University of Cambridge

Part IIB Engineering

MEng Final Report

Symbolic Algebra for Gaussian Processes: Final Report

May 31, 2017

List of Figures

1.1 *Relationships between different Gaussian linear models* 4

2.1 *Inheritance diagram for relevant classes for our SuperMat collection* 10
2.2 *Inheritance diagram for classes in our SuperMat collection* 12
2.3 *The three fundamental operations.* 16

List of Tables

3.1 *Sequence of operations to obtain recursive equations for Kalman Filter using SymGP and hand calculations.* 19
3.2 *Sequence of operations to obtain recursive equations for FITC GP approximation using SymGP and hand calculations.* 22

Chapter 1

Introduction

1.1 Motivation

Multivariate Gaussian (MVGs) distributions are ubiquitous in machine learning. Examples of models where they are used are: factor analysis (FA), principal components analysis (PCA), probabilistic PCA (pPCA), linear Gaussian state-space models (LGSSMs) (see [Murphy, 2012] for introductions to all these methods), inter-battery factor analysis (IBFA) [Tucker, 1958], canonical correlation analysis (CCA) [Murphy, 2012], slow feature analysis (SFA) [Wiskott and Sejnowski, 2002], Bayesian linear regression (BLR) [Murphy, 2012] and Gaussian processes (GPs) [Murphy, 2012], [Rasmussen and Williams, 2005]. Figure 1.1 shows how these models are related and are extended illustrating the different varieties of these models.

Manipulating these distributions requires operations such as multiplication and convolution of Gaussians, conditioning on variables, marginalising out variables, etc. These operations are usually tedious to perform by hand as they require a practitioner to apply the same rules repeatedly simply to get to the desired form of the distribution-of-interest's parameters. In addition, hand calculations are prone to error as different levels of care and detail have to be applied when dealing with models that have varying complexities of expressions. These calculations really shouldn't have to be done repeatedly as it is a massive waste of time that could be better spent on more intellectually demanding tasks that allow for a practitioner's increased productivity.

As a more concrete illustration of this problem, we provide a derivation of the Kalman Filter equations which demonstrates many of the operations that must be used when the derivation is done by hand. These operations are common to all Gaussian models and so this example highlights many problems that apply in general.

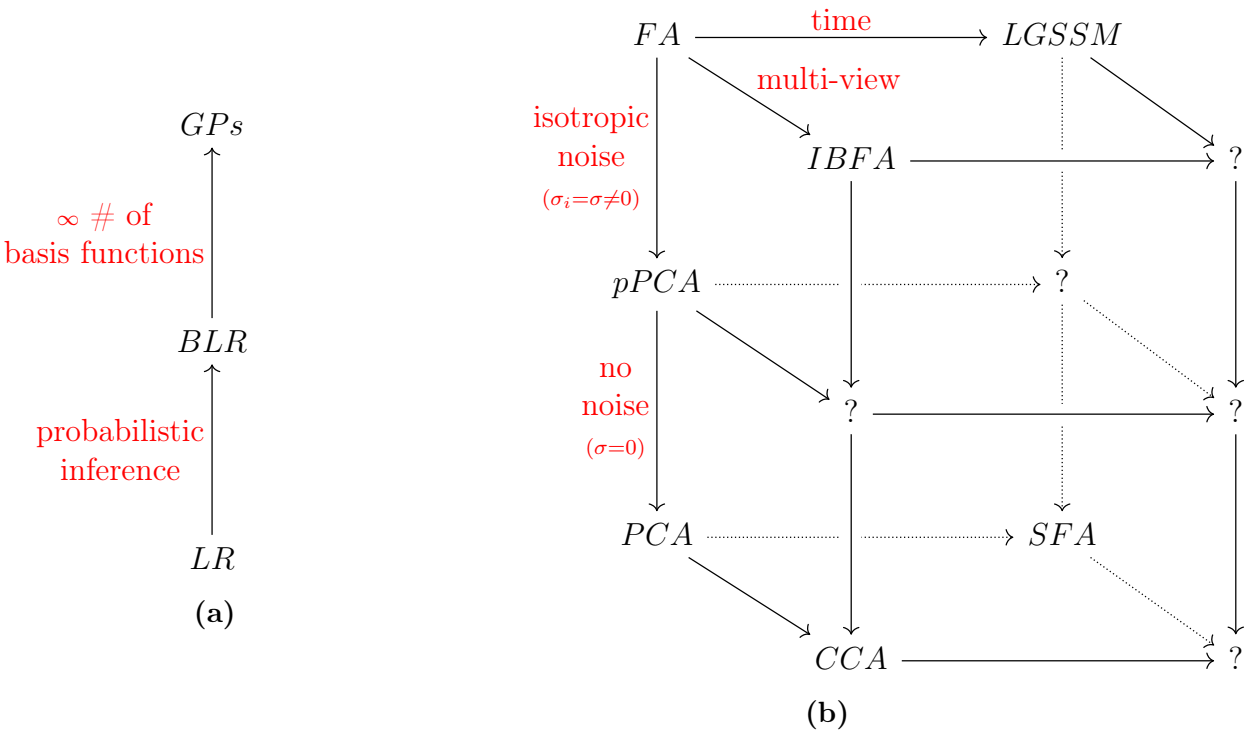


Figure 1.1: Relationships between different Gaussian linear models. *(a) LR - Linear regression, BLR - Bayesian Linear Regression, GP - Gaussian Process. (b) FA - Factor Analysis, LGSSM - Linear Gaussian State-Space Model, PCA - Principal Components Analysis, pPCA - probabilistic PCA, IBFA - Inter-Battery Factor Analysis, CCA - Canonical Correlation Analysis, SFA - Slow Feature Analysis. The question marks ("?") indicate models that have no formal name.*

1.2 Example: Derivation of Kalman Filter

Let us consider the following linear Gaussian state space model (LGSSM):

$$x_t = Ax_{t-1} + b_t + w_t$$
$$y_t = Cx_t + d_t + e_t$$
$$w_t \sim \mathcal{N}(0, Q), \qquad e_t \sim \mathcal{N}(0, R)$$

with b_t and d_t being known vectors based on a known input.

