Humanoid Learns to Detect Its Own Hands

Jürgen Leitner*, Simon Harding†, Mikhail Frank*, Alexander Förster*, Jürgen Schmidhuber*

*Dalle Molle Institute for Artificial Intelligence (IDSIA)
Faculty of Informatics, Università della Svizzera Italiana (USI)
CH-6928 Manno-Lugano, Switzerland
Email: juxi@idsia.ch

†Machine Intelligence Ltd
South Zeal, United Kingdom
Email: simon@machineintelligence.co.uk

***Abstract*—Robust object manipulation is still a hard problem in robotics, even more so in high degree-of-freedom (DOF) humanoid robots. To improve performance a closer integration of visual and motor systems is needed. We herein present a novel method for a robot to learn robust detection of its own hands and fingers enabling sensorimotor coordination. It does so solely using its own camera images and does not require any external systems or markers. Our system based on Cartesian Genetic Programming (CGP) allows to evolve programs to perform this image segmentation task in real-time on the real hardware. We show results for a *Nao* and an *iCub* humanoid each detecting its own hands and fingers.**

I. Introduction

Although robotics has seen advances over the last decades robots are still not in wide-spread use outside industrial applications. Yet proposed scenarios range from cleaning tasks, grocery shopping to elderly care and helping in hospitals, etc. These involve the robots working together, helping and coexisting with humans in daily life. From this the need to manipulate objects in unstructured environment arises. Yet autonomous object manipulation is still a hard problem in robotics. Humans, in contrast, are able to quickly and without much thought, perform a variety of object manipulation tasks on arbitrary objects. To achieve this, Visual Motor Integration (or visuomotor integration, VMI), also referred to as eye-hand coordination, is of importance. For example, if someone wants to grab a certain object from the table, the brain takes in the location of the object as perceived by the eyes, and uses this visual information to guide the arm. To perform this guidance both the object and the arm need to be detected visually.

In robotics traditionally a multi-step approach is used to reach for an object. First the object's position is determine in operational space. If there are no active vision systems (RGB-D cameras/Kinect) or external trackers/markers (Vicon) available, the most common approach is using a stereo camera pair. After detecting the object in the image, projective geometry is used to estimate the operational space coordinate [1]. This is the desired position to be reached with the end-effector. The joint configuration of the robot for reaching this position is determined by applying inverse kinematics techniques [2]. The obvious problem in this chain of operations is that errors are propagated and sometimes amplified through these transformations. Visual servoing [3], [4] has been proposed to overcome some of these problems. It is using the distance between of the object and the arm in the visual plane as the control signal.

It stands to reason that humanoids require a certain level of visuomotor coordination to develop better manipulation skills. This is similar to the development in studying of human visual and motor processing, which have previously been investigated separately, but more evidence emerged that those two systems are very closely related [5]. For such an approach robust visual perception, such as, e.g. precisely and quickly detecting the hands, is of importance. This is especially of interest in highly-complex robots such as humanoids, as generally there is no precise knowledge about the kinematic structure (body) of the robot available, generating significant errors when transforming back and forth into operational space.

In this paper we present a novel method for humanoid robots to visually detect their own hands. We use a machine learning technique called Cartesian Genetic Programming (CGP) to learn a robust method to filter out the hands and fingers in the camera stream from an *iCub* and a *Nao* robot. This opens up future research into sensorimotor coordination on humanoid robot platforms. This is the first time, to our knowledge, for a robot to learn how to detect its own hands, from vision alone.

II. Related Work

The detection of hands in robotics has so far been mainly focused on detecting human hands in camera images. For example, Kölsch and Turk presented a method for robustly classifying different hand postures in images [6]. These approaches usually use the quite uniform colour of the skin [7], motion flow [8] and particle filtering [9].

Hand-eye coordination for robotics has previously been investigated using extra sensors like LASERs or other helpers, such as, LEDs, bright coloured symbols, etc., mounted on the end-effector. Langdon and Nordin, already explored GP for evolving hand-eye coordination on a robot arm with a LED on the end-effector [10]. Hülse et al. used machine learning to grasp a ball with a robot arm, yet only the ball, not the arm was visually detected [11]. Another often used option to track and detect the position of the robots end-effector are external motion capturing and imaging systems. For example, [12], [13].

Recently DLR has investigated methods for better position (and orientation) estimation for their humanoid robot *Justin*. An approach combining a RGB-D (Kinect) and a model-based approach was described and was able to qualitatively decrease the error from the purely kinematic solution [14].

