Creating a Smarter Membrane: Automatic Code Generation for Modular Self-Reconfigurable Robots

Jeremy Kubica[1]
Carnegie Mellon University
jkubica@ri.cmu.edu

Eleanor Rieffel
FXPAL
rieffel@pal.xerox.com

Abstract

This work extends previous research on developing control software for modular robotic smart membranes to a second module type with more complicated movement, to 3-D membranes, to the presence of gravity, and to less easily manipulatable objects. Moreover, it extends the capabilities of the membranes from simple filtering to more complex sorting tasks. The control software we developed is completely decentralized, and automatically generated.

1 Introduction

Both biological and manmade systems contain a myriad of devices that filter or sort at a variety of scales. Most of these devices, however, perform a fixed task on a limited set of objects. Previously, Bennett et. al. [1] explored the use of modular robots to create programmable smart membranes that have great flexibility as to the type of task and the set of objects. Here, we extend that work.

Smart membranes composed of modular robots have the advantage of being highly flexible. The modules can configure into a membrane, a lattice of modules, in which the modules work together to manipulate objects. The same code used to sort molecules with nano-scale modules could sort parts in a manufacturing process with larger modules. Further, since the modules can be reprogrammed the membrane can be reused for different tasks. For example, through reprogramming, a factory can use the same membrane to sort nuts, then to sort bolts, and then to filter defective nails.

Modular robots, made up of numerous identical modules, are a natural choice of hardware for creating such flexible devices. However, programming these robots is known to be a difficult task, and little work has been done in this area. We look at generating a particularly elegant form of software for modular robots: completely decentralized software. Thus, we look at the case where there is no central controller and every module runs the same piece of software.

[1]Supported by FXPAL

This problem is attractive for exploring automated program generation: the desired behavior is easy to specify, but finding software to accomplish that behavior is not easy. Distributed control is recognized as a difficult problem. In our case, the problem is particularly complex because the connectivity topology of the modules constantly changes. Here we describe the application of genetic programming to generating completely decentralized software to enable a modular robot, made up of modules being built at Xerox PARC, to perform filtering and sorting tasks.

2 Related Work

Previous research in the use of modular robots for smart membranes was done by Bennett et. al. [1], who used genetic programming to evolve control programs for modular robots consisting of sliding-style modules [2, 8]. Their programs enabled a 2-dimensional membrane to filter objects by color. Additionally, Kubica et. al. [6] looked at control programs for object manipulation within a membrane.

This research extends previous research on smart membranes in several ways. First, it extends to a second module type for which basic movement is more complicated. Second, earlier work with automatic code generation for modular robotic membranes dealt only with 2-D membranes, where as our membrane is three-dimensional with gravity acting along the negative Z-axis. The control problem is more complicated in our case, because objects must be supported by the membrane if they are not to move in the negative Z direction. Third, we do not give the modules the capability of grabbing objects and pulling them. The modules are able to push an object by extending an arm. Further, to move the objects through the membrane, the modules must use gravity by forming gaps for the objects to fall into. Thus the amount and type of manipulation that can be preformed by the modules is limited. Finally, we develop membranes that perform the more complex action of sorting objects.

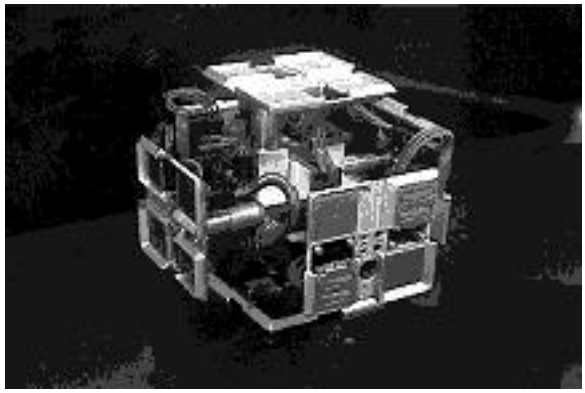

Figure 1: One telecube module.