The goal of the Kalman Filter is to make a probabilistic prediction about the state given some observations i.e. $p(x_t|y_{1:t})$ for $t = 1 \dots T$ and/or to return the marginal likelihood, $p(y_{1:t})$. As the mean and covariance of $p(x_0|y_0) = p(x_0)$ are known, we can derive the moments of $p(x_t|y_{1:t})$ by recursively defining it in terms of those of $p(x_{t-1}|y_{1:t-1})$ as all distributions used are Gaussian. The sequence of operations that we take can then be broken into five steps[1]:

[1]Note that some of these operations can be combined and done in one step by hand e.g. the first and second steps can be combined by using the result that when $p(y|x) = \mathcal{N}(y; Ax + b, \Sigma_y)$ and $p(x) = \mathcal{N}(x; \mu_x, \Sigma_x)$,

- **Multiplication**. $p(x_t, x_{t-1}|y_{1:t-1}) = p(x_t|x_{t-1})p(x_{t-1}|y_{1:t-1})$ - Here $p(x_t|x_{t-1}) = \mathcal{N}(x_t; Ax_{t-1} + b_t, Q)$ and $p(x_{t-1}|y_{1:t-1}) = \mathcal{N}(x_{t-1}; \hat{x}_{t-1|t-1}, P_{t-1|t-1})$, where $\hat{x}_{t-1|t-1}$ and $P_{t-1|t-1}$ are the mean and covariance we want to update . In this operation, we have a special form of multiplication where we multiply a conditional and a marginal distribution to obtain a joint. Normally, we would need to expand the exponents of the two distributions and collect appropriate terms (e.g. completing the square) but this can be simplified by noticing the special structure relating the distributions.

  Consider the general case $p(a, b) = \mathcal{N}(\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]; \left[\begin{smallmatrix} \mu_a \\ \mu_b \end{smallmatrix}\right], \left[\begin{smallmatrix} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{smallmatrix}\right])$, where when we condition on b, we obtain the equations: $p(a|b) = \mathcal{N}(a; \underbrace{\mu_a + \Sigma_{ab}\Sigma_{bb}^{-1}(b - \mu_b)}_{\mu_{a|b}}, \underbrace{\Sigma_{aa} - \Sigma_{ab}\Sigma_{bb}^{-1}\Sigma_{ba}}_{\Sigma_{a|b}})$ and $p(b) = \mathcal{N}(b; \mu_b, \Sigma_{bb})$. Therefore, to return to $p(a, b)$, we need to solve for the unknowns $\Sigma_{ab} = \Sigma_{ba}^T, \mu_a, \Sigma_{aa}$ from the knowns $\mu_{a|b}, \Sigma_{a|b}, \mu_b, \Sigma_{bb}$ which requires manipulating these knowns so as to put them into a form that allows for matching coefficients with the unknowns e.g. for the mean we can match the coefficient of b. This gives the standard result (derivation given in section 2.3): If $\mu_{a|b} = \Lambda + \Omega b$, the unknown parameters are given as

$$\begin{aligned} \Sigma_{ab} &= \Omega\Sigma_{bb} \\ \mu_a &= \Lambda + \Omega\mu_b \\ \Sigma_{aa} &= \Sigma_{a|b} + \Omega\Sigma_{bb}\Omega^T \end{aligned}$$

  This is done for the KF equations above to get:

$$p(x_t, x_{t-1}|y_{1:t-1}) = \mathcal{N}\left(\begin{bmatrix} x_t \\ x_{t-1} \end{bmatrix}; \begin{bmatrix} A\hat{x}_{t-1|t-1} + b_t \\ \hat{x}_{t-1|t-1} \end{bmatrix}, \begin{bmatrix} AP_{t-1|t-1}A^T + Q & AP_{t-1|t-1} \\ P_{t-1|t-1}A^T & P_{t-1|t-1} \end{bmatrix}\right)$$

- **Marginalisation**. $p(x_t|y_{1:t-1}) = \mathcal{N}(x_t; A\hat{x}_{t-1|t-1}+b_t, AP_{t-1|t-1}A^T+Q) = \mathcal{N}(x_t; \hat{x}_{t|t-1}, P_{t|t-1})$ - Here we simply select the corresponding entries in the mean vector and covariance matrix of $p(x_t, x_{t-1}|y_{1:t-1})$.

- **Multiplication**. $p(y_t, x_t|y_{1:t-1}) = p(y_t|x_t)p(x_t|y_{1:t-1})$ - For $p(y_t|x_t) = \mathcal{N}(y_t; Cx_t+d_t, R)$ we get:

$$p(y_t, x_t|y_{1:t-1}) = \mathcal{N}\left(\begin{bmatrix} y_t \\ x_t \end{bmatrix}; \begin{bmatrix} C\hat{x}_{t|t-1} + d_t \\ \hat{x}_{t|t-1} \end{bmatrix}, \begin{bmatrix} CP_{t|t-1}C^T + R & CP_{t|t-1} \\ P_{t|t-1}C^T & P_{t|t-1} \end{bmatrix}\right)$$

$\int p(y|x)p(x)dx = p(y) = \mathcal{N}(y; A\mu_x + b, A\Sigma_x A^T + \Sigma_y)$. However, the detail shown here is to illustrate the underlying process that is needed which may be used in other models.

which we calculate using the same method as above.

- **Conditioning**. $p(x_t|y_{1:t}) = \text{cond}_{y_t}\{p(y_t, x_t|y_{1:t-1})\} = \mathcal{N}(x_t; \hat{x}_{t|t}, P_{t|t})$ where:

$$\tilde{y}_t = y_t - C\hat{x}_{t|t-1} - d_t \tag{1.1}$$
$$K_t = P_{t|t-1}C^T(CP_{t|t-1}C^T + R)^{-1} \tag{1.2}$$
$$\hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t\tilde{y}_t \tag{1.3}$$
$$P_{t|t} = (I - K_tC)P_{t|t-1} \tag{1.4}$$

  For this we need to use the conditioning formula again and then simplify and define terms appropriately to obtain the recursive formulae for $\hat{x}_{t|t}$ and $P_{t|t}$.

This derivation is fairly lengthy and involves many laborious tasks such as solving linear equations for unknowns, keeping track of lengthy equations and their constituent variables, repetition of methods to derive similar results and noticing areas where simplifications can be made. These calculations are prone to error when they are used in more complicated models or even cases where notation can cause confusion and thus results in plenty of man hours being wasted trying to correct them. Clearly, a better method is needed.

1.3 Automation?

A natural way to solve this problem would be to automate the operations. An intuitive method of doing this is to utilise computer algebra systems (CASs) to manipulate the algebraic expressions for the parameters of the distributions. This way the desired expressions can be obtained in a few lines of code in a fairly standard and consistent manner that does away with the errors the hand calculations may produce and drastically reduces the amount of time spent on them.

The aim of this project was then to design and implement a software library that performs these operations using a symbolic algebra package. Using our solution we can perform the Kalman Filter derivation using four lines of code (after some initial setup), which is a drastic improvement over the length of calculations needed to perform these operations (as shown in 3.1 in section 3.1):

```
p_xt_xprev = p_xt*p_xprev     # p(x_t,x_{t-1}|y_{1:t-1})
p_xt_predict = p_xt_xprev.marginalise([x_prev])     # p(x_t|y_{1:t-1})
p_yt_xt = p_yt*p_xt_predict     # p(y_t,x_t|y_{1:t-1})
p_xt_update = p_yt_xt.condition([y_t])     # p(x_t|y_{1:t})
```

1.4 Relations to Other Work

During the project,

1.5 Contributions of project

The main contribution is the creation of a fairly flexible software library that performs the MVG operations and produces easily interpretable/useful output. Two main features of the library are:

- A simple and small API to create and manipulate multivariate Gaussian distributions. There are only a handful of functions that need to be used to perform standard operations.
- Several utility functions to perform operations such as simplifications of expressions using substitutions, numerical evaluation, production of LaTeX code.

