GPTIPS

Genetic Programming & Symbolic Regression for MATLAB

User Guide

26 November 2009 v1.0

CONTENTS

1. WHAT IS GPTIPS?

GPTIPS is a genetic programming (GP) tool for use with MATLAB.

GP is a biologically inspired machine learning method that evolves computer programs to perform a task. It does this by randomly generating a population of computer programs (represented by tree structures) and then mutating and crossing over the best performing trees to create a new population. This process is iterated until the population contains programs that (hopefully) solve the task well.

When the task is building an empirical mathematical model of data acquired from a process or system, the GP is often known as *symbolic regression*.

Unlike traditional regression analysis (in which the user must specify the structure of the model), GP automatically evolves both the structure and the parameters of the mathematical model.

GPTIPS provides a number of convenient functions for exploring the population of evolved models, investigating model behaviour, post-run model simplification and export to different formats, e.g. graphics file, LaTex expression, symbolic math object or standalone m file.

One of the main features of GPTIPS is that it can be configured to evolve multigene individuals. Simpler, more accurate models can be created by using this feature.

1.1.1 WHAT IS MULTIGENE SYMBOLIC REGRESSION?

A multigene [1] individual consists of one or more genes, each of which is a "traditional" GP tree. Genes are acquired incrementally by individuals in order to improve fitness (e.g. to reduce a model's sum of squared errors on a data set). The overall model is a weighted linear combination of each gene. In GPTIPS, the optimal weights for the genes are automatically obtained (using ordinary least squares to regress the genes against the output data). The resulting pseudo-linear model can capture non-linear behaviour.

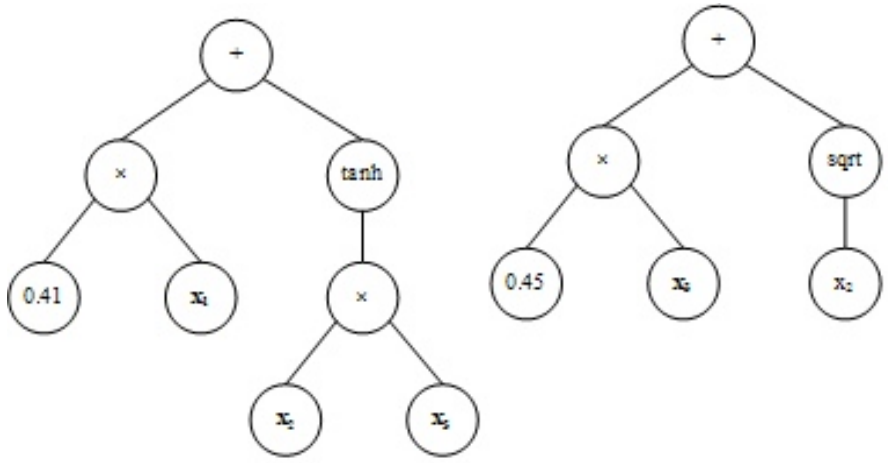

Figure 1. A pseudo-linear multigene model of output y with inputs x_1, x_2 and x_3.

The weights d_0, d_1 and d_2 are automatically obtained by least squares.

Multiple gene model:

$$y = d_0 + d_1(\ 0.41x_1 + \tanh(x_2.x_3)) + d_2(\ 0.45x_3 + \mathrm{sqrt}(x_2))$$

1.1 GPTIPS FEATURES

1.1.1 GENERAL GP FEATURES

Multiple tree (multigene) individuals.

Tournament selection & lexicographic tournament selection [5].

Standard subtree crossover operator.

Elitism.

Early run termination criterion.

Graphical population browser showing best and non-dominated individuals (fitness & complexity).

Graphical summary of fitness over GP run.

6 different mutation operators (see Appendix 1).

1.1.2 SYMBOLIC REGRESSION FEATURES

4 symbolic regression demos.

Graphical display of results of symbolic regression.

Statistical analysis of multigene symbolic regression models[†].

Multigene regression model reduction using "gene knockouts".

Rendering to PNG (portable network graphics) file of evolved symbolic regression models[‡].

Conversion of evolved multigene models to Latex format‡.

Conversion of evolved multigene models to symbolic math objects file for use outside GPTIPS ‡.

Conversion of evolved multigene models to M file for use outside GPTIPS‡.

Mathematical simplification of evolved multigene symbolic regression models‡.

Runtime validation on 'holdout' data set.

[†] Requires Statistics toolbox.
[‡] Requires Symbolic Math toolbox.

1.2 REQUIREMENTS

A relatively recent version of MATLAB (testing of GPTIPS was done using MATLAB 7.6.0). No toolboxes are required for basic functionality, but the *Symbolic Math Toolbox* is exceptionally useful for the simplification and visualisation of the evolved models when using GPTIPS for symbolic regression. The *Statistics Toolbox* is also useful for computing statistical information about multigene models.

1.3 DOCUMENTATION

There is additional information in the comments and help sections of the individual files that comprise GPTIPS. If there is sufficient interest the documentation will be improved.

1.4 INSTALLATION

Unzip the contents of the file gptips1.0.zip to a suitable directory.

1.5 REFERENCING

If you use GPTIPS for academic or commercial purposes please provide a reference. E.g.:

Searson, D. GPTIPS: Genetic Programming & Symbolic Regression for MATLAB, http://gptips.sourceforge.net, 2009.