3 Modular Robotic Hardware and Simulator

The following experiments were run by connecting FXPAL's genetic programming system to a modular robot simulator, built by J. Kubica and S. Vassilvitskii. The robot modules we consider are the TeleCube modules currently being developed at Xerox PARC [13] and shown in Figure 1. The modules are cube shaped with an extendable arm on each face, similar in design to the modules designed and built at Dartmouth [9, 11]. The arms are assumed to extend independently up to half of the body length, giving the robot modules an overall 2:1 expansion ratio. For simplicity, we restrict the arms to fully extended or fully retracted states. The expansion and contraction of these arms provide the modules with their only form of motion.

Latches at the end of each arm enable two aligned modules to connect to each other. The arm motion together with the latching and unlatching capability means that the connectivity topology of a modular robot can change greatly over time. As shown in [14], this motion is sufficient to enable arbitrary reconfiguration within a large class of shapes.

Further, the modules are assumed to be able to push objects by extending their arms. In order to push an object, the object has to be close enough to push and the object's motion cannot be blocked. If either of these conditions is not met, we say the module is unable to push the object and the object does not move.

The simulated world is composed of a three-dimensional grid in which the length of one grid cube is equal to the length of a fully extended arm. Fully contracted modules occupy eight grid locations arranged in a 2 x 2 x 2 cube. Each extended arm occupies an additional four grid locations.

3.1 Sensors and Communication

Each module is assumed to have simple sensing and communication abilities that resemble capabilities that will be given to the Telecube modules currently being built at Xerox PARC. Modules can send messages to their immediate neighbors. Each module is also assumed to be able to detect contact at the end of each arm, how far each arm is extended, connections for each arm, and adjacent modules that are within two arm's length distance from the module body.

3.2 Connectivity

Maintaining global connectivity is vital for power routing when using a single power source, as can be expected for an application such as a membrane. It also greatly facilitates alignment of connecting modules and inter-module communication.

Modules in the simulator maintain maximum connectivity by watching for neighbors. If a new neighbor is sensed, the module extends the arm in that direction and tries to connect with the neighbor. Connections are maintained unless the module moves or explicitly retracts its arm. Note that an efficient global connectivity check can be implemented in hardware by checking power interruptions from a global power supply. This check prevents modules from disconnecting if doing so would result in a global disconnection.

3.3 Movement

Movement of a single TeleCube module is a complex task that requires the module to disconnect from some neighbors while pushing and pulling off others. Due to the difficulty of simple motion, we provide the modules with a movement primitive: an explicit sequence of actions that enables a module to move in a given direction as illustrated in Figure 2. The module moves by simultaneously extending an arm behind it while contracting an arm in front, effectively "sliding along its arms". The arms moving can either be the module's or its neighbor's depending on whose arms are already extended. This action can be synchronized by direct neighbor-to-neighbor communication. Prior to moving, the module:

1. confirms that it has a neighbor along the direction of motion off of which it can push or pull,

2. checks there is a free space ahead of it, by expanding its arm (if it is not already expanded) and checking the contact sensor, and

3. disconnects from all neighbors perpendicular to the direction of movement.

At any point during movement, the movement can fail, in which case the module reverses what has been done so far and returns the fact the movement failed.

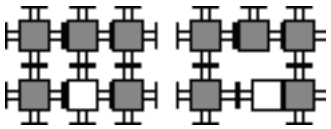

Figure 2: Module in white moves towards right.

4 Membrane Problem and Genetic Programming Setup

We examine two different types of membrane problems: filtering membranes and sorting membranes. Both types require the membrane to use some real valued, [0,1], measurable property of the objects to determine how it should treat them. This property can vary depending on the membrane application. For example, reflectivity could be used to distinguish objects being filtered for recycling, color could be used to sort different components, and weight could be used to check for defective parts.