Machine learning has been used in computer vision previously but has mainly focussed on techniques such as Support Vector Machines (SVM), k-Nearest Neighbour (kNN) and Artificial Neural Networks (ANN). Herein we use Genetic Programming (GP) which is a search technique inspired by concepts from Darwinian evolution [15]. It can be used in many domains, but is most commonly used for symbolic regression and classification tasks. GP has also been used to solve problems in image processing, however previous attempts typically use a small set of mathematical functions to evolve kernels, or a small number of basic image processing functions (such as erode and dilate). Previously Spina used GP to evolve programs performing segmentation based on features calculated from partially labelled fore- and background [16].

Given the maturity of the field of image processing, it should be possible to construct programs that use much more complicated image operations and hence incorporate domain knowledge. For example, Shirakawa evolved segmentation programs that use many high-level operations such as mean, maximum, Sobel, Laplacian and product [17].

III. Cartesian Genetic Programming for Image Processing

Herein we are using Cartesian Genetic Programming (CGP), in which programs are encoded in partially connected feed forward graphs [18], [19]. The genotype, given by a list of nodes, encodes the graph. For each node in the genome there is a vertex, represented by a function, and a description of the nodes from where the incoming edges are attached.

The basic algorithm works as follows: Initially, a population of candidate solutions is composed of randomly generated individuals. Each of these individuals, represented by its genotype, is tested to see how well it performs the given task. This step, known as evaluating the 'fitness function', is used to assign a numeric score to each individual in the population. Generally, the lower this error, the better the individual.

In the next step of the algorithm, a new population of individuals is generated from the old population. This is done by taking pairs of the best performing individuals and performing functions analogous to recombination and mutation. These new individuals are then tested using the fitness function. The process of test and generate is repeated until a solution is found or until a certain number of individuals have been evaluated.

CGP offers some nice features, for instance, not all of the nodes of a solution representation (the genotype) need to be connected to the output node of the program. As a result there are nodes in the representation that have no effect on the output, a feature known in GP as 'neutrality'. This has been shown to be very useful [20] in the evolutionary process. Also, because the genotype encodes a graph, there can be reuse of nodes, which makes the representation distinct from a classically tree-based GP representation (Fig. 1).

Our implementation CGP for Image Processing (CGP-IP) draws inspiration from much of the previous work in the field (see e.g. [21]). It uses a mixture of primitive mathematical and high level operations. It's main difference to previous implementation is that it encompasses domain knowledge, i.e. it allows for the automatic generation of computer programs using a large subset of the OpenCV image processing library functionality [22]. With over 60 unique functions, the function set is considerably larger than those typically used with GP. This does not appear to hinder evolution, and we speculate that the increased number of functions provides greater flexibility in how the evolved programs can operate.

Executing the genotype is a straightforward process. First, the active nodes are identified. This is done recursively, starting at the output node, and following the connections used to provide the inputs for that node. In CGP-IP the final node in the genotype is used as the output. Next, the phenotype can be executed on an image. The input image (or images) are used as inputs to the program, and then a forward parse of the phenotype is performed to generate the output.

The efficacy of this approach was shown for several different domains (including basic image processing, medical imaging, terrain classification, object detection in robotics and defect detection in industrial application) by Harding et al. [23] and Leitner et al. [24].

A. Fitness Function

Depending on the application, different fitness functions are available in CGP-IP. The thresholded output image of an individual is compared to a target image using the Matthews Correlation Coefficient (MCC) [25], [26], which has previously been observed to be useful for classification problems solved using CGP [27]. The MCC is calculated based on the 'confusion matrix', which is the count of the true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN).A coefficient of 0 indicates that the classifier is working no better than chance. A score of 1 is achieved by a perfect classifier, -1 indicates that the classifier is perfect, but has inverted the output classes. Therefore, the fitness of an individual

$$fitness = 1 - |c| \tag{1}$$

with values closer to 0 being more fit.

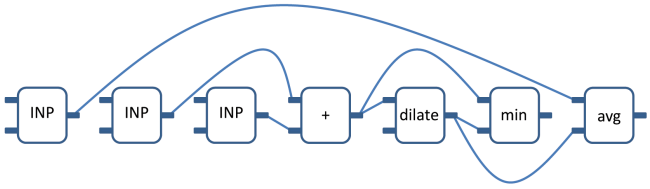

Fig. 1. Example illustration of a CGP-IP genotype. Internally each node is represented by several parameters. In this example, the first three nodes obtain the image components from the current test case (i.e. a grey scale representation and the red and green channels). The fourth node adds the green and red images together. This is then dilated by the fifth node. The sixth node is not referenced by any node connected to the output (i.e. it is neutral), and is therefore ignored. The final node takes the average of the fifth node and the grey scale component from the current test case.