2. SYMBOLIC REGRESSION

GPTIPS was written mainly for the purposes of multigene symbolic regression. The multigene approach (i.e. linear combination of 'smaller' low depth trees) often gives simpler models than evolving models consisting of one monolithic GP tree.

2.1 DEMOS

There are 4 symbolic regression demos included with GPTIPS. Have a look at these files to see how to configure, access and display information from a run.

gpdemo1: runs ordinary GP symbolic regression on Koza's 1992 quartic polynomial problem [2].

gpdemo2: runs multigene GP symbolic regression on a data set generated from a non-linear mathematical function comprising one output (y) and 4 inputs (x_1, …, x_4). A separate test data set is used to evaluate the evolved models. The function is $y = \exp(2x_1\sin(\pi x_4)) + \sin(x_2x_3)$ and was taken from [4]. Both testing & training data is noise free.

gpdemo3: runs multigene GP symbolic regression on a data set generated from a simulation of a non-linear pH neutralisation process. This also comprises i)one output (pH) and 4 inputs. A test data set is used to evaluate the evolved models. Both testing & training data is noise free.

gpdemo4: runs multigene GP symbolic regression on a "real life" data set comprising one output (concrete compressive strength) and 8 inputs. A holdout validation data set and a testing data set are used to evaluate the evolved models.

2.2 HOW MULTIGENE SYMBOLIC REGRESSION WORKS

In multigene symbolic regression, each prediction $\hat{y}$ of the output variable y is formed by the weighted output of each of the trees/genes in the multigene individual plus a bias term. Each tree is a function of zero or more of the N input variables $x_1, \ldots x_N$.

Mathematically, a multigene regression model can be written as:

$$\hat{y} = d_0 + d_1 \times \text{tree1} + \ldots + d_M \times \text{tree}M$$

where d_0 = bias (offset) term and $d_1, \ldots, d_M$ are the gene weights and M is the number of genes (i.e. trees) comprising the current individual. The weights (i.e. regression coefficients) are automatically determined by a least squares procedure for each multigene individual.

The number and structure of the trees is evolved automatically during a run (subject to user defined constraints) using *training data*, i.e. a set of existing input values and corresponding output values. *Testing data* (another set of input and corresponding output values from the process or system you are modelling) can be used, after the run, to evaluate the evolved models. The testing data is not used to evolve the models and serves to give an indication of how well the models generalise to new data. For further details about the way multigene individuals are handled in GPTIPS see Appendix 2.

In GPTIPS, the fitness function regressmulti_fitfun.m implements multigene symbolic regression and is used by gpdemo2, gpdemo3 and gpdemo4. Using this fitness function GPTIPS attempts to minimise the root mean squared error (RMSE) between the output y and the predicted output $\hat{y}$. In the following section it is outlined how you can use this fitness function for your own symbolic regression problems.

2.3 USING GPTIPS FOR SYMBOLIC REGRESSION

An example of a simple configuration file my_config.m for multiple gene symbolic regression is shown in Figure 2.

It is assumed that your data is located in the current directory in the file mydata.mat and comprises the training input data variable (xtrain), the training output variable (ytrain) as well as a testing data set (xtest and ytest). The data should be arranged by columns, e.g. the nth column of xtrain should contain the observations of the nth input variable.

```
function gp = my_config(gp);
% Example of a GPTIPS configuration file for multiple gene symbolic regression.

%Define population size, number of generations, fitness function name and
%optimisation type.
gp.runcontrol.pop_size = 100;
gp.runcontrol.num_gen = 100;
gp.fitness.fitfun = @regressmulti_fitfun;
gp.fitness.minimisation = true;

%Load the variables xtrain, ytrain, xtest and ytest and assign to gp structure
load mydata
gp.userdata.xtrain = xtrain;
gp.userdata.ytrain = ytrain;
gp.userdata.xtest = xtest;
gp.userdata.ytest = ytest;

%Define the number of inputs
gp.nodes.inputs.num_inp = size(gp.userdata.xtrain,2);

%Enable multiple gene mode, set tree depth and set max number of genes per
individual.
gp.genes.multigene = true;
gp.genes.max_genes = 4;
gp.treedef.max_depth = 5;

%Define function nodes
gp.nodes.functions.name = {'times','minus','plus'};
```

Figure 2. Example GPTIPS configuration file for multiple gene symbolic regression.

In this example, only a few GPTIPS settings are specified. The full list of user parameters can be found in Appendix 1. Any user parameters not explicitly set automatically use the default values. However, the user must at least specify the fitness function, the input and output data and the function nodes to be used in their configuration file.

This configuration file first sets population size = 100 and number of generations =100. The fitness function regressmulti_fitfun.m is then specified as is the fact that this is an error minimisation problem.

Next, the user data file mydata.mat is loaded and the variables within this file (here called xtrain, ytrain, xtest and ytest but they could be called anything) are assigned to the gp.userdata field. Note that the training data inputs *must* be assigned to gp.userdata.xtrain and the outputs to gp.userdata.ytrain . It is not a requirement that testing data be provided when using regressmulti_fitfun.

After this is the number of input variables is set as the number of columns in the inputs training data matrix. Next, multigene mode is enabled and the maximum number of genes per individual is set to 4.