4.1 Primitives

The problems all shared a common set of primitives. These primitives were largely derived directly from the basic actions and abilities of the modules and simple computational constructs. In addition, gradient primitives, shown to be effective for communication in modular robots [3, 6, 12], were added to the basic communication functions. The primitives are:

- `(Move direction)` the module tries to move in `direction`. This function returns `true` if the module was able to make the move.
- `(MoveAwayFrom direction)` the module tries to move away from `direction`. If the module cannot move in the opposite direction, it tries to move in each of the perpendicular directions in a random order. Returns `true` if the module was able to move.
- `(Push direction)` the module attempts to push in `direction` by extending its arm. This function returns `true` if it hit an object and was able to continue expanding its arm.
- `(RetractArm direction)` the module retracts its arm in `direction`. This function returns `true` if the module was able to retract its arm.
- `(OppDir direction)` returns the direction opposite to `direction`.
- `(ReadReg index)` reads a value from register `index` and returns this value. All registers are initialized to zero.
- `(SetReg index value)` writes `value` to register `index` and returns `value`.
- `(GetMessage direction type)` reads the last message of `type` from `direction`. Returns `0` if the message queue does not contain an instance of this message type.
- `(SendMessage direction type value)` sends a message of `type` to `direction` with the value of `value`. Returns `value`.
- `(GetGradientVal type)` returns the current value of gradient `type`.
- `(GetGradientDir type)` returns the current direction of gradient `type` with ties broken randomly.
- `(SendGradient type val)` emits a gradient of `type` with strength of `val`.
- `(ReadSensorIsObject direction)` returns `true` if there is an object within one arm length from the
- The Boolean operators `(And arg1 arg2)`, `(Not arg1)`, and `(LT arg1 arg2)`.
- The mathematical operators `(Add arg1 arg2)` and `(Sub arg1 arg2)`.
- The flow constructs `(If test arg1 arg2)` and `(ProgN arg1 arg2)`.

The following terminals were also included:

- Numeric constants: `0.0`, `0.1`, `0.2`, `0.4`, `0.6`, `0.8`, and `1.0`.
- `(RandDir)` returns a random direction value.
- `(NormalizeDensity)` attempts to move in a randomized fashion so as to maintain a distance of 1 arm length from all neighboring modules.

We also gave the genetic programming runs additional primitives for each problem. These primitives were designed to aid genetic programming in finding a solution and either encapsulated problem specific information or low-level information that was thought to be helpful for obtaining a solution. Determining which information to add was the result of parallel attempts to examine the unsuccessful results produced by the genetic programming and attempts to hand code problem solutions. See [7] for a description of our process. The specific primitives added for each problem are discussed with setup of the related problem.

All values, types, directions, and indices are encoded as floating point numbers. The different possibilities are encoded as evenly spaced floats between 0 and 1. Other numbers used as input are brought into the range [0,1] via the modulus operation and rounded to the nearest valid number. The directions $+Z$ axis, $-Z$ axis, $-Y$ axis, $+Y$ axis, $+X$ axis, and $-X$ axis are encoded as 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0 respectively. The number of types of registers, messages, and gradients are 6, 3, and 4 respectively.

4.2 Settings

The genetic programming runs were carried out on multiple processors on a toroidal grid, each with a population of 100 individuals. At the end of each generation, and for each of the four directions, 2% of the population on a processor was randomly chosen to be sent to a neighboring processor. The genetic programming parameters were determined by empirical success and held fixed for all problems. The leaf and internal crossover rates were 2% and 78% respectively. Leaf and internal mutation rates were 9% and 10% respectively. Cloning made up the remaining 1%. Parents were chosen via tournament selection with tournament size 3. Maximum tree depths were 5, 9 and 3 for the initial population, results of crossover and mutation, and subtrees created during mutation respectively. We used strongly typed genetic programming [4].

4.3 Initial Membrane

The problems use an initial membrane of aligned modules that are equally spaced at one arm's length apart. The size of the membrane varies with the run, but is held constant for all trials.

5 Filtering Membrane Problem and Results

Filtering membranes either accept or reject an object based on its value. As mentioned above, this value can vary depending on the application. An accepted object should pass directly through the membrane, while rejected objects should remain on top of the membrane. We examined the case of accepting any objects whose value was greater than 0.5.

5.1 Fitness Function

The fitness, which we try to minimize, was determined by running each module for 60 time steps on 50 test worlds with a 6 x 3 x 4 membrane. Half of the test worlds included objects that should be passed and half that should be rejected. The placements of the objects on top of the membrane were randomly generated once at the start of the GP run.