1.6 Outline of report

The report is structured as follows:

- Chapter 2 - We describe the main features of the software library and the design choices behind them.
- Chapter 3 - We give some detailed examples of applications of the library to show e.g. FITC, Kalman Filter (which is compared to the above example) and CCA.
- Chapter 4 - Extra features of the library such as simplification, LaTeX printing and numerical evaluation are described.
- Chapter 5 - We conclude the report and give directions for further work.

Chapter 2

Software Library: SymGP

In designing a software library to automate the hand calculations, we desire the following attributes:

- **Flexibility**. The library should be able to work with distributions that are functions of different numbers of variables[1] that could be intersecting sets. It should be able to conduct operations between a wide range of distributions.
- **Abstraction**. Manipulating the expressions for the MVG moments uses identities such as the conditioning formulas in section 1.2 which ends up creating long convoluted expressions that are difficult to parse and may be costly to manipulate at every step computationally. Therefore, finding a way to abstract these expressions allows us to work on shorter expressions that makes it easy to decipher and manipulate them.
- **Ease of use**. To provide a significant improvement over the hand calculations, the API of the library must be very simple to use and only require a few operations to get to a desired distribution and desirably a few operations to obtain a suitable form for its parameters.
- **Useful output**. The library should produce an output that is easily interpretable and serves the requirements of the user adequately. For example, we should be able to extract output that can be easily inserted into other media or can be used directly for numerical evaluation.

With this in mind, we started designing the library by first considering the API for an MVG object and the fundamental operations that it must implement. These operations

[1] *Variable* here means a discrete representation of a set of real-world variables that is stored in a computer e.g. the handle `v` in a Python program can represent a $n \times 1$ vector of real-world variables

boiled down to: *multiplication of Gaussians*, *conditioning on a set of variables* and *marginalising out some variables*. We want the multiplication operation to work between MVGs over possibly intersecting sets of variables and the MVG representation to represent distributions over any number of variables within computational constraints. These requirements mean that we would need a representation for the moments of a distribution that scales in a straightforward manner with the number of variables but still allows us to retain the same handle when conducting operations with other MVGs. In other words, we need a symbolic matrix/vector representation that abstracts away the full expression of the parameters of the distribution and can flexibly store these full expressions in a matrix form for multiple variables using common data structures or a as a simple single matrix expression for a single variable. Also, as we want to conduct matrix operations with these parameters, the representation should work for variables and constants.

Considering this, we would first need a software package that can manipulate matrix expressions symbolically. As it was decided beforehand to use Python as the programming language (due to its relative simplicity and extensibility), we constrained the search to symbolic algebra packages in Python. We chose SymPy [Meurer et al., 2017] because:

- It is free whilst other packages that are more flexible, popular packages such as Maple and Mathematica cost hundreds of dollars in licenses.
- It is written in Python meaning that we can easily extend it.
- It is lightweight compared to other libraries such as Sage which are very large and thus may have more dependencies than SymPy.

Using SymPy, we sought to rewrite the matrix expression/operation classes and functions such that our library can have complete control over how the matrices and vectors that are used interact and the resulting expressions simplify. We fulfilled this by creating a collection of classes that reimplement the corresponding matrix classes for matrix expressions and operations in SymPy. We collectively call these classes the SuperMat collection of classes.

However, before we delve into these classes we must understand the underlying matrix/-matrix expression representations in SymPy so as to motivate our decision to create these new classes.

2.1 Background: SymPy's MatrixExpr classes

The MatrixExpr class is the superclass for matrix expressions that represents abstract matrices. All the classes that we subclass for the SuperMat collection inherit from this class as shown in Figure 2.1.

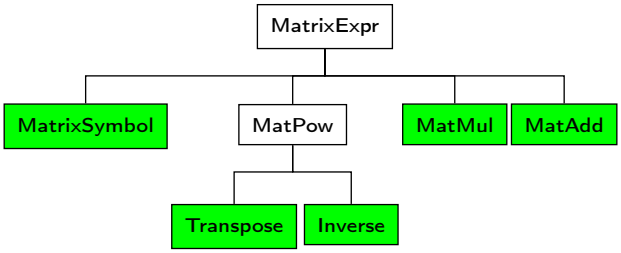


Figure 2.1: Inheritance diagram for relevant classes for our SuperMat collection. *The classes in green are the ones we subclass*

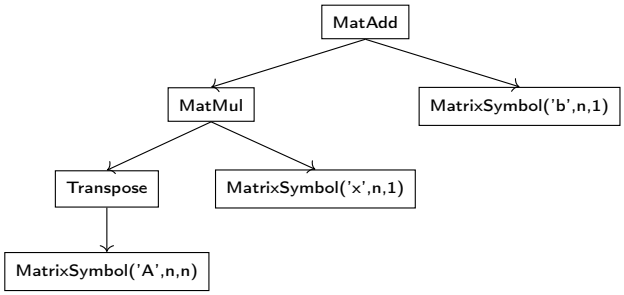
The classes that we will use are:

- MatrixSymbol. This represents a standard Matrix. We specify the name and shape when we initialise it e.g. A = MatrixSymbol('A', n, n) gives a $ne \times n$ matrix symbol called A where n = Symbol('n') (Symbol is SymPy's generic representation of a variable).

- Transpose and Inverse. These simply encapsulate the MatExprs in that they take a MatExpr as an argument. For example, we can get the inverse of a MatrixSymbol by calling Inverse(A) or A.I.

- MatAdd and MatMul. These represent matrix addition/subtraction and multiplication respectively. They are similar to Transpose and Inverse in the way they operate as an encapsulation except they can take several MatExprs as arguments. As an example, if B = MatrixSymbol('B', n, n), we can form a MatMul by writing C = A*B or C = MatMul(A, B) or C = MatMul(*[A, B]).

The usefulness of these classes for our purposes comes from the way expressions are represented as trees. Take for example the MatExpr expression A.T*x + b where A = MatrixSymbol('A',n,n), x = MatrixSymbol('x',n,1), b = MatrixSymbol('b',n,1) and n = Symbol('n'). Its expression tree is given as:



We can see that we can easily extract any sub-expression by accessing the corresponding node on this tree. This functionality allows us to easily replace sub-expressions with simpler ones thus fulfilling one of the goals of the library: abstraction. Therefore, when creating

our new classes, we would want to subclass MatrixExpr derived classes so as to leverage this property.

In the next section we delve deeper into the design and interaction of the SuperMat classes. After that we look at how these classes are used in the representation of the multivariate Gaussian distribution.