Finally, the function nodes times, minus and plus are specified. All function nodes must be MATLAB m files (or builtin functions) on the MATLAB path. Furthermore, all function nodes must accept a fixed number of input arguments. E.g. The MATLAB rand function is not allowed to be a function node in GPTIPS because it can be called in various forms, rand(1), rand(1,1) etc.

Note that the configuration file *must* take the structure gp as an input argument *and* return it as an output argument.

This configuration can now be run using gp = rungp('my_config'). When the run is completed the data structure gp appears in the MATLAB workspace.

There are functions provided in GPTIPS to analyse the results of GP runs and some functions specifically for use with multigene symbolic regression. These will be outlined in the following sections.

2.3.1 POST-RUN ANALYSIS

At the end of any GPTIPS run enter summary(gp) to plot the best (log values) and mean fitness over the course of the run. An example of this is shown in Figure 3.

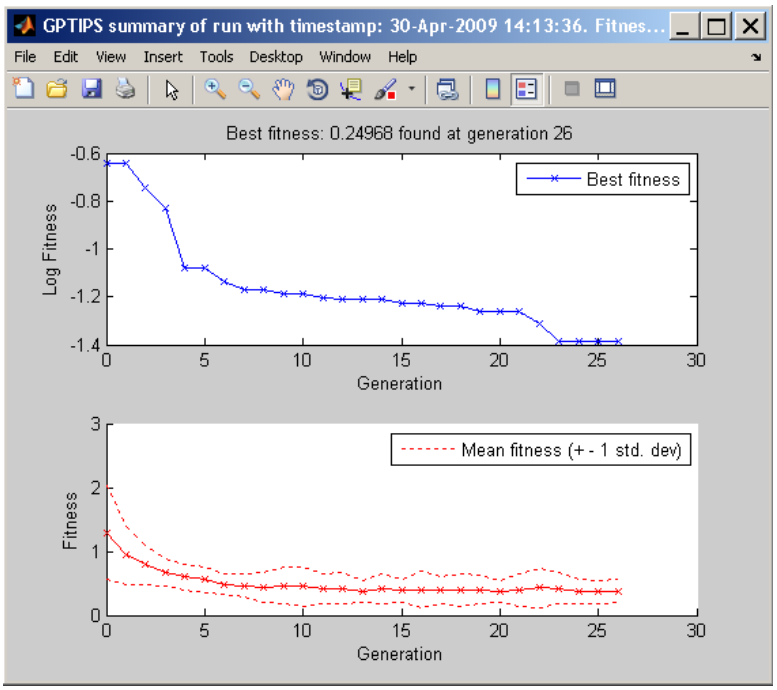

Figure 3. Result of running summary on a gp structure returned after running multiple gene symbolic regression with gpdemo3.

Furthermore, you can enter runtree(gp,'best') to examine the behaviour of the best individual from the run on your fitness function.

Alternatively, you can also enter runtree(gp, 2) to examine the behaviour of the individual with index = 2 in the current population on your fitness function (note: this is just an index and not the 2nd best individual in the population).

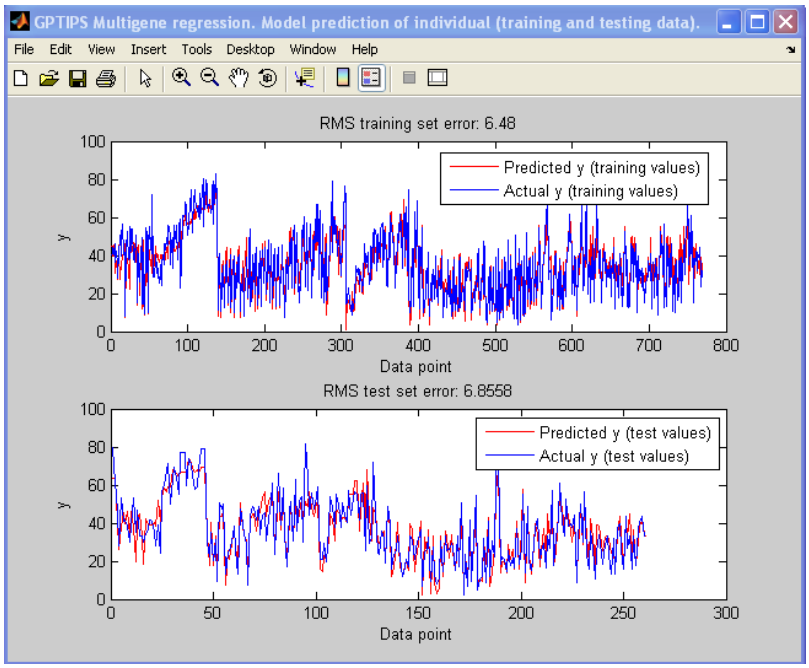

Figure 4. Result of runtree on a gp structure returned after running multiple gene symbolic regression with gpdemo4.

2.3.2 MULTIGENE SYMBOLIC REGRESSION WITH A 'HOLDOUT' VALIDATION DATA SET

As well as training and testing data sets, an additional 'holdout' validation data set can be specified to help mitigate against overfitting. GPTIPS evaluates the "best" individual at each generation against the validation data set and then tracks the results over the run. This does not affect the GP run as such but it allows the user to identify, after the run has completed, which individual performed best on a data set which was not used to build the model. This is often not the individual that performed best on the training data and it may generalise better to unseen data.