The fitness function for the filtering membrane problem consisted of a penalty for incorrectly passing or rejecting, and a penalty for the membrane breaking apart. The former was $(25 * errZ)$ where $errZ$ was the difference in the correct final height and the actual final height. The later was assigned by adding 2 to the fitness for each module with height above the original membrane surface and 1 for each empty block of eight (2 x 2 x 2) within the membrane.

5.2 Additional Primitives

The `(ReadSensorShouldPass direction)` returns `true` if and only if there exists an object in `direction` that should be passed by the membrane. The reason for including this primitive is that it encodes information specific to the problem. While the genetic programming system could deduce the criteria, in general the users of a filter know what criteria they would want to filter on. Thus, there is no reason not to explicitly give the modules this information instead of making the genetic program determine both the problem and the solution.

5.3 Results

This problem was run on eight processors. A solution was found on generation 31.

```
(If (ReadSensorObjectShouldPass 0.1) (If (If (ProgN
(Move 1.0) (RetractArm 0.8)) (If (ProgN (Move
1.0) (RetractArm 0.8)) (Move (ProgN 1.0 0.2))
(Move 1.0)) (RetractArm 0.1)) (Move 1.0) (Move
(OppDir .6))) (If NormalizeDensity (If (If (Move
(SendGradient(GetMessage (OppDir 0.1) 1.0) 0.8))
NormalizeDensity (If NormalizeDensity (ProgN (Move
1.0) (RetractArm 0.8)) (ProgN (MoveAwayFrom RandDir)
(RetractArm 0.8)))) (Move 1.0) (LT 0.0 (GetMessage
(SendMessage (SetReg 0.6 0.6) 0.2 (GetGradientVal
0.0)) (OppDir (SendGradient 1.0 (GetGradientVal
0.2)))))) (Not NormalizeDensity)))
```

The behavior induced by this program is unsurprising. Undesirable objects remain on top of the membrane, supported by modules from below. Desirable objects move though the membrane by falling into the gaps created when the modules beneath them moved out of the way. After an object has passed, the membrane non-deterministically moves so as to close the gap. This "passing" behavior is shown in Figure 3 at three times during a single run. The modules in the figure are shown as semi-transparent blocks to allow the spherical object to be seen.

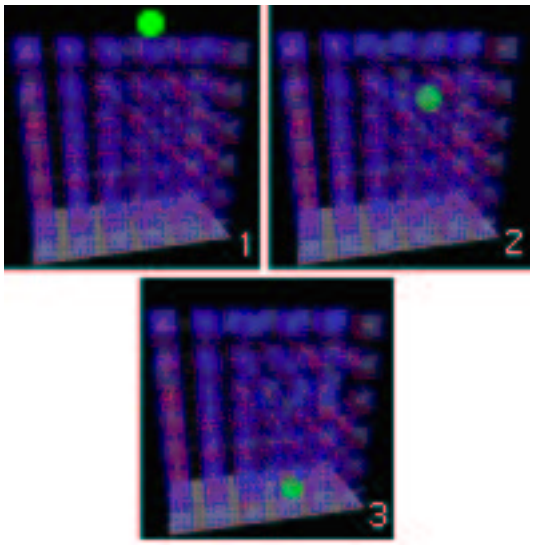

Figure 3: The membrane correctly passes an object.

Further trials on 200 test worlds with an 8 x 3 x 5 membrane and random object values and placements showed an approximately 98% generalization to unseen cases. All unsuccessful cases consisted of objects becoming incorrectly stuck in or on top of the membrane. This means the membrane had a 100% success rate in correctly rejecting objects. Furthermore, as the membrane was given more time, the overall success rate approached 100%. Thus, the membrane successfully filtered the undesirable objects and with a high probability let the desirable objects through.

In addition to showing good generalization to membranes of different sizes, the program also generalized reasonably well to multiple objects (Figure 4). Unfortunately, this generalization suffers from difficulties arising from close proximity desirable and undesirable objects. Specifically when two such objects are placed close together, within about 2 module widths of each other, undesirable objects can roll into gaps opened to allow desirable objects to pass through. Despite this difficulty, the resulting program performs reasonably well on multiple, adequately spaced objects.