2.2 The SuperMat classes

We create our specialised representations of matrix expressions/operations by subclassing the corresponding SymPy classes and writing some new ones (the inheritance diagram is shown in Figure 2.2):

- SuperMatBase. This is an abstract class that implements operator functions (i.e. multiplication, addition, subtraction, negation, etc.) for the other classes in this list which are all subclassed from this.

- SuperMatSymbol. This is the main matrix symbol class, derived from SymPy's MatrixSymbol, which represents a generic matrix/vector. It has attributes for the detailed expressions the class abstracts away and the relations an object from this class has to other matrices/vectors. We subclass special cases that are often used in practice to simplify the API. These are: Mean, Variable, Covariance and Constant (constant matrices/vectors).

- SuperMatTranspose and SuperMatInverse. These are specialised versions of SymPy's Transpose and Inverse classes respectively. They extend them in a similar manner to SuperMatSymbol.

- SuperMatAdd and SuperMatMul. These extend SymPy's MatAdd and MatMul to make them more suited for the matrix classes above.

- KernelMatrix. This represents an evaluation of a kernel function that is used for Gaussian process applications of the library. The kernel function is represented as the Kernel class which we define.

As these classes share some properties and some are implemented for convenience (e.g. SuperMatAdd and SuperMatMul), we focus on two specific, interesting classes, SuperMatSymbol and KernelMatrix, to examine the design decisions behind all these classes.

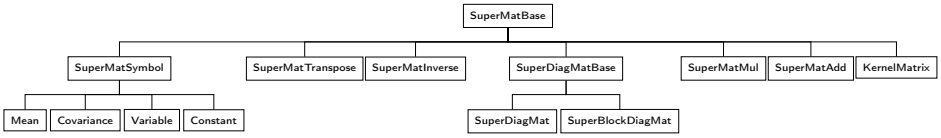


Figure 2.2: Inheritance diagram for classes in our SuperMat collection *The direct subclasses of SuperMatBase except SuperDiagMatBase and KernelMatrix inherit from their corresponding classes in SymPy i.e. SuperMatSymbol inherits from MatrixSymbol and so on. SuperDiagMatBase subclasses MatExpr directly and so does KernelMatrix with MatrixSymbol.*

2.2.1 SuperMatSymbol class

The main function of this class is to abstract away the detailed matrix expressions that it represents. The two main attributes that allow for these are:

- expanded. This stores the full matrix expression for a single variable parameter represented by this symbol. This is simply a SymPy MatrixExpr that can be formed from a combination of vanilla SymPy matrix expression classes and the relevant matrix classes we have defined. An example would be the expanded form for the mean of $p(x_t|x_{t-1})$ in the Kalman Filter example in Section 1.2, $Ax_{t-1} + b_t$, which would be given as expanded=A*x_prev + b_t where x_prev is the SuperMatSymbol for x_{t-1}. The non-expanded form would be the name of the SuperMatSymbol assigned on creation: S_{x_t|x_{t-1}}.

- blockform. This stores the block matrix/vector representation for multi-variable parameters such as a mean vector or a covariance matrix. It is stored as a Python list or list of lists where each element can be another SuperMatSymbol or a normal matrix expression (MatrixExpr). The single list is used when the symbol represents a vector such as a mean vector (e.g. [m_x, m_y]) or the vector of the variables (e.g. [x, y]. The list of lists is used for matrices such as the covariance matrix. Using a SuperMatSymbol placeholder allows us to represent large matrices/vectors using shorter expressions and also offers the possibility of inspecting the detailed expressions at different levels. This last functionality is implemented as the class methods expand_partition which expands the blockform by replacing SuperMatSymbol (or SuperMatTranspose/SuperMatInverse) elements with their blockforms if they exist and to_full_expr which replaces the elements with the expanded forms.

  An example of a blockform is the block matrix representation of the covariance of $p(x_t, x_{t-1}|y_{1:t-1})$ from the Kalman Filter example in Section 1.2. Algebraically, the

covariance expression is given as:

$$\Sigma_{x_t,x_{t-1}|y_{1:t-1}} = \begin{bmatrix} AP_{t-1|t-1}A^T + Q & AP_{t-1|t-1} \\ P_{t-1|t-1}A^T & P_{t-1|t-1} \end{bmatrix} \tag{2.1}$$

The non-expanded name for this symbol is S_{x_t,x_{t-1}|y_{1:t-1}} whereas the blockform is stored as: [[S_{x_t,x_t}, S_{x_t,x_{t-1}}], [S_{x_{t-1},x_t}, S_{x_{t-1},x_{t-1}}]]. All the elements of this list of lists are SuperMatSymbols with the given names. We can see that this way we can represent matrices of arbitrary size by representing sub-blocks with a SuperMatSymbol. We can then obtain the full covariance expression i.e. Equation 2.1 by calling the to_full_expr method of S_{x_t,x_{t-1}|y_{1:t-1}} which gives the blockform: [[Q + A*P_{t-1|t-1}*A', A*P_{t-1|t-1}], [P_{t-1|t-1}*A', P_{t-1|t-1}]]. Q, A, P_{t-1|t-1} are the names of the SuperMatSymbols that represent Q, A and $P_{t-1|t-1}$ respectively.

These two attributes are accessed and written to by the multivariate Gaussian objects' operations which are described below (Section 2.3). Their structure easily allows for operations between multivariate Gaussians objects of single and multiple variables because the expanded can be easily converted to a blockform by encapsulating it in a list and the blockform's elements can be easily accessed to give single expressions as illustrated by the blockform example above.

The class also supports automatic naming for parameters symbols such as means and covariances. The name is based on the variables for which a distribution is over and the conditioned variables if they exist such as in the blockform example above. This feature alleviates the need to specify the names of distribution parameters which at times may be impossible as distribution objects are created automatically.

The SuperMatInverse and SuperMatTranspose classes are then similar to SuperMatSymbol where the appropriate expandeds and blockforms are calculated. For the blockform we calculate the inverse block matrix by applying the formulas for the inverse of a matrix partitioned into four blocks, recursively[2]. By this we mean that if any of the four SuperMatSymbols representing the blocks has a blockform, we apply the matrix inversion lemma to them accordingly. This calculation could pose a serious computational cost if the matrix is large but as the number of variables that will be used are not expected to be numerous, this cost will not become a bottleneck.

[2]The inverse of a matrix partitioned into four blocks is given as: $\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1}+A^{-1}B(D-CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D-CA^{-1}B)^{-1} \\ -(D-CA^{-1}B)^{-1}CA^{-1} & (D-CA^{-1}B)^{-1} \end{bmatrix}$

2.2.2 Kernel & KernelMatrix class

The KernelMatrix class represents an instantiation of the kernel function defined by the Kernel class. The Kernel represents a GP covariance function/kernel. The class supports addition, subtraction and multiplication operations between Kernel objects such that we can symbolically compose complicated kernels from base kernels where we only specify a name. As an example we can multiply K = Kernel(name='K') with Q = Kernel(name='Q') as S = K*Q to give a new Kernel object with the Python handle S.