To use a holdout validation set the validation input and output values must be loaded into the fields gp.userdata.xval and gp.userdata.yval respectively. Also, the line: gp.userdata.user_fcn = @regressmulti_fitfun_validate; must be added to the run configuration file. This tells GPTIPS to run the function regressmulti_fitfun_validate (which runs the best individual of the run so far on the validation data) once each generation.

An example of a configuration file my_config2.m for multiple gene symbolic regression with training, testing and holdout validation data sets is shown in Figure 6. Again, it is assumed that the data is located in the current directory (in the file mydata2.mat) and comprises the training input data (xtrain and ytrain), the testing data set (xtest and ytest) and the holdout validation data set (xval and yval).

```
function gp = my_config2(gp);
% Example of a GPTIPS configuration file for multiple gene symbolic regression %
using training, testing and holdout validaton data sets.

%Define population size, number of generations, fitness function name and
%optimisation type.
gp.runcontrol.pop_size = 100;
gp.runcontrol.num_gen = 100;
gp.fitness.fitfun = @regressmulti_fitfun;
gp.fitness.minimisation = true;

%Load the variables xtrain, ytrain, xtest and ytest and assign to gp structure
load mydata2
gp.userdata.xtrain = xtrain;
gp.userdata.ytrain = ytrain;
gp.userdata.xtest = xtest;
gp.userdata.ytest = ytest;
gp.userdata.xval = xval;
gp.userdata.yval = yval;

%Tell GPTIPS to call validation set once per generation
gp.userdata.user_fcn = @regressmulti_fitfun_validate;

%Define the number of inputs
gp.nodes.inputs.num_inp = size(gp.userdata.xtrain,2);

%Enable multiple gene mode, set tree depth and set max number of genes per
individual.
gp.genes.multigene = true;
gp.genes.max_genes = 4;
gp.treedef.max_depth = 5;

%Define function nodes
gp.nodes.functions.name = {'times','minus','plus'};
```

Figure 6. Example GPTIPS configuration file for multiple gene symbolic regression with a holdout validation data set.

After a run using holdout validation has completed, the individual with the best performance on the validation data can be run using:

```
runtree(gp,'valbest')
```

2.4 SYMBOLIC REGRESSION: MODEL SIMPLIFICATION & ANALYSIS

2.4.1 MODEL SIMPLIFICATION

If the Symbolic Math toolbox is installed use:

```
gppretty(gp,'best')
```

to display a mathematically simplified version of the best model (single or multigene) in the population (for symbolic regression problems only) in the Matlab command window. Similarly, to simplify the multigene model with index = 5 in the population use:

```
gppretty(gp,5)
```

Furthermore, if using a holdout validation data set it is possible to simplify the best individual on the validation data set using:

```
gppretty(gp,'valbest')
```

2.4.2 EXPORTING MODELS TO LATEX AND PORTABLE NETWORK GRAPHICS (PNG) FORMAT.

The gppretty function can also return other outputs, for instance to extract a string containing the LaTeX version of the simplified genes of a multigene expression:

```
gene_latex_expr =  gppretty(gp,'best')
```

The resulting string gene_latex_expr can then be copied and pasted into a LaTeX document. For instance, after running gpdemo3 the best individual was manually rendered (using this web-based LaTeX processor and renderer: http://sciencesoft.at/latex/). (Enter help gppretty at the command line to find out more about rendering multigene models in Latex.)

Each line in the model below corresponds to a weighted gene (the first line also contains the model bias term).

$$
\begin{aligned}
y = \quad & 9.941 + 0.2858\, {x_3}^2 \\
- \quad & 0.001220\, \tanh\left(x_1 - x_3\right)\left(x_3\, x_1 + 7.663473\, x_4\right) \\
- \quad & 28.31\, x_3 \\
+ \quad & 1079.0\, \tanh\left(\tanh\left(0.02395\, x_3\right)\right)
\end{aligned}
$$

In gpdemo3, multigene models are restricted to a maximum of 4 genes with each gene restricted to depth 4 trees. This resulted in a relatively simple model that explains more than 99% of the variation in the response variable (R^2 = 0.994). By increasing tree depth and the maximum number of genes it is possible to evolve even more accurate models, *but* at the cost of increased model complexity.

Note that, in the case above, the genes are first individually mathematically simplified and *then* converted to Latex format for rendering (hence why each line is a simplified gene). But it is also possible to mathematically simplify the whole model *before* conversion to Latex. This is done with the following function call:

```
[gene_latex_expr, full_latex_expr] = gppretty(gp,'best');
```

In this case, the string full_latex_expr contains, in Latex format, the result of simplification applied to the whole model rather than the individual genes.

Additionally, the function renderLatex can be used to directly render the full simplified expression as a PNG (portable network graphics) file. This function sends the Latex version of the expression via HTTP to http://sciencesoft.at/latex/ and automatically retrieves and displays the graphics file. The PNG file is written to the current directory as gptips_model.png where it may be then copied and pasted into other applications (e.g. Word, PowerPoint , OpenOffice etc.).