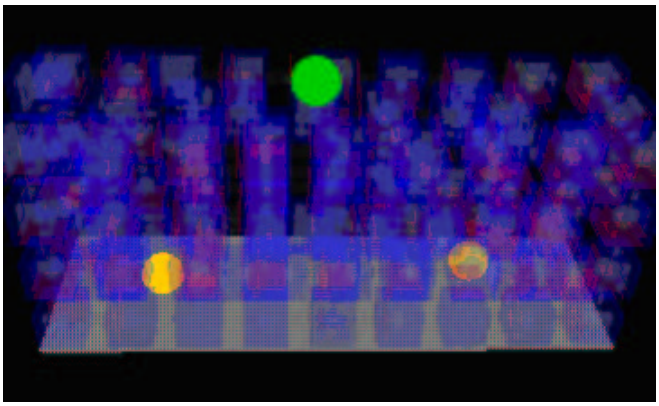

Figure 4: Multiple objects being filtered.

6 Binary Sorting Membrane Problem and Results

Sorting membranes accept all objects, but sort them as they pass through the membrane. Below we look at the case of a binary membrane, which sorts objects along a single direction into two distinct categories. This problem can be viewed as placing objects in two bins and is illustrated in Figure 5.

6.1 Fitness Function

The fitness was determined by running each module for 50 time steps on 30 test worlds with a 5 x 3 x 4 membrane. The exact locations of objects on top of the membrane and of object values were randomly generated once at the start of the GP run. In addition, a run was stopped if the object's Z coordinate became zero. In other words, the run stopped as soon as the object "fell out" of the membrane.

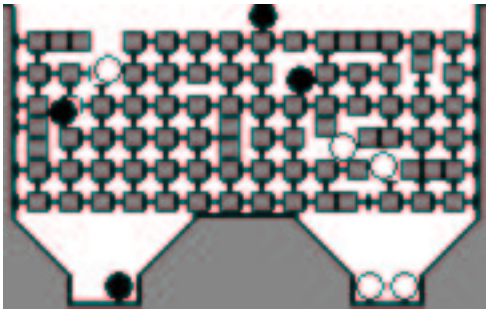

Figure 5: Example binary membrane.

The fitness function for the binary sorting membrane problem consisted of penalties for sorting the object incorrectly, failing to transport an object all the way through the membrane, and the membrane breaking apart. The penalty for incorrectly sorting was $(10{*}errX)$ where $errX$ is the distance between the object's X coordinate and the correct edge of the membrane to which the object was to be sorted. The penalty for failing to pass the object was $(10 * finalheight)$. The penalty for breaking apart was identical to that used for the filtering membrane.

6.2 Additional Primitives

The following functions were added:

- `(ReadSensorShouldPassB direction)` returns `true` if there exists an object in `direction` and the object is over the correct half of the membrane. This check is made by comparing the object's value with the module's X coordinate.
- `(ReadSensorObjectVal direction)` returns the value of the object in `direction` or `zero` if there is no such object.

The following terminals were also added:

- `(ReadSensorNextToObject)` returns `true` if the module is adjacent to an object.
- `(AMBObjectMoveDir)` returns a direction. If the module is adjacent to an object, this terminal returns the direction the object should be moved as determined by comparing the object's value with the module's X coordinate. If there are multiple adjacent objects, the closest one is used with ties broken randomly. If there is no such object, returns `0.0`.
- `(GetX)` returns the module's position [0,1] where 0 is the 0 value of the coordinate and 1 is the maximum value of the coordinate.