In addition, the library supports covariance functions of the form $Q(x_i, x_j) = K_1(x_i, u)$ $M_{u,u}K_2(u, x_j)$ where x_i, x_j are vector inputs to kernels K_1 and K_2 and $M_{u,u}$ is a square matrix of $\dim(u)$ where u is a vector of fixed values (e.g. pseudo-points in GP approximation models). We can initialise a multiplicative Kernel object of this form by specifying the $M_{u,u}$ matrix in the constructor. The left (K_1) and right (K_2) kernels are stored internally in a list.

This Kernel class then has a method K(xi, xj) which *evaluates* the composition of kernels by recursively evaluating the left and right kernels stored in sub_kernels of a kernel expression (additive, subtractive or multiplicative). The Kernels are evaluated to give KernelMatrix objects. These KernelMatrix objects inherit from MatrixSymbol and SuperMatBase meaning that they act like a normal MatrixSymbol but have their operator functions overridden by SuperMatBase so as to allow them to share properties with the other SuperMat classes.

2.3 MVG representation: MVG

With the SuperMat classes defined, we now have the components to create our representation of a multivariate Gaussian distribution. A natural way to represent the distribution would be to store the symbolic expressions for the distribution parameters. We store this in either moment form (e.g. mean, covariance) or natural parameter form (e.g. precision). These parameters are the main attributes that are set when new MVG objects are created.

The crux of MVG's functionality comes from the three main operations we had identified previously. They work as follows (see Figure 2.3):

- **Multiplication**. The __mul__ operator (*) function of the MVG class is overridden to implement multiplication between MVG objects. The question we then have to ask is what kind of multiplications should be supported. As we are only concerned with cases that are likely to appear as opposed to the full set of the combination of variables, we initially consider three classes: the variables of the distributions are the same ($p(a, b|.)q(a, b|.)$), they overlap ($p(a, b|.)q(b, c|.)$)) or they are distinct ($p(a|.)p(b|.)$). As the last case can be considered a special case of the second one, we can implement

both using the same piece of code. Also, as the conditional case ($p(a|b,.)p(b|.)$) can be solved using a specific as a special case we implement a separate section of code for this case. The three cases we then obtain are:

- *Same set of variables* ($p(a,b|.)q(a,b|.)$): We can calculate the new MVG parameters by simply using completion of the squares in the exponent of the two Gaussians i.e. $\Sigma_c = (\Sigma_1^{-1} + \Sigma_2^{-1})^{-1}$ and $\mu_c = \Sigma_c(\Sigma_1^{-1}\mu_1 + \Sigma_2^{-1}\mu_2)$.
- *Conditional case* ($p(a|b,.)p(b|.)$): Here we solve for the parameters of the new joint distribution $p(a,b|.)$[3]: $\mu_{a,b} = \left[\begin{smallmatrix}\mu_a \\ \mu_b\end{smallmatrix}\right]$ and $\Sigma_{a,b} = \left[\begin{smallmatrix}\Sigma_a & \Sigma_{a,b} \\ \Sigma_{a,b}^T & \Sigma_b\end{smallmatrix}\right]$ (The parameters in red are our unknowns). Here $p(a|b,.) = \mathcal{N}(a;\mu_{a|b},\Sigma_{a|b})$ and $p(b|.) = \mathcal{N}(b;\mu_b,\Sigma_b)$. We can then solve for the three unknowns by using the conditional formulas and matching terms. That is, if $\mu_{a|b} = \Lambda + \Omega b$ and given the conditioning formulae

$$\mu_{a|b} = \mu_a + \Sigma_{a,b}\Sigma_b^{-1}(b - \mu_b) \tag{2.2}$$
$$= \mu_a - \Sigma_{a,b}\Sigma_b^{-1}\mu_b + \Sigma_{a,b}\Sigma_b^{-1}b \tag{2.3}$$
$$\Sigma_{a|b} = \Sigma_a - \Sigma_{a,b}\Sigma_b^{-1}\Sigma_{a,b}{}^T \tag{2.4}$$

  we obtain

$$\Sigma_{ab} = \Omega\Sigma_{bb}$$
$$\mu_a = \Lambda + \Omega\mu_b$$
$$\Sigma_{aa} = \Sigma_{a|b} + \Omega\Sigma_{bb}\Omega^T$$

- *Variables overlap* ($p(a,b|.)p(b,c|.)$): In this case we first condition on the non-overlapping variables for each MVG i.e. $p(b|a,.)$ and $p(b|c,.)$. We then multiply these two using the first case to form $p(b|a,c,.)$ and do the same for the marginals i.e. $p(a,c|.) = p(a|.)p(c|.)$ using a special case where the variables do not overlap[4]. The final distribution is then obtained as: $p(a,b,c|.) = p(b|a,c,.)p(a,c|.)$ where this step is done using the conditional case.

- **Conditioning** - The condition function simply applies the conditioning formulae 2.2 and 2.4 to the mean and covariance expressions of the MVG. This only works for multi-variable MVGs as we need a variable to condition on. We do this by using utility functions to perform matrix operations (multiplication, addition, inversion, etc.) where

[3] We omit |. for clarity.

[4] This involves assigning the entries of the joint's mean and covariance blockforms with the corresponding entries from the constituent MVGs

we treat Python lists/list of lists as vectors and matrices, respectively. This way we can apply the formulae to any sized matrices and vectors.

- **Marginalisation** - The marginalise function only works on multi-variable MVGs like condition and it simply selects the corresponding sub-matrices/sub-vectors for the covariance and mean blockforms respectively.

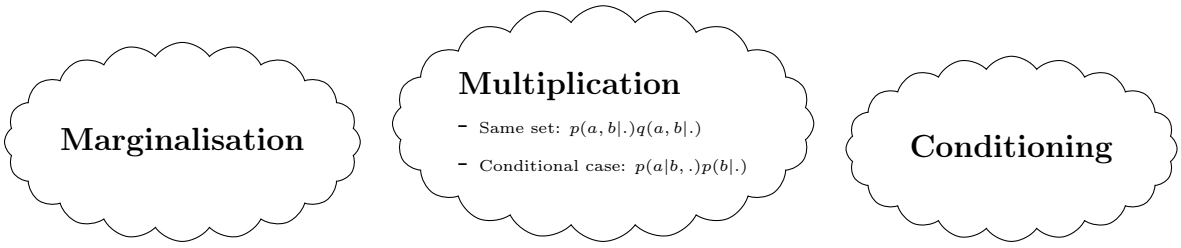


Figure 2.3: The three fundamental operations. The intersecting variables multiplication case, $p(a, b|.)p(b, c|.)$, can be performed using the other two multiplication operations.

Chapter 3

Implementations of some examples

Here we will give some examples of the application of the library and compare it to the hand-written examples.