This is done with the following function call:

```
renderLatex(gp,'best');
```

For instance, the following PNG file was automatically generated using this function for the previous example:

$$y = 9.941 + 0.2858\,{x_3}^2 - 0.001220\,\tanh\left(x_1 - x_3\right)\left(x_3\,x_1 + 7.663473\,x_4\right) - 28.31\,x_3 + 1079.0\,\tanh\left(\tanh\left(0.023953\,x_3\right)\right)$$

Note: If you are behind a web proxy you will need to tell MATLAB about it, otherwise this function will not work. This can be set up by going to File menu, selecting Preferences and then Web :

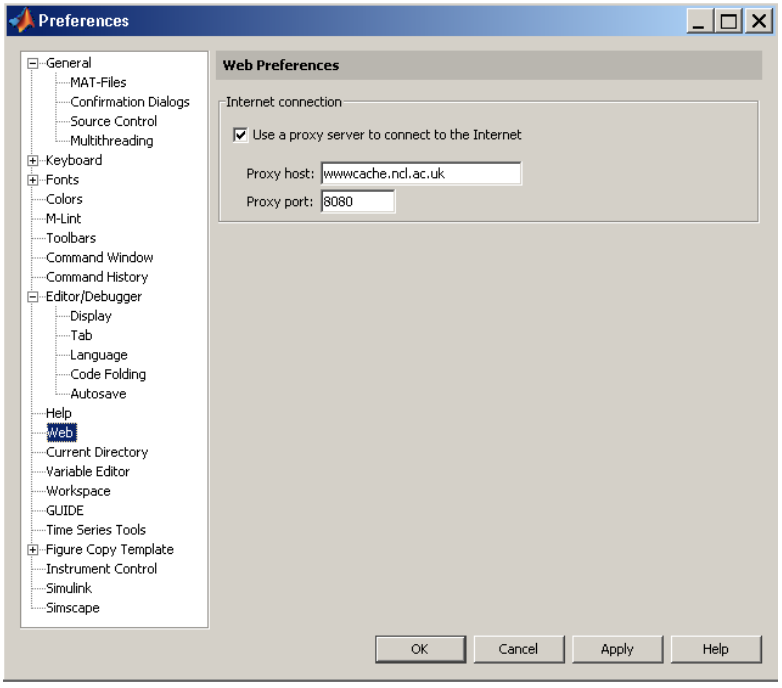

Figure 8. Example of how to configure Web proxy in MATLAB in order to enable the renderLatex function.

2.4.3 EXPORTING MODELS TO SYMBOLIC MATH OBJECTS.

The gppretty function can also output the evolved multigene models as MATLAB symbolic math objects. These can subsequently be manipulated in MATLAB like any other symbolic equation. The format for this is:

```
[gene_latex_expr, full_latex_expr, expr_sym, gene_expr_sym] = gppretty(gp,'best');
```

Where expr_sym is a symbolic math object comprising the simplification of the entire model and gene_expr_sym is a cell array of symbolic math objects where each object is a simplified gene.

These math objects are portable in the sense that they can be used to make model predictions without using any GPTIPS functions. For instance, the Symbolic Math toolbox function subs can be used as follows. First, enter new input variable values, e.g.:

```
x1 = 15;  x2 = 0.5; x3 = 16; x4 = 30;
```

Next, evaluate the overall model for these input values using the Symbolic Math toolbox subs function.

```
subs(expr_sym);
```

The model response value is then returned to the workspace.

2.4.4 EXPORTING A MULTIGENE MODEL TO A STANDALONE M FILE

The function gpmodel2mfile (which requires the Symbolic Math toolbox to be present) allows the conversion of evolved models to a portable, standalone M file that can be executed without the need for any GPTIPS functions. For example, to convert the 'best' model in the population into an M file in the current directory enter:

```
gpmodel2mfile(gp,'best','modelname');
```

This creates the file modelname.m which may be called to predict the output **y** given inputs in the matrix **x** using:

```
ypred = modelname(x);
```

For instance, gpmodel2mfile was run using the previous example:

```
gpmodel2mile(gp,'best','mymodel');
```

which yielded the following m file:

```
function ypred=mymodel(x)
% This model file was automatically generated by the GPTIPS function
gpmodel2mfile at 30-Apr-2009 12:44:00

ypred=9.941+.2858.*x(:,3).^2-.1220e-2.*tanh(x(:,1)-
x(:,3)).*(x(:,3).*x(:,1)+7.663473.*x(:,4))-28.31.*x(:,3)+1079.*tanh(tanh(.23953e-
1.*x(:,3)))
```

Figure 8. Example of a standalone MATLAB function automatically generated using the function gpmodel2mfile

Note that the function gpmodel2mfile can take any of the arguments that the function gppretty can take, e.g.

```
gpmodel2mfile(gp,2,'model');
```

Converts the 2nd individual (index not rank) in the population to the file model.m.

2.4.5 POST RUN POPULATION BROWSING WITH POPBROWSER

The popbrowser function may be used to display the population of evolved models in terms of their complexity (number of nodes) as well as their fitness.

The popbrowser is called using:

```
popbrowser(gp);
```

The popbrowser can be used to identify symbolic models that perform relatively well and are much less complex than the "best" model in the population (which is highlighted with a red circle).

The green circles comprise the pareto-optimal models in the population (i.e. models that are not strongly dominated by other models in the population in terms of fitness and model complexity). So, each green circle represents a model that is not outperformed by any other model in terms of **both** complexity and fitness.

By left clicking on a member of the population (any circle) the index of the member is revealed. The performance of this member can then be examined using the functions outlined above.