6.3 Results

The problem was run on twelve processors. A solution was found on generation 435.

```
(If (If (RetractArm 0.1) (If (If (Push
AMBObjectMoveDir) NormalizeDensity (Move
(ReadSensorObjectVal AMBObjectMoveDir))) (If
(RetractArm AMBObjectMoveDir) (If (If (Push
AMBObjectMoveDir) (RetractArm 1.0) (RetractArm
0.1)) (MoveAwayFrom AMBObjectMoveDir) (MoveAwayFrom
AMBObjectMoveDir)) (MoveAwayFrom AMBObjectMoveDir))
(MoveAwayFrom AMBObjectMoveDir)) (And (If
ReadSensorNextToObject NormalizeDensity
NormalizeDensity) (Push 0.0))) (If (If (ProgN
(ProgN (MoveAwayFrom (GetGradientVal 0.6))
ReadSensorNextToObject) ReadSensorNextToObject)
(Not (MoveAwayFrom AMBObjectMoveDir)) (And
ReadSensorNextToObject (And ReadSensorNextToObject
ReadSensorNextToObject))) NormalizeDensity
(If (Push 0.1) ReadSensorNextToObject (If (If
(MoveAwayFrom AMBObjectMoveDir) NormalizeDensity
ReadSensorNextToObject) (And (Push 0.8) (Push
AMBObjectMoveDir)) (If (Move (ReadSensorObjectVal
AMBObjectMoveDir)) ReadSensorNextToObject
(MoveAwayFrom AMBObjectMoveDir))))) (If (ProgN
NormalizeDensity NormalizeDensity) (If (If
(MoveAwayFrom AMBObjectMoveDir) NormalizeDensity
ReadSensorNextToObject) ReadSensorNextToObject
ReadSensorNextToObject) (And NormalizeDensity
ReadSensorNextToObject)))
```

The behavior induced by this program surprised us. Since objects occupy the same number of blocks as a module in the simulator, they can be supported either fully, by all four spaces below, or by only some of the four spaces underneath. Objects that are not fully supported "roll," if possible, one unit in the direction in which they are not supported. The membrane used this property to roll objects along the top of the membrane. Once an object reaches the edge of the membrane, the same underlying motion used to roll, opens a gap at the edge of the membrane. Further, since the membrane rolls the object along a single line of modules parallel to the X axis, the membrane can sort multiple objects as long as the objects are in different rows.

This program generalizes well, successfully sorting 100 of 100 objects with an 5 x 3 x 4 membrane, 100 of 100 objects with an 6 x 3 x 5 membrane, and 25 of 25 objects with an 8 x 5 x 7 membrane. All of these trials used random object values and placements.

Despite the success of the above program, rolling the object on top of the membrane has potential problems. Objects may roll off the membrane and not end up in the correct position. Since objects are largely only moving in the plane, collisions and deadlocks may be more likely limiting the number of objects that can be sorted at a single time. Finally, friction is not modeled in the simulator and may produce undesired effects. On the other hand, simply using the top of the membrane has the advantage that extremely thin membranes can sort: a membrane of thickness two performs as well as membranes of greater thickness.

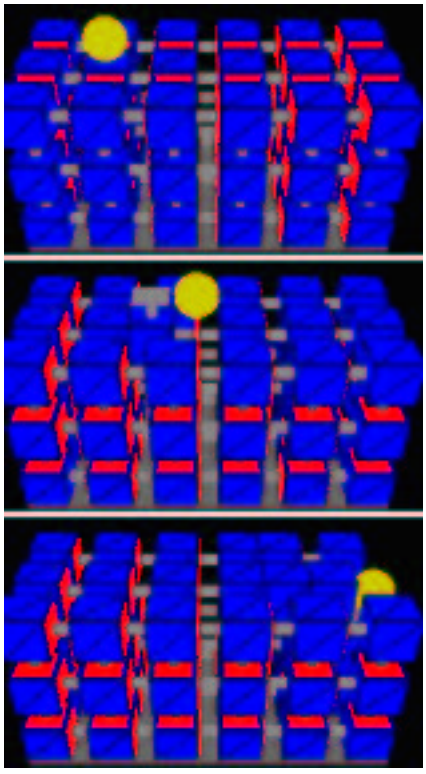

Figure 6: The membrane sorts an object.

For completeness we looked at evolving membranes that sort within the membrane.

6.4 Fitness Function: Second Trial

To attempt to evolve membranes that sort objects within the membrane, a penalty of 20 was added to the fitness function if the object was still on top of the membrane after the fourth time step.