3.1 Kalman Filter

We compare the SymGP implementation of the derivation of the Kalman Filter with the hand calculations as given in Section 1.2.

After we have imported the appropriate libraries, we set up the variables and constant parameters that we will use:

```
# Shapes: m - shape of state (x_t), n - shape of output (y_t),
# t - current time step
m, n, t = symbols('m n t')

# Variables
x_prev, x_t, y_prev, y_t, b_t, d_t, w_t, e_t =
                utils.variables('x_{t-1} x_t y_{1:t-1} y_t b_t d_t w_t e_t',
                                        [m, m, (n,t-1), n, m, n, m, n])

# Constant parameters
A, C = utils.constants('A C',[(m,m), (n,m)])
Q = Covariance(w_t, name='Q')
R = Covariance(e_t, name='R')
```

The utils.constants and utils.variables functions are utility functions that we use to create many Constants and Variables quickly.

We then create the distributions that we initially specify ($p(x_t|x_{t-1})$, $p(x_{t-1}|y_{1:t-1})$ and $p(y_t|x_t)$):

```
# p(x_(t-1)|y_(1:t-1))
f_prev = Mean(x_prev, name='m_{t-1|t-1}')
F_prev = Covariance(x_prev, name='P_{t-1|t-1}')
```

```
p_xprev = MVG([x_prev], mean=f_prev, cov=F_prev, cond_vars=[y_prev])

# p(x_t|x_(t-1))
p_xt = MVG([x_t], mean=A*x_prev + b_t, cov=Q, cond_vars=[x_prev])

# p(y_t|x_t)
p_yt = MVG([y_t], mean=C*x_t + d_t, cov=R, cond_vars=[x_t])
```

When creating an MVG, we usually specify the expanded/blockform expressions as the mean and cov arguments. For p_xprev, as we want to use a specific name for the mean and covariance, we create the Mean and Covariance parameters explicitly. This way, when we use LaTeX printing (using our utility function utils.matLatex) we would print the names we defined as opposed to automatically generated ones. However, most MVGs such as p_xt and p_yt aren't created this way and can be defined in one line giving a very short constructor. With these distributions defined, we can now perform the derivation in four lines of code (Table 3.1) to then obtain $p(x_t|y_{1:t})$ as p_xt_update.

After extracting the expressions for the mean and covariance for p_xt_update and converting it to LaTeX (using our utility function utils.matLatex, see Section 4.2 for more details), we get

$$p\left(\mathbf{x_t}|\mathbf{y_{1:t-1}}, \mathbf{y_t}\right) = \mathcal{N}\left(\mathbf{x_t}; \mathbf{m}_{\mathbf{x_t}|\mathbf{y_{1:t-1}},\mathbf{y_t}}, \mathbf{\Sigma}_{\mathbf{x_t}|\mathbf{y_{1:t-1}},\mathbf{y_t}}\right) \tag{3.1}$$

$$\begin{aligned}\mathbf{m}_{\mathbf{x_t}|\mathbf{y_{1:t-1}},\mathbf{y_t}} = \mathbf{A}\mathbf{m}_{\mathbf{t-1}|\mathbf{t-1}} + \left(\mathbf{Q} + \mathbf{A}\mathbf{P}_{\mathbf{t-1}|\mathbf{t-1}}\mathbf{A}^T\right)\mathbf{C}^T\left(\mathbf{R} + \mathbf{C}\left(\mathbf{Q} + \mathbf{A}\mathbf{P}_{\mathbf{t-1}|\mathbf{t-1}}\mathbf{A}^T\right)\mathbf{C}^T\right)^{-1} \\ \left(\mathbf{y_t} - \left(\mathbf{C}\left(\mathbf{A}\mathbf{m}_{\mathbf{t-1}|\mathbf{t-1}} + \mathbf{b_t}\right) + \mathbf{d_t}\right)\right) + \mathbf{b_t}\end{aligned} \tag{3.2}$$

$$\begin{aligned}\mathbf{\Sigma}_{\mathbf{x_t}|\mathbf{y_{1:t-1}},\mathbf{y_t}} = \mathbf{Q} + \mathbf{A}\mathbf{P}_{\mathbf{t-1}|\mathbf{t-1}}\mathbf{A}^T - \\ \left(\mathbf{Q} + \mathbf{A}\mathbf{P}_{\mathbf{t-1}|\mathbf{t-1}}\mathbf{A}^T\right)\mathbf{C}^T\left(\mathbf{R} + \mathbf{C}\left(\mathbf{Q} + \mathbf{A}\mathbf{P}_{\mathbf{t-1}|\mathbf{t-1}}\mathbf{A}^T\right)\mathbf{C}^T\right)^{-1} \\ \mathbf{C}\left(\mathbf{Q} + \mathbf{A}\mathbf{P}_{\mathbf{t-1}|\mathbf{t-1}}\mathbf{A}^T\right)\end{aligned} \tag{3.3}$$

which is similar to those from the hand calculations except a small substitution for the predicted mean ($\hat{x}_{t|t-1}$) has been made in the hand calculations in Table 3.1.

For the hand calculations, we can then simplify these expressions to get those in 1.3 and 1.4 by making defining new variables (K_t and $\tilde{y}_t$) and making appropriate substitutions. We accomplish this similarly in SymGP using another utility function utils.simplify which produces the best suggestions for simplifications of expressions using a simple heuristic (see 4.3 for details).

From Table 3.1 we can see that the number of operations required to get to the final

equations requires only four lines of code but many more lines of hand-written calculations illustrating the improvement the library provides.