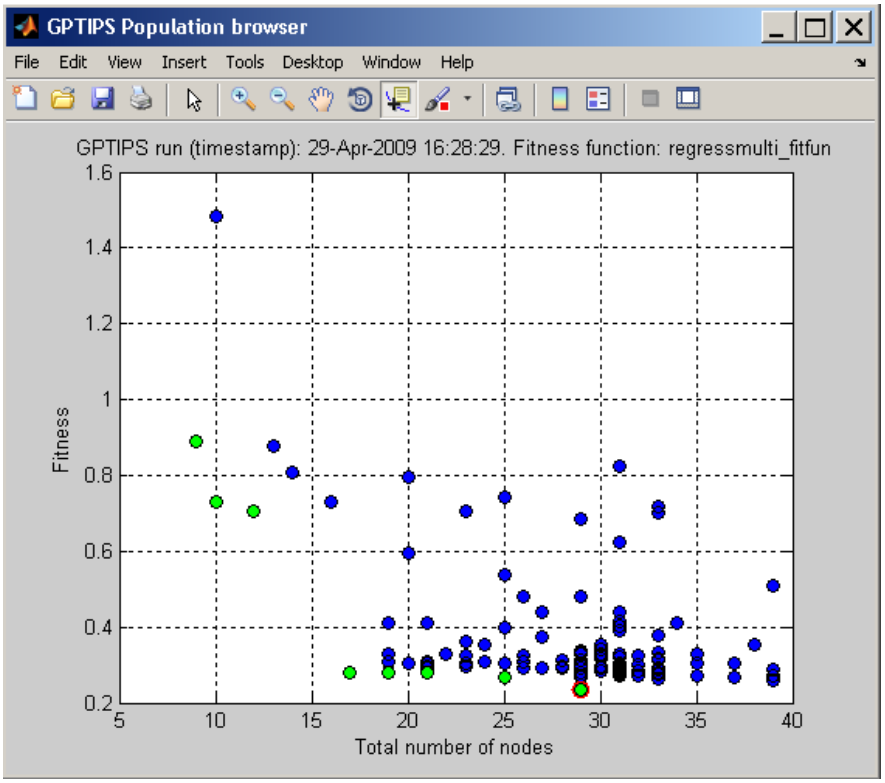

Figure 5. Example of popbrowser on a gp structure returned after running multigene symbolic regression.

3. YOUR OWN APPLICATIONS

To run GPTIPS on your own problems, e.g. non symbolic regression problems, you need two things: (1) a configuration file and (2) a file that contains your fitness function.

The configuration file is used to set GP parameters (such as what functions and inputs your problem requires) and load and set up whatever data you need for the problem. The fitness function file must actually evaluate a GP symbolic expression (or expressions in the multigene case) and asses its performance in solving whatever problem it is you have specified. The best way to see how this can be done is to look at the demos.

3.1 GUIDANCE ON WRITING YOUR OWN FITNESS FUNCTION

You can define any fitness function you like as long as the function definition is of the following form:

```
[fitness,gp] = fitnessfunction(evalstr,gp)
```

Where gp is the data structure supplied by GPTIPS to all fitness functions. In addition: fitnessfunction.m must be a valid filename on the MATLAB path and must be declared in the field gp.fitness.fitfun by your configuration file.

fitness must be a scalar numeric value produced by your fitness calculations and must be the first output argument.

evalstr must be a single row cell array where each entry/gene is a symbolic expression that MATLAB can execute using the eval command. Unless you have configured the multigene options in your configuration file, this will be a (1x1) cell array containing a single string expression. Therefore, in most cases, you will use evalstr{1} to access the string expression.

Remarks

The fitness function must contain at least one eval command to evaluate the string expression(s) contained in the evalstr cell array.

Example

The following statement evaluates the first string expression in the evalstr array and assigns the result to the variable value.

```
eval(['value = ' evalstr{1} ';']);
```

You can extract any data you like from the gp structure and use it as part of your fitness calculations. It is recommended that you use the gp.userdata field to do this.

Example

The following statements extract data from the gp structure and assign that data to 2 inputs and an output.

```
x1 = gp.userdata.xdat(:,1)

x2 = gp.userdata.xdat(:,2)

y = gp.userdata.ydat(:,1);
```

You can also write values to the gp structure. This can be useful for storing parameters from your fitness calculations for later use. Data written to the cell array in the field gp.fitness.returnvalues are stored, tracked and cached by the program automatically.

Example

If you were performing regression then the regression coefficient(s) for the current individual could be written to gp.fitness.returnvalues as follows:

`i = gp.state.current_individual;` gets the population index of the current individual.

`gp.fitness.returnvalues{i} = coeffs;` writes the regression coefficient(s) to the appropriate position in the gp.fitness.returnvalues cell array.

Note that gp.fitness.returnvalues is a cell array and so braces {} must be use to reference its entries. Note also that if you have more than one vector terminal (input) then these should all be of the same length.

3.2 COMMON PROBLEMS AND HOW TO WORK AROUND THEM

Depending on the nature of the inputs and fitness calculations, the value returned by evaluating a string expression may not be a scalar and/or it may be complex or contain Inf or NaN values. You can deal with these situations explicitly in your fitness calculations if this is causing problems.