B. High Level Operations

Previous work on imaging processing with GP operates on a convolutional approach. Here, a program is evolved that operates as a kernel. For each pixel in an image, the kernel operates on a neighbourhood and outputs a new pixel value. This is also the typical approach when other machine learning approaches are applied to imaging. In GP, the kernel is typically an expression composed from primitive mathematical operators such as $+, -, \times$ and $\div$. For example, this approach was used in [28], [29], [30] to evolve noise reduction filters. In [31], many different image processing operations (e.g. dilate, erode, edge detection) were reverse-engineered.

The function set not only contains high-level image processing functions, but also primitive mathematical operations. In CGP-IP a large number of commands from the OpenCV library are available to GP. Additionally, higher level functions, such as, Gabor filtering are available. A complete list can be found in [23]. The primitive operators also work on entire images i.e. addition will produce a new image adding the values of corresponding pixels from two input images. However, this method does not directly allow for kernel-style functionality to be found. Instead, GP has to use a combination of shifts and rotations and other existing kernel methods to get information about a pixel's neighbourhood. This is similar to the methods proposed in [31] to allow for efficient parallel processing of images on Graphics Processing Units (GPUs).

Using OpenCV we can also be confident about using high quality, high speed software. In CGP-IP, individuals are evaluated at the rate of 100s per second on a single core. This makes it both efficient to evolve with, but also means that the evolved filters will run quickly if deployed. Much of the previous work on imaging with GP has focused on the use of grey scale images. Often this is for performance considerations. But also this is out of consideration for how the images will be handled within the program. In CGP-IP, all functions operate on single channel images. The default treatment for colour images is to separate them into both RGB (red, green and blue) and HSV (hue, saturation and value) channels, and provide these as available inputs. A grey scale version of the image is also provided. Each available channel is presented as an input to CGP-IP, and evolutions selects which inputs will be used.

C. Pruning Generated Programs

Our implementation of CGP-IP generates human readable C# or C++ code based on OpenCV. Although CGP offers bloat free evolution, it often appears to leave redundant nodes in the evolved program. Examples of these include operations that add 0 to every value, or produce a uniformly black image. Whilst the neutrality is important in the genotype, in the phenotype (i.e. the evolved program) it is undesirable.

To optimise the generated code, we implemented a method of pruning that identifies unnecessary operations. The process of pruning a program is as follows: for each active node in the CGP program, replace it with a NOP and re-run the fitness evaluation. If the fitness does not change (or improves), leave the modification in place, and move to the next node. If the fitness degrades, replace the instruction with a node that generates a completely black image, and retest. Again, keep the change if it does not affect the program's output. If it does alter the fitness, a final substitution attempt is made with a white image. If all of these changes degrade performance, the original operation is restored. It is typically found that this process reduces the number of used instructions, and hence reduces the execution time of the evolved program.

D. CGP Parameters

A feature generally with CGP implementations is the low number of parameters required for configuration. The same is valid for CGP-IP. The main parameters are:

- Graph length (i.e. the number of nodes in the genotype), set to 50 here.
- Mutation rate, 10% of all genes in the graph are mutated when an offspring is generated. The threshold parameter is mutated with a probability of 1%.
- Size of mutations
- Number of islands, this depends on the available computing resources. CGP-IP has been tested successfully from 1 to 24 islands.
- Number of individuals per island, which is set to 5, keeping the typical 1+4 evolutionary strategy.
- Synchronisation interval between islands. Here each island compares their best individual to the servers individual every 10 generations.

It is important to note that in the work presented here the parameters have not been optimized other than by casual experimentation. It may be possible to improve the performance of CGP-IP by more carefully selecting these parameters. In particular, we would expect the mutation rate, genotype size and number of islands to be the most important parameters to adjust.

CGP-IP needs a number of additional parameters encoded in each node, compared to classical CGP. They are listed in Table I. These are needed because often the functions used require additional parameters, with specific requirements as to their type and range. Connection 0 and 1 contain the relative address of the node used as first and second input. If a relative address extends beyond the extent of the genome it is connected to one of the inputs. Specialised 'Input' functions are also provided (e.g. INP, SKIP), which manipulate a pointer that indexes the available inputs and return the currently indexed input. A full description can be found in [32]. An illustrative example is shown in Fig. 1. In addition to the graph representing the program, the genotype also includes a value used for thresholding the output.

TABLE I. PARAMETERS OF CGP-IP ENCODED AT EVERY NODE.