SymGP	*Hand calculations*
1. Multiplication	
p_xt_xprev = p_xt*p_xprev	- $p(x_t, x_{t-1}\|y_{1:t-1}) =$ $\mathcal{N}(x_t; Ax_{t-1}+b_t, Q)\mathcal{N}(x_{t-1}; \hat{x}_{t-1\|t-1}, P_{t-1\|t-1})$ - $p(x_t, x_{t-1}\|y_{1:t-1}) =$ $\mathcal{N}\left(\begin{bmatrix} x_t \\ x_{t-1}\end{bmatrix}; \begin{bmatrix} \mu_{x_t} \\ \hat{x}_{t-1\|t-1}\end{bmatrix}, \begin{bmatrix} \Sigma_{x_t,x_t} & \Sigma_{x_t,x_{t-1}} \\ \Sigma_{x_t,x_{t-1}}{}^T & P_{t-1\|t-1}\end{bmatrix}\right)$ (Unknowns in red) - $\mu_{x_t\|x_{t-1},y_{1:t-1}} = \mu_{x_t} + \Sigma_{x_t,x_{t-1}} P^{-1}_{t-1\|t-1}\left(x_{t-1} - \hat{x}_{t-1\|t-1}\right)$ $= \mu_{x_t} - \Sigma_{x_t,x_{t-1}} P^{-1}_{t-1\|t-1}\hat{x}_{t-1\|t-1} + \Sigma_{x_t,x_{t-1}} P^{-1}_{t-1\|t-1} x_{t-1} =$ $\Lambda + \Omega x_{t-1}$ - In this case $\mu_{x_t\|x_{t-1},y_{1:t-1}} = Ax_{t-1} + b_t$ and so $\Lambda = b_t$ and $\Omega = A$. - $\therefore \Sigma_{x_t,x_{t-1}} = AP_{t-1\|t-1} \implies \mu_{x_t} =$ $AP_{t-1\|t-1}P^{-1}_{t-1\|t-1}\hat{x}_{t-1\|t-1} + b_t = A\hat{x}_{t-1\|t-1} + b_t$ - Also, $\Sigma_{x_t\|x_{t-1},y_{1:t-1}} = \Sigma_{x_t,x_t} - \Sigma_{x_t,x_{t-1}} P_{t-1\|t-1}\Sigma_{x_t,x_{t-1}}{}^T =$ $Q \implies \Sigma_{x_t,x_t} = AP_{t-1\|t-1}A^T + Q$ - $p(x_t, x_{t-1}\|y_{1:t-1}) =$ $\mathcal{N}\left(\begin{bmatrix} x_t \\ x_{t-1}\end{bmatrix}; \begin{bmatrix} A\hat{x}_{t-1\|t-1}+b_t \\ \hat{x}_{t-1\|t-1}\end{bmatrix}, \begin{bmatrix} AP_{t-1\|t-1}A^T+Q & AP_{t-1\|t-1} \\ P_{t-1\|t-1}A^T & P_{t-1\|t-1}\end{bmatrix}\right)$
2. Marginalisation	
p_xt_predict = p_xt_xprev.marginalise([x_prev])	- We select the corresponding entries of $p(x_t, x_{t-1}\|y_{1:t-1})$: $p(x_t\|y_{1:t-1}) = \mathcal{N}(x_t; A\hat{x}_{t-1\|t-1} + b_t, AP_{t-1\|t-1}A^T + Q) =$ $\mathcal{N}(x_t; \hat{x}_{t\|t-1}, P_{t\|t-1})$
3. Multiplication	
p_yt_xt = p_yt*p_xt_predict	- We use the same method as in 1. to give: $p(y_t, x_t\|y_{1:t-1}) =$ $\mathcal{N}\left(\begin{bmatrix} y_t \\ x_t\end{bmatrix}; \begin{bmatrix} C\hat{x}_{t\|t-1}+d_t \\ \hat{x}_{t\|t-1}\end{bmatrix}, \begin{bmatrix} CP_{t\|t-1}C^T+R & CP_{t\|t-1} \\ P_{t\|t-1}C^T & P_{t\|t-1}\end{bmatrix}\right)$.
4. Conditioning	
p_xt_update = p_yt_xt.condition([y_t])	We apply the conditioning formulae to get $p(x_t\|y_{1:t}) =$ $\mathcal{N}(x_t; \hat{x}_{t\|t}, P_{t\|t})$ where: $\hat{x}_{t\|t} = \hat{x}_{t\|t-1} + P_{t\|t-1}C^T(CP_{t\|t-1}C^T + R)(y_t - C\hat{x}_{t\|t-1} - d_t)$ $P_{t\|t} = P_{t\|t-1} - P_{t\|t-1}C^T(CP_{t\|t-1}C^T + R)^{-1}CP_{t\|t-1}$

Table 3.1: *Sequence of operations to obtain recursive equations for Kalman Filter using SymGP and hand calculations.*

One can say that for the third step, we do not have to repeat the hand calculation as we can derive the results by matching with corresponding variables/coefficients from step one. However, in general this may not be possible and so the steps may have to be repeated for different types of multiplications between distributions.

3.2 Gaussian Processes: FITC

To demonstrate how the library is used in Gaussian process applications, we use SymGP to derive the parameters of the predictive distribution of a sparse approximate Gaussian process model for regression, namely the FITC (Fully Independent Training Conditional) approximation. This example follows that given in [Quiñonero Candela and Rasmussen, 2005].

In standard GP regression, given a training dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i), i = 1, ..., n\}$ of n pairs of inputs $\boldsymbol{x}_i$ and noisy outputs y_i, we seek to determine the function f such that

$$y_i = f(\boldsymbol{x}_i) + \epsilon_i, \qquad \text{where} \quad \epsilon_i \sim \mathcal{N}(0, \sigma^2_{noise})$$

where σ^2_{noise} is the variance of the noise and we model f with a Gaussian process of the form

$$f \sim \mathcal{GP}(0, k)$$

where k is the covariance function of the GP.

To perform inference, we first put a joint GP prior on our training and test latent function values, $\boldsymbol{f}$ and $\boldsymbol{f}_*$:

$$p(\boldsymbol{f}, \boldsymbol{f}_*) = \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \boldsymbol{K}_{\boldsymbol{f},\boldsymbol{f}} & \boldsymbol{K}_{*,\boldsymbol{f}} \\ \boldsymbol{K}_{\boldsymbol{f},*} & \boldsymbol{K}_{*,*} \end{bmatrix}\right)$$

where K is subscript by the variables between which the covariance is computed. We then combine this with the likelihood $p(\boldsymbol{y}|\boldsymbol{f}) = \mathcal{N}(\boldsymbol{y}; \boldsymbol{f}, \sigma^2_{noise} I)$ and Bayes' rule to obtain the joint posterior

$$p(\boldsymbol{f}, \boldsymbol{f}_*|\boldsymbol{y}) = \frac{p(\boldsymbol{y}|\boldsymbol{f})p(\boldsymbol{f}, \boldsymbol{f}_*)}{p(\boldsymbol{y})}$$

We can then use this to get the predictive distribution as:

$$p(\boldsymbol{f}_*|\boldsymbol{y}) = \mathcal{N}(\boldsymbol{f}_*; K_{*,\boldsymbol{f}}(K_{\boldsymbol{f},\boldsymbol{f}} + \sigma^2_{noise} I)^{-1}\boldsymbol{y}, K_{*,*} - K_{*,\boldsymbol{f}}(K_{\boldsymbol{f},\boldsymbol{f}} + \sigma^2_{noise} I)^{-1} K_{\boldsymbol{f},*})$$

The problem with this expression is that calculating the mean and covariance requires inversion of an $n \times n$ matrix which has a computational complexity of $O(n^3)$. This is not practical for large n and so we need a method of approximating this distribution with fewer points.