Example

The following statements ensure that only the real parts of the result returned by the string expression are stored in the value variable

```
eval(['value = ' evalstr{1} ';']);

value = real(value);
```

Another problem that can arise is that, even when using vector inputs, some string expressions can return scalar outputs. For instance, this can occur when an expression does not contain any inputs and consists solely of operations on constants (which are scalar). Therefore, it is necessary to compensate for this if your fitness calculations require a vector. One way to get around this problem is to pre-allocate (as an appropriately sized vector) the variable in which you are storing the expression result and use vector addressing operator (:) to populate the vector with copies of the scalar returned by the expression.

Example

If x1 is an input vector then:

```
n = length(x1);

value = zeros(n,1);

eval(['value(:) = ' evalstr{1} ';']);
```

will ensure that the variable value is a vector of the correct length.

4. MISCELLANEOUS

4.1 ACKNOWLEDGEMENTS

GPTIPS is very loosely based on old GP code by Mark Willis, Ben McKay, Hugo Hiden, Mark Hinchliffe and me whilst at Newcastle University, UK.

4.2 REFERENCES

[1] Genetic programming: on the programming of computers by means of natural selection, Koza JR, MIT Press, Cambridge, MA, USA, 1992.

[2] Modelling chemical process systems using a multi-gene genetic programming algorithm, Hinchliffe MP, Willis MJ, Hiden H, Tham MT, McKay B, Barton, GW. In Genetic Programming: Proceedings of the First Annual Conference (late breaking papers). The MIT Press, USA, 1996.

[3] Co-evolution of non-linear PLS model components, Searson, DP, Willis MJ, Montague GA, Journal of Chemometrics, Wiley, 21: 592-603, 2007. DOI: 10.1002/cem.1084

[4] Comparison of adaptive methods for function estimation from samples, Cherkassky V, Gehring D, Mulier F, IEEE Transactions on Neural Networks, 7 (4): 969-984, 1996.

[5] Lexicographic parsimony pressure, Luke S, Panait L., Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), 2002.

4.3. LICENSE

but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/

APPENDIX 1: GPTIPS USER PARAMETERS

Run control parameters	Value Type	Description	Default
gp.runcontrol.pop_size	Integer > 1	Population size.	100
gp.runcontrol.num_gen	Integer > 0	Number of generations to run for (including generation zero).	100
gp.runcontrol.verbose	Integer ≥ 0	Set to *n* to display run information to screen every *n* generations	10
gp.runcontrol.quiet	Boolean	Set to True to suppress output to screen.	False
gp.runcontrol.savefreq	Integer ≥ 0	Set to *n* to save state info every n generations to file gptips_tmp.mat.	0

Fitness parameters	Value Type	Description	Default
gp.fitness.fitfun	MATLAB function handle	Function handle of the fitness function.	N/A
gp.fitness.minimisation	Boolean	True to minimise the fitness function. False to maximise it.	True
gp.fitness.terminate	Boolean	True to terminate GPTIPS run if the best fitness equals or exceeds a threshold value.	False
gp.fitness.terminate_value	Double	Termination threshold value.	0

Selection parameters	Value Type	Description	Default
gp.selection.method	String	Only tournament selection is currently implemented so this must be set to 'tour'.	'tour'
gp.selection.tournament.size	Integer > 0	The size of the tournament.	2
gp.selection.tournament.lex_pressure	Boolean	True to use plain lexicographic tournament selection [5].	True
gp.selection.elite_fraction	0 ≤ Double <1	Elitism, i.e. the fraction of population to copy directly to the next generation without modification.	0.05

Terminal nodes	Value Type	Description	Default
gp.nodes.inputs.num_inp	Integer > 0	The number of "input" nodes (not including ephemeral random constants).	N/A
gp.nodes.const.range	(1 x 2) Double vector.	The range that constant nodes are generated from with uniform probability.	[-10 10]
gp.nodes.const.p_ERC	0 ≤ Double ≤ 1	Probability that a constant node, rather than an input node, will be generated when adding a terminal node to a tree.	0.2

Function nodes	Value Type	Description	Default
gp.nodes.functions.name	(1 x *F*) cell array of function names	Cell array of the user's *F* function function nodes. Each entry must be a string containing the name of the corresponding m file.	N/A
gp.nodes.const.active	(1 x *F*) Boolean vector.	A Boolean array of length F. Determines if the function nodes in gp.nodes.functions.name are included in the current run. True = included. False = excluded.	(1 X F) array containing True entries.

Genetic operators (must sum to 1)	Value Type	Description	Default
gp.operators.mutation.p_mutate	0 ≤ Double ≤ 1	Probability of GP tree mutation.	0.1
gp.operators.crossover.p_cross	0 ≤ Double ≤ 0	Probability of GP tree crossover.	0.85
gp.operators.directrepro.p_direct	0 ≤ Double ≤ 1	Probability of GP tree direct copy.	0.05

Tree build parameters	Value Type	Description	Default
gp.treedef.build_method	1 ≤Integer ≤ 3	Tree building algorithm to use. 1 = full 2 = grow 3 = ramped 1/2 and 1/2.	3
gp.treedef.max_depth	Integer > 1	Maximum depth of trees.	6
gp.treedef.max_nodes	Integer ≥1	Maximum number of nodes per tree.	Inf
gp.treedef.max_mutate_depth	Integer > 1	Maximum depth of subtrees created by mutation.	6