Parameter	Type	Range
Function	Int	# of functions
Connection 0	Int	# of nodes and inputs
Connection 1	Int	# of nodes and inputs
Parameter 0	Real	no limitation
Parameter 1	Int	$[-16, +16]$
Parameter 2	Int	$[-16, +16]$
Gabor Filter Frequ.	Int	$[0, 16]$
Gabor Filter Orient.	Int	$[-8, +8]$

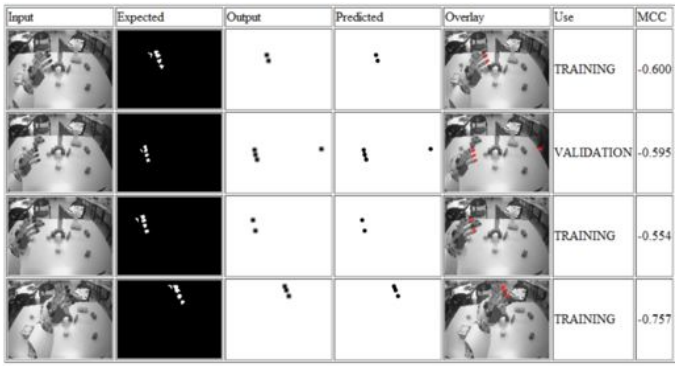

Fig. 2. The various stages of the supervised learning approach. On the left the (grey-scaled) input image, followed by the hand-segmented output (ground truth). The next column shows the output of the current individual using this specific input image, followed by the thresholded output (Expected). The overlay allows a quick visual verification by overlaying the output (in red) over the grey-scaled input image.

All parameters are kept constant throughout the experiments presented below. Again, whilst it may be possible to improve performance for a given problem by optimising the parameters, we believe that the apparent parameter robustness is an important feature of this technique.

E. Training

A handful of examples, in which the object of interest has been correctly segmented, in our case the fingers and hands, are used as a training set for CGP-IP. The selection of this training set is of importance for the capabilities of the found solution. If chosen correctly, the resulting programs are very robust filters. In Fig. 2 the various stages are shown. The original image is used to hand-label points of interest (in this case the finger tips). This is then used as a training set for CGP-IP. The output of the filter is shown in the third column and is used again, just for visualisation, in the last column as an overlay on the grey-scale input image.

An example of the human-readable output, in the form of a C++ computer program, is shown in Listing 1. On a single core of a modern desktop PC, the evolved programs run quickly and as these are largely linear in form, the memory requirements are low. Speed and simplicity of processing is important in many embedded systems, making this approach suitable for implementation in constrained computing environments.

IV. Experiments

For testing our system we employed two humanoid robot platforms. Both these systems are quite different both in their appearance and the sensor systems available. Our first platform is the *Nao* humanoid built by Aldebaran Robotics, our second robot is the *iCub* humanoid, which was developed during the European funded RoboCub project and is currently actively used in a variety of projects both at European and international level. In each of these robots we aim for the humanoid to learn how to detect its own hand and fingers, while moving the arm through the visible workspace.

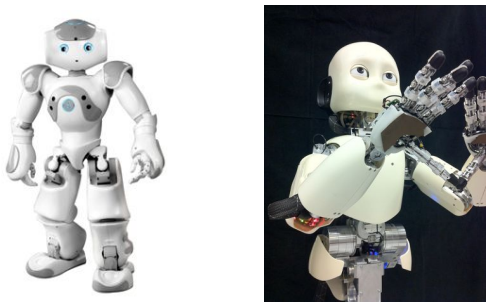

Fig. 3. Our robotic platforms: the Nao (left) and the *iCub* (right).

A. Nao Hand Detection

Our first experiment is performed with the *Nao* humanoid by Aldebaran Robotics (see Fig. 3). The 53cm tall robot has 25 DOF (including the actuated hands), an inertial measurement unit (with accelerometer and gyrometer), four ultrasonic sensors, eight force-sensing resistors and two bumpers. It also features four microphones, two speakers, and two cameras — one looking forward (forehead placement) and one looking down (chin placement).

A dataset of pictures was collected from both of the cameras (though the hand was more often visible in the lower camera, due to its placement). Fig. 4 shows some of these pictures. When collecting these the hand was randomly moved through the field of view of the robot. We apply our CGP-IP approach described above to detect the robots fingers and hands. A hand-labeled training set was generated and used (as described above; similar to Fig. 2).

An filter was found after short evolution. Fig. 5 shows the result of one of the top-performing filters for this task. The output of the filter is again used as an overlay on the grey-

Fig. 4. Some of the images in the dataset for the Nao robot.