6.5 Additional Primitives: Second Trial

Additional primitives, all terminals, were added to aid the finding of a robust solution.

- `(Drop)` runs a simplified version of the best filtering membrane solution described in section 5.3. Returns the result returned by that code.
- `(HalfThrough)` returns `true` if the module's $z < (h/2)$ where h is the height of the membrane.
- `(GetZ)` returns the module's position [0,1] where 0 is the 0 value of the coordinate and 1 is the maximum value of the coordinate.

The above solutions to both membrane problems do not use the message and gradient primitives as desired. Attempts at hand-coding solutions suggested restricting the type of gradients to positive and negative and including explicit gradient handling code to make it easier for the modules to use the gradients.

- `(SendPositiveGradient)` Emits a positive gradient, type 0.0, of value 2.
- `(SendNegativeGradient)` Emits a negative gradient, type 0.2, of value 2.
- `(HandleGradient)` Tries to follow the gradients. Returns `true` if a gradient was detected and the module was able to move to follow it.

Because explicit types of gradients were included, the following primitives were removed: `(GetMessage direction type)`, `(SendMessage direction type value)`, `(GetGradientVal type)`, `(GetGradientDir type)`, and `(SendGradient type val)`.

6.6 Results: Second Trial

The problem was run on 48 processors. The best solution was found on generation 702.

```
(If (If (If (Push AMBObjectMoveDir) (If (If
(RetractArm (Add 0.6 0.8)) (RetractArm 0.6)
(MoveAwayFrom 0.1)) (If (Push (LoadReg 0.4))
(RetractArm 0.6) (RetractArm 0.8)) (RetractArm 0.6))
(ProgN (Move 0.0) (Not (If (If SendNegativeGradient
(Push AMBObjectMoveDir) SendNegativeGradient)
(RetractArm 0.1) (ReadSensorIsObject 0.1))))) (If
(RetractArm GetZ) (Not ReadSensorNextToObject)
HalfThrough) (MoveAwayFrom 0.0)) (ProgN (ProgN
(ProgN (ProgN (RetractArm 0.4) (RetractArm 0.1)) (If
(And (ProgN (RetractArm 0.6) SendNegativeGradient)
(Push 0.8)) (If (RetractArm GetZ) (Not (LT
RandDir 1.0)) (RetractArm 1.0)) (MoveAwayFrom
AMBObjectMoveDir))) (Move (Sub (ProgN 0.1 0.6) (Sub
0.6 0.2)))) (If (MoveAwayFrom AMBObjectMoveDir)
NormalizeDensity (ProgN (ReadSensorIsObject
1.0) (If ReadSensorNextToObject (MoveAwayFrom
AMBObjectMoveDir) (MoveAwayFrom (If Drop 0.6
RandDir)))))) (If (Push (ReadSensorObjectVal 1.0))
(If (ProgN SendNegativeGradient NormalizeDensity)
(RetractArm AMBObjectMoveDir) (ProgN (Move
0.0) (Not (If (Push AMBObjectMoveDir) (Push
0.8) (ReadSensorIsObject 0.1))))) (If (Move
AMBObjectMoveDir) SendNegativeGradient (If (If
(MoveAwayFrom AMBObjectMoveDir) NormalizeDensity (If
SendNegativeGradient (Push 0.0) (ProgN (RetractArm
0.6) (RetractArm 0.1)))) (ProgN SendPositiveGradient
NormalizeDensity) (MoveAwayFrom (ReadSensorObjectVal
(Sub 0.4 GetZ)))))))
```

The behavior of the program is as originally expected. Objects move diagonally through the membrane in the correct direction. Unfortunately, this behavior did not generalize as well as the rolling behavior. On 200 random test runs, the membrane preformed at `97%` success on 5 x 3 x 4 membranes and `98.5%` success on 6 x 3 x 5 membranes.

It is interesting to note that the code above does not use communication between modules. While the code contains instructions for sending gradients, it does not include code for receiving or handling them.

7 Conclusions and Future Work

We have exhibited decentralized control software that enables a modular robotic membrane to accomplish sorting and filtering tasks. Genetic programming was effective in automatically generating such hard-to-write software, and proved to generate software robust to varying the numbers of modules and objects.