Multigene parameters	Value Type	Description	Default
gp.genes.multigene	Boolean	True enables multigene individuals.	False
gp.genes.max_genes	Integer ≥ 1	Maximum number of genes per individual.	1
gp.genes.operators.p_cross_hi	0 ≤ Double ≤ 1	If multigene is enabled, this is the proportion of crossover events that are high level gene crossovers (the swapping of 1 or more complete genes between individuals).	0.2

Mutation settings	Value Type	Description	Default
gp.operators.mutation.mutate_par	(1 x 6) Double vector.	Probabilities of mutation type N occurring is the Nth entry in this vector*.	[0.9 0.05 0.05 0 0 0]
gp.operators.mutation.gaussian.std_dev	0 ≤ Double	Standard deviation of perturbation applied in mutation type 3.	0.1

*The 6 mutation types are:

1 Ordinary sub-tree mutation.

2 Switch an input terminal to another randomly selected input terminal.

3 Gaussian perturbation of a randomly selected constant.

4 Set a randomly selected constant to zero.

5 Substitute a randomly selected constant with a randomly generated constant.

6 Set a randomly selected constant to 1.

APPENDIX 2: DESCRIPTION OF MULTIGENE GP IN GPTIPS

The "standard" GP representation is based on the evaluation of a single tree expression J_k. The multigene representation considers a single GP individual to be constructed from a number of genes, each of which is a tree expression [1].

E.g. if the number of genes in any particular individual in the population is denoted by N_g^k then that individual can be written as $J_k = G_{1,k}, G_{2,k}, ..., G_{N_g^k,k}\}$ where $G_{1,k}$ represents the first gene of the kth individual in the population, $G_{2,k}$ represents the second gene of the kth individual in the population and so on.

The maximum number of genes that an individual can contain is set by the user and is denoted by N_{mg}. Since it is desirable to let each individual acquire genes incrementally, it is not required that it have the maximum allowable number of genes. Hence, for any given individual $1 \leq \mathrm{V}_g^k \leq \mathrm{V}_{mg}$.

Several modifications to the GP algorithm must be made in order to accommodate the existence of multigene individuals [1,3]. These are described in the following sections.

INITIAL GENERATION

The creation of individuals in the initial generation is straightforward. An individual containing a random number of genes between one and N_{mg} is generated using the "standard" algorithm for constructing symbolic expressions.

In the interests of maximising the population diversity at the beginning of the run, checks are made so that duplicate genes do not appear in newly created individuals, although no such restriction is imposed on individuals in subsequent generations. This is mainly due to the additional computational cost considerations in isolating duplicate and functionally similar genes.

RECOMBINATION: CROSSOVER AND MUTATION

Individuals that have been selected for recombination should be able to acquire new genes and swap one or more complete genes with other individuals.

In the multigene algorithm used here, both functions are performed with the *two-point high-level crossover* operator. Once the two parent individuals have been selected, two gene crossover points are selected within each parent. Then the genes enclosed by the crossover points are swapped between parents to form two

new offspring. To ensure conformity to gene size restrictions, any offspring with more than N_{gm} genes have genes randomly deleted.

Consider a case of two point high level crossover between an individual J_1 consisting of the genes $[G_{1,1}\ \ G_{2,1}\ \ G_{3,1}]$ and an individual J_2 consisting of the genes $[G_{1,2}\ \ G_{2,2}\ \ G_{3,2}\ \ G_{4,2}]$ where N_{gm} = 5. Two randomly selected crossover sections are shown below. The gene sections enclosed by the crossover points are denoted by $\langle \ldots \rangle$.

J_1 $\qquad$ $[G_{1,1}\ \langle \mathbf{G}_{2,1} \rangle\ G_{3,1}]$

J_2 $\qquad$ $[G_{1,2}\ \langle \mathbf{G}_{2,2}\ \mathbf{G}_{3,2}\ \mathbf{G}_{4,2} \rangle]$

The two crossover sections (highlighted in boldface) are exchanged resulting in the offspring O_1 and O_2. Note that this crossover mechanism facilitates the acquisition of new genes for both individuals as well as causing the reduction in the overall number of genes for one individual and an overall increase for the other.

O_1 $\qquad$ $[G_{1,1}\ \mathbf{G}_{2,2}\ \ \mathbf{G}_{3,2}\ \mathbf{G}_{4,2}\ G_{3,1}]$

O_2 $\qquad$ $[G_{1,2}\ \ \mathbf{G}_{2,1}]$

Provision is also made for recombinative processes at the single gene level, i.e. single genes can undergo mutation and crossover in much the same way as the standard tree expression in ordinary GP. If no provision were made for this, the GP algorithm would merely reshuffle the genes that were created in the initial generation. Hence, the mutation operator works in an almost identical manner to its standard GP counterpart. A single gene is randomly selected from the parent and the normal subtree mutation is performed. The mutated gene then replaces the original gene in the offspring.

The corresponding crossover operation is referred to as *low level crossover* [1]. A single gene is extracted at random from each parent. These genes then undergo the standard GP crossover procedure, and the two resulting genes replace the originals in the offspring.

Finally, the direct reproduction operator is unchanged from the standard GP implementation. The entire individual is simply copied to the next generation without modification.

It is a simple matter to decide what recombination operator to apply when building a new population. When crossover is selected an additional probability parameter P_{high} is used to determine whether two-point high level crossover or low level crossover is used.