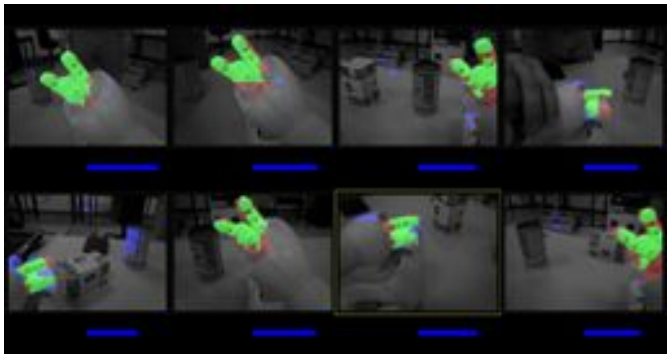

Fig. 5. One of the results, when learning to detect the hands of the *Nao* robot. In green the areas that are correctly detected, red areas that were marked in the supervised training set yet have not been detected by this individual, blue are areas that are wrongly detected to be part of the robot's hand.

```
icImage NaoFingers::runFilter() {
  icImage node0 = InputImages[3].min(InputImages[4]);
  icImage node1 = InputImages[1].unsharpen(11);
  icImage node2 = node0.Max();
  icImage node3 = node0.gauss(7, -7);
  icImage node4 = node2.addc(6.71329411864281);
  icImage node5 = InputImages[0].mul(node1);
  icImage node6 = node3.dilate(6);
  icImage node7 = node5.sobely(15);
  icImage node9 = node4.absdiff(node6);
  icImage node10 = InputImages[5];
  icImage node13 = node10.unsharpen(15);
  icImage node14 = node7.dilate(6);
  icImage node17 = node9.gauss(11);
  icImage node18 = node17.SmoothBlur(11);
  icImage node19 = node13.min(node14);
  icImage node27 = node19.SmoothBlur(11);
  icImage node31 = node18.Exp();
  icImage node32 = node31.erode(4);
  icImage node37 = node27.mul(node32);
  icImage node38 = node37.ShiftDown();
  icImage node41 = node38.ShiftDown();
  icImage node49 = node41.laplace(19);
  return node49;
}
```

Listing 1. Generated C# code from CGP-IP for detecting the Nao's fingers. `icImage` is a wrapper class to allow portability within our framework [33].

scale images. From the 8 inputs one was selected as validation image (highlighted by a golden frame). The performance of the filter is rather good, with green indicating a correct detection. Red shows areas that are, according to the provided input labels, part of the hand, but not detected by our filter. The hand was cleanly detected in each of the 8 images and with very little (pixel-wise) error. The generated program to perform this segmentation is shown in Listing 1.

B. iCub *Finger and Hand Detection*

The second set of experiments, are performed with the *iCub* [34], an open-system robotic platform, providing a 41 degree-of-freedom (DOF) upper-body, comprising two arms, a head and a torso (see Fig. 3). The *iCub* is generally considered an interesting experimental platform for cognitive and sensorimotor development and embodied Artificial Intelligence [35] research, with a focus on object manipulation. To allow for manipulation, the object of interest has to first be detected and located using the available sensors. The robot has to rely, on a visual system consisting of two cameras in the head, analogous to human visual perception. The errors during this localisation, as well as, in the kinematic model are significant. Therefore we aim to use visual feedback to perform better reaching and grasping. In comparison to the NAO hand the *iCub* sports a rather complex end-effector. The hand and fingers are both mechanically complicated and visually hard to detect (Fig. 6).

Fig. 6. The *iCub*'s hands and fingers are quite complicated mechanically and complex to detect visually.

We again apply our CGP-IP approach to detect the *iCub*'s fingers and hands, a necessity to learn hand-eye coordination. As in the section above, we collect a dataset while the robot was moving its arms around. A handful of pictures was hand-labelled and used as a training set. The detection of the hand is split into two separate problems: the identification of the fingertips only, and, secondly of the full hand. This separation allows us to create a finer level of control for tasks, such as, grasping, that involve controlled motion of the fingers.

The fingertips are made out of a black rubber protecting the touch sensors within. Fig. 7 shows the result of an evolved filter detecting the fingertips of the *iCub*'s hands. To make sure that the solution is not just identifying any black part the images contain black objects in the background, such as, chairs and computer cases. The solution for filtering out the fingertips is found rather quickly, after only a bit more than 8 minutes of evolution (about 55000 individual evaluations).