We plan to place the software on actual hardware to see how it performs once a sufficient number of modules have been built. It is likely that we would also need to enhance the simulator with more detailed properties of the actual physical robot, but given the robustness of the method to different module styles, we expect the method can generate working software for actual physical hardware. To support this belief, experiments should be done in simulation to generate software that is robust against module failures and inaccuracies in sensors or motion.

Also of interest is extending the work to yet more complicated sorting tasks, such as sorting objects along a continuous line, into any number of bins, or along two perpendicular directions. We could also increase the complexity of the features according to which a membrane sorts. Creating a membrane that sorts objects by shape would be of particular interest.

8 Acknowledgements

We would like to thank: Forrest Bennett for suggestions on the membrane problem, as well as help with FXPAL's GP system and computing cluster; Brad Dolin for help in setting up the GP runs and suggestions on the membrane problem; Sergei Vassilvitskii for help in creating the simulator; Wolfgang Polak for help in setting up the computing cluster; and Adil Qureshi for Gpsys Java 1.1 GP source code on which FXPAL's GP system is based.

References

[1] Bennett III, F.H., Dolin, B., Rieffel, E.G., Programmable Smart Membranes: Using Genetic Programming to Evolve Scalable Distributed Controllers for a

Novel Self-Reconfigurable Modular Robotic Application, Genetic Programming: Proceedings of EuroGP 2001, Springer LNCS 2038 234 - 245

[2] Bennett III, F.H., Rieffel, E.G.: Design of Decentralized Controllers for Self-Reconfigurable Modular Robots Using Genetic Programming. Proceedings of the 2nd NASA/DoD Workshop on Evolvable Hardware (2000) 43-52

[3] Bojinov, H. Casal A., Hogg, T.: Emergent Structures in Modular Self-Reconfigurable Robots. IEEE Int. Conf. On Robotics and Automation (2000) 1734-1741

[4] Haynes, T., Wainwright R., Sen, S., Schoenfeld, D.: Strongly Typed Genetic Programming in Evolving Cooperation Strategies. Proceedings of the Sixth International Conference on Genetic Algorithms. Morgan Kaufmann (1995) 271-278

[5] Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)

[6] Kubica, J., Casal, A., Hogg, T.: Agent-based Control for Object Manipulation with Modular Self-reconfigurable Robots. International Joint Conference of Artificial Intelligence (2001) 1344-1352

[7] Kubica, J., Rieffel, E.: Collaborating with a Genetic Programming System to Generate Modular Robotic Code. Submitted to GECCO 2002.

[8] Pamecha, A., Ebert-Uphoff, I., Chirikjian, G. S.: Design and Implementation of Metamorphic Robots. Proceedings 1996 ASME Design Engineering Tehcnical Conference and Computers and Engineering Conference. ASME Press (1996) 1 - 10.

[9] Rus, D., Vona, M.: Crystalline Robots: Self-reconfiguration with Compressible Unit Modules. Autonomous Robots. January 2001. 10 (1): 107-124.

[10] Rus, D., Vona, M.: Self-Reconfiguration Planning with Compressible Unit Modules. Proceedings of the 1999 IEEE Int. Conf. On Robotics and Automation 2513-2520.

[11] Rus, D. Vona, M.: A Physical Implementation of the Crystaline Robot. Proceedings of the 2000 IEEE Int. Conference of Robotics and Automation.

[12] Shen, W.M., Salemi, B., Lu, Y., Will, P.: Hormone-Based Control for Self-Reconfigurable Robots. 2000 International Conference on Autonomous Agents. Barcelona, Spain.

[13] Suh, J.W., Homans, S.B., Yim, M.: Telecubes: Mechanical Telecubes: Mechanical Design of a Module for Self-Reconfigurable Robotics. To appear in IEEE Int. Conference of Robotics and Automation (2002)

[14] Vassilvitskii, S., Kubica, J., Rieffel, E., Suh, J., and, Yim, M.: On the General Reconfiguration Problem for Expanding Cube Style Modular Robots. To appear in IEEE Int. Conference of Robotics and Automation (2002)