Cooperative grasping and transport using multiple quadrotors. *Springer Tracts in Advanced Robotics*, 83 STAR:545–558, 2012.

[9] M. W. Mueller and R. D'Andrea. Stability and control of a quadrocopter despite the complete loss of one, two, or three propellers. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 45–52, May 2014.

[10] R. Naldi, F. Forte, A. Serrani, and L. Marconi. Modeling and control of a class of modular aerial robots combining under actuated and fully actuated behavior. *IEEE Transactions on Control Systems Technology*, 23(5):1869–1885, Sept 2015.

[11] R. Oung and R. DAndrea. The distributed flight array: Design, implementation, and analysis of a modular vertical take-off and landing vehicle. *The International Journal of Robotics Research*, 33(3):375–400, 2014.

[12] J. Paulos, N. Eckenstein, T. Tosun, J. Seo, J. Davey, J. Greco, V. Kumar, and M. Yim. Automated self-assembly of large maritime structures by a team of robotic boats. *IEEE Transactions on Automation Science and Engineering*, 12(3):958–968, 2015.

[13] D. Saldaña, B. Gabrich, G. Li, M. Yim, and V. Kumar. ModQuad: The flying modular structure that self-assembles in midair. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 691–698, May 2018.

[14] D. Saldaña, P. M. Gupta, and V. Kumar. Design and control of aerial modules for inflight self-disassembly. *IEEE Robotics and Automation Letters*, 4(4):3410–3417, 2019.

[15] D. Saldaña, P. M. Gupta, and V. Kumar. Design and control of aerial modules for inflight self-disassembly. *IEEE Robotics and Automation Letters*, 4(4):3410–3417, Oct 2019.

[16] D. Saldaña, B. Gabrich, M. Whitzer, A. Prorok, M. F. Campos, M. Yim, and V. Kumar. A decentralized algorithm for assembling structures with modular robots. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2736–2743. IEEE, 2017.

[17] J. Seo, J. Paik, and M. Yim. Modular reconfigurable robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 2018.

[18] A. Walter, M. McKay, R. Niemiec, and F. Gandhi. Trim analysis of a classical octocopter after single-rotor failure. In *2018 AIAA/IEEE Electric Aircraft Technologies Symposium (EATS)*, pages 1–20. IEEE, 2018.

[19] J. Werfel and R. Nagpal. Three-dimensional construction with mobile robots and modular blocks. *The International Journal of Robotics Research*, 27(3-4):463–479, 2008.

[20] H. Yang, S. Park, J. Lee, J. Ahn, D. Son, and D. Lee. Lasdra: Large-size aerial skeleton system with distributed rotor actuation. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7017–7023, May 2018.

[21] M. Yim, W.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, and E. Klavins. Modular Self-reconfigurable Robot Systems: Challenges and Opportunities for the Future. *IEEE Robotics & Automation Magazine*, 14(1):43–52, 2007.

[22] M. Zhao, K. Kawasaki, T. Anzai, X. Chen, S. Noda, F. Shi, K. Okada, and M. Inaba. Transformable multirotor with two-dimensional multilinks: Modeling, control, and whole-body aerial manipulation. *The International Journal of Robotics Research*, 37(9):1085–1112, 2018.