To detect the full hand is a more complicated task. Here we used 10 images to train the filter and an additional four to validate each solution against unseen data. For this more complicated task the evolution took more than 5700000 evaluations (about a hundred times more than for the fingers alone), taking about 12 hrs on a desktop computer. In Fig. 8 the performance of the learnt filter is shown. Generally a very good detection can be seen, although there are some minor issues. For example, a reduced level of detection can be seen around the edges of the frame. As we will be gazing upon a specific object we want to manipulate, the precise control will be of importance when the hand reaches close to the centre of the image frame. Another issue seems to be the stark difference in visual appearance between the front and back side of the hand. This can be seen in the second to the right image in the middle row of the figure. Again in most of the planned scenarios for object manipulation the back of the hand will be visible. Therefore we do not see this as an important issue. Furthermore if a more thorough detection, also of the front side, is required another separate filter just for the front can be evolved.

By combining the two separate detectors, for the fingertips and the hand, we get a very precise, almost full identification of the *iCub*'s hands.

Fig. 7. Results when visually locating the fingertips of the *iCub*.

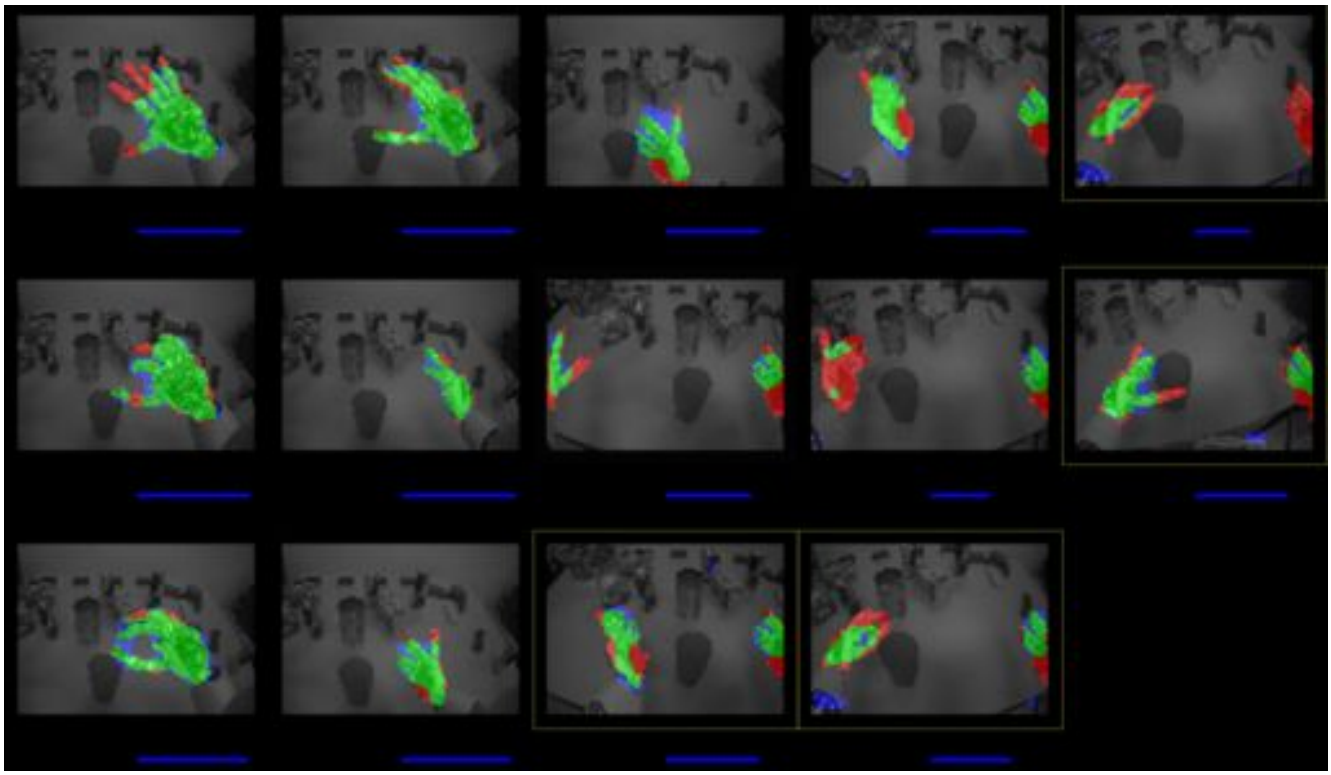

Fig. 8. Results when visually locating the fingertips of the *iCub* with an evolved filter. The images with golden borders are for validation. Green shows the correct detection; red indicates parts that were not detected by our filter (false negative); blue indicates where the filter falsely detected the hand (false positive).

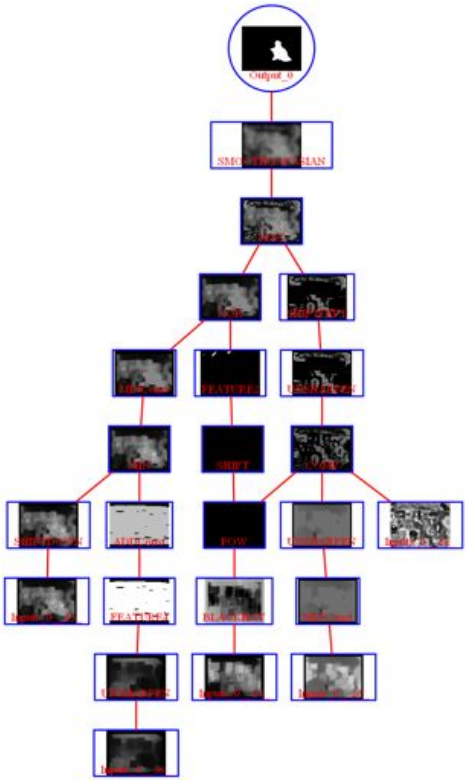

Fig. 9. The operations performed by the filter to detect the *iCub*'s hands. The (binary) segmentation of the hands is the top-most image.

The programs to perform this segmentation are generated the same way as for the NAO and also provide C# code. The complexity though is higher, providing an indication that the detection of the *iCub*'s hands is a more complex task than the NAO's. The complexity of the filter can also be visualised by plotting the operations and their connections of the found CGP-IP solution (Fig. 9).

V. Conclusion

We presented a novel approach for a robot to detect its own hands and fingers using vision by applying Cartesian Genetic Programming (CGP). Detecting hands in images has previously focused mainly on detecting human hands. Allowing the robot to detect its own hands in real-time enables hand-eye coordination, in tasks, such as, reaching for and picking-up objects.

Whilst the results are hard to compare with other published work, due to lack of suitable datasets, the results indicate that CGP-IP is highly competitive. Further, we have seen that CGP-IP can work with well-known image processing operations and generate human readable programs.

REFERENCES

[1] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision*, 2nd ed. Cambridge University Press, 2000.

[2] R. P. Paul, *Robot manipulators: mathematics, programming, and control*. MIT press, 1981.

[3] S. Hutchinson, G. D. Hager, and P. I. Corke, "A tutorial on visual servo control," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 5, pp. 651–670, 1996.

[4] F. Chaumette and S. Hutchinson, "Visual servo control, Part I: Basic approaches," *IEEE Robotics and Automation Magazine*, vol. 13, no. 4, pp. 82–90, 2006.

[5] M. Goodale, "Visuomotor control: Where does vision end and action begin?" *Current Biology*, vol. 8, no. 14, p. 489, 1998.

[6] M. Kölsch and M. Turk, "Robust hand detection," in *Proc. of the Intl. Conference on Automatic Face and Gesture Recognition*, 2004, p. 614.

[7] X. Zhu, J. Yang, and A. Waibel, "Segmenting hands of arbitrary color," in *Proc. of the Intl. Conference on Automatic Face and Gesture Recognition*, 2000, pp. 446–453.

[8] R. Cutler and M. Turk, "View-based interpretation of real-time optical flow for gesture recognition," in *Proc. of the Intl. Conference on Automatic Face and Gesture Recognition*, 1998, pp. 416–421.

[9] M. Isard and A. Blake, "Condensationconditional density propagation for visual tracking," *International journal of computer vision*, vol. 29, no. 1, pp. 5–28, 1998.

[10] W. B. Langdon and P. Nordin, "Evolving Hand-Eye Coordination for a Humanoid Robot with Machine Code Genetic Programming," in *Proc. of the European Conference on Genetic Programming (EuroGP)*, 2001.

[11] M. Hulse, S. McBrid, and M. Lee, "Robotic hand-eye coordination without global reference: A biologically inspired learning scheme," in *Proc. of the Intl. Conference on Developmental Robotics*, 2009.

[12] I. Oikonomidis, N. Kyriazis, and A. Argyros, "Efficient model-based 3d tracking of hand articulations using kinect," in *Proc. of the British Machine Vision Conference"*, 2011.

[13] Y. Ehara, H. Fujimoto, S. Miyazaki, S. Tanaka, and S. Yamamoto, "Comparison of the performance of 3d camera systems," *Gait & Posture*, vol. 3, no. 3, pp. 166–169, 1995.

[14] O. Porges, K. Hertkorn, M. Brucker, and M. A. Roa, "Robotic hand pose estimation using vision," Poster Session at the IEEE RAS Summer School on "Robot Vision and Applications", 2012.

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

[16] T. V. Spina, J. A. Montoya-Zegarra, A. X. Falcao, and P. A. V. Miranda, "Fast interactive segmentation of natural images using the image foresting transform," in *Proc. of the Intl. Conference on Digital Signal Processing*, 2009, pp. 1–8.

[17] S. Shirakawa and T. Nagao, "Feed forward genetic image network: Toward efficient automatic construction of image processing algorithm," in *Proc. of the Intl. Symposium on Visual Computing*, ser. Lecture Notes in Computer Science, vol. 4842. Springer, 2007, pp. 287–297.

[18] J. F. Miller, "An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach," in *Proc. of the Genetic and Evolutionary Computation Conference*, 1999, pp. 1135–1142.

[19] J. F. Miller, Ed., *Cartesian Genetic Programming*, ser. Natural Computing Series. Springer, 2011.

[20] J. F. Miller and S. L. Smith, "Redundancy and computational efficiency in cartesian genetic programming," in *IEEE Transactions on Evoluationary Computation*, vol. 10, 2006, pp. 167–174.

[21] L. Sekanina, S. L. Harding, W. Banzhaf, and T. Kowaliw, "Image processing and CGP," in *Cartesian Genetic Programming*, ser. Natural Computing Series, J. F. Miller, Ed. Springer, 2011, ch. 6, pp. 181–215.

[22] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[23] S. Harding, J. Leitner, and J. Schmidhuber, "Cartesian genetic programming for image processing," in *Genetic Programming Theory and Practice X (in press)*. Springer, 2013.

[24] J. Leitner, S. Harding, A. Förster, and J. Schmidhuber, "Mars Terrain Image Classification using Cartesian Genetic Programming." in *Proc. of the Intl. Symposium on AI, Robotics and Automation in Space*.

[25] B. W. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme." *Biochimica et Biophysica Acta*, vol. 405, no. 2, pp. 442–451, 1975.

[26] Wikipedia, "Matthews correlation coefficient — wikipedia, the free encyclopaedia," 2012, [accessed 21-March-2012]. [Online]. Available: `http://en.wikipedia.org/w/index.php?title=Matthews_correlation_coefficient&oldid=481532406`

[27] S. Harding, V. Graziano, J. Leitner, and J. Schmidhuber, "Mt-cgp: Mixed type cartesian genetic programming," in *Proc. of the Genetic and Evolutionary Computation Conference*, 2012.

[28] S. Harding and W. Banzhaf, "Genetic programming on GPUs for image processing," *International Journal of High Performance Systems Architecture*, vol. 1, no. 4, pp. 231–240, 2008.

[29] S. L. Harding and W. Banzhaf, "Distributed genetic programming on GPUs using CUDA," in *Workshop on Parallel Architectures and Bioinspired Algorithms*, 2009, pp. 1–10.

[30] T. Martínek and L. Sekanina, "An evolvable image filter: Experimental evaluation of a complete hardware implementation in fpga," in *Proc. of the Intl. Conference Evolvable Systems: From Biology to Hardware*, 2005, pp. 76–85.

[31] S. Harding, "Evolution of image filters on graphics processor units using cartesian genetic programming," in *Proc. of the World Congress on Computational Intelligence*, 2008, pp. 1921–1928.

[32] S. Harding, W. Banzhaf, and J. F. Miller, "A survey of self modifying cartesian genetic programming," in *Genetic Programming Theory and Practice VIII*, R. Riolo, T. McConaghy, and E. Vladislavleva, Eds. Springer, 2010, vol. 8, pp. 91–107.

[33] J. Leitner, S. Harding, M. Frank, A. Förster, and J. Schmidhuber, "An integrated, modular framework for computer vision and cognitive robotics research (icvision)," in *Biologically Inspired Cognitive Architectures 2012*, A. Chella, R. Pirrone, R. Sorbello, and K. R. Jhannsdttir, Eds. Springer, 2013, vol. 196, pp. 205–210.

[34] N. G. Tsagarakis *et al.*, "iCub: the design and realization of an open humanoid platform for cognitive and neuroscience research," *Advanced Robotics*, vol. 21, pp. 1151–1175, Jan. 2007.

[35] G. Metta *et al.*, "The iCub humanoid robot: An open-systems platform for research in cognitive development," *Neural Networks*, vol. 23, no. 8-9, pp. 1125–1134, Oct. 2010.