One way of doing this is by introducing a set of m latent variables $\boldsymbol{u} = [u_1, \ldots, u_m]^T$ that are known as *inducing variables*. These are values of the GP corresponding to a set of

input locations $X_{\boldsymbol{u}}$ (known as the *inducing inputs*). As the GP is *consistent*[1], we can obtain $p(\boldsymbol{f}_*, \boldsymbol{f})$ by marginalising out $\boldsymbol{u}$ from the joint GP prior $p(\boldsymbol{f}_*, \boldsymbol{f}, \boldsymbol{u})$

$$p(\boldsymbol{f}_*, \boldsymbol{f}) = \int p(\boldsymbol{f}_*, \boldsymbol{f}, \boldsymbol{u}) d\boldsymbol{u} = \int p(\boldsymbol{f}_*, \boldsymbol{f}|\boldsymbol{u}) d\boldsymbol{u}$$

where $p(\boldsymbol{u}) = \mathcal{N}(\boldsymbol{u}; \boldsymbol{0}, K_{\boldsymbol{u},\boldsymbol{u}})$.

We make the approximation by assuming conditional independence of $\boldsymbol{f}$ and $\boldsymbol{f}_*$ given $\boldsymbol{u}$ which enforces the fact that $\boldsymbol{f}$ and $\boldsymbol{f}_*$ can only communicate through $\boldsymbol{u}$ (this then means that we only need to work with a smaller set of variables $\boldsymbol{u}$ to obtain the predictive distribution) such that

$$p(\boldsymbol{f}_*, \boldsymbol{f}) \approx q(\boldsymbol{f}_*, \boldsymbol{f}) = \int q(\boldsymbol{f}|\boldsymbol{u}) q(\boldsymbol{f}_*|\boldsymbol{u}) p(\boldsymbol{u}) d\boldsymbol{u}$$

The true conditionals are given as:

$$\begin{aligned} p(\boldsymbol{f}|\boldsymbol{u}) &= \mathcal{N}(\boldsymbol{f}; K_{\boldsymbol{f},\boldsymbol{u}} K_{\boldsymbol{u},\boldsymbol{u}}^{-1} \boldsymbol{u}, K_{\boldsymbol{f},\boldsymbol{f}} - Q_{\boldsymbol{f},\boldsymbol{f}}), \\ p(\boldsymbol{f}_*|\boldsymbol{u}) &= \mathcal{N}(\boldsymbol{f}_*; K_{*,\boldsymbol{u}} K_{\boldsymbol{u},\boldsymbol{u}}^{-1} \boldsymbol{u}, K_{*,*} - Q_{*,*}) \end{aligned}$$

where $Q_{\boldsymbol{a},\boldsymbol{b}} \triangleq K_{\boldsymbol{a},\boldsymbol{u}} K_{\boldsymbol{u},\boldsymbol{u}}^{-1} K_{\boldsymbol{u},\boldsymbol{b}}$. In the FITC model, the conditionals $q(\boldsymbol{f}|\boldsymbol{u})$ and $q(\boldsymbol{f}_*|\boldsymbol{u})$ (which are not the same as the exact conditionals) are given as

$$\begin{aligned} q(\boldsymbol{f}|\boldsymbol{u}) &= \mathcal{N}(\boldsymbol{f}; K_{\boldsymbol{f},\boldsymbol{u}} K_{\boldsymbol{u},\boldsymbol{u}}^{-1} \boldsymbol{u}, \mathrm{diag}[K_{\boldsymbol{f},\boldsymbol{f}} - Q_{\boldsymbol{f},\boldsymbol{f}}]) \\ q(\boldsymbol{f}_*|\boldsymbol{u}) &= p(\boldsymbol{f}_*|\boldsymbol{u}) \end{aligned}$$

With $p(\boldsymbol{u})$ and $p(\boldsymbol{y}|\boldsymbol{f})$ as given above, we can then use Bayes' rule as in the normal regression case to obtain the predictive distribution

$$q_{FITC}(\boldsymbol{f}_*|\boldsymbol{y}) = \mathcal{N}(\boldsymbol{f}_*; Q_{*,\boldsymbol{f}}(Q_{\boldsymbol{f},\boldsymbol{f}} + \Lambda)^{-1}\boldsymbol{y}, K_{*,*} - Q_{*,\boldsymbol{f}}(Q_{\boldsymbol{f},\boldsymbol{f}} + \Lambda)^{-1} Q_{\boldsymbol{f},*}) \tag{3.4}$$

where $\Lambda = \mathrm{diag}[K_{\boldsymbol{f},\boldsymbol{f}} - Q_{\boldsymbol{f},\boldsymbol{f}} + \sigma_{noise}^2 I]$. Now we perform this derivation in SymGP.

As in the Kalman Filter example, we first define our shapes, variables and a Kernel object that has the default name of K:

```
m, n, l = symbols('m n l')   # Shapes
s_y = symbols('\u03c3_y')   # The standard deviation of noise
K = Kernel()

# Inducing inputs
u = Variable('u',m,1)
```

[1]By *consistent*, we mean that the random variables obey the usual rules of marginalisation, etc.

```
# Our variables
f, fs, y = utils.variables("f f_{*} y",[n, l, n])
```

and we set up the distributions similarly:

```
# p(u) = N(u; 0, K_{u,u})
p_u = MVG([u],mean=ZeroMatrix(m,1),cov=K(u,u))

# Training conditional
q_fgu = MVG([f], mean=K(f,u)*K(u,u).I*u,
                 cov=SuperDiagMat(K(f,f)-K(f,u)*K(u,u).I*K(u,f)),
                 cond_vars=[u],
                 prefix='q_{FITC}')

# Test conditional
q_fsgu = MVG([fs], mean=K(fs,u)*K(u,u).I*u,
                   cov=K(fs,fs)-K(fs,u)*K(u,u).I*K(u,fs),
                   cond_vars=[u],
                   prefix='q_{FITC}')

# Likelihood
p_ygf = MVG([y],mean=f,cov=s_y**2*Identity(n),cond_vars=[f])
```

Note that we also specify the name of the distribution (through the argument prefix) for when we want to pretty print the MVG object.

SymGP	*Hand calculations*
1. Multiplication: $q(\boldsymbol{f}, \boldsymbol{f}_*\|\boldsymbol{u}) = q(\boldsymbol{f}\|\boldsymbol{u})q(\boldsymbol{f}_*\|\boldsymbol{u})$	
`q_f_fs_g_u = q_fgu*q_fsgu`	- As $\boldsymbol{f}$ and $\boldsymbol{f}_*$ are conditionally independent, we simply create the joint by input the correct entries from the conditonals
2. Multiplication: $q(\boldsymbol{f}, \boldsymbol{f}_*, \boldsymbol{u}) = q(\boldsymbol{f}, \boldsymbol{f}_*\|\boldsymbol{u})p(\boldsymbol{u})$	
`q_f_fs_u = q_f_fs_g_u*p_u`	- Solve for the joint parameters using the same method from the Kalman Filter example. The larger matrix means that we there will be more parameters to shuffle around thus increasing the probability of an error being made.
3. Marginalisation: $q(\boldsymbol{f}, \boldsymbol{f}_*) = \text{marg}_{\boldsymbol{u}}\{q(\boldsymbol{f}, \boldsymbol{f}_*, \boldsymbol{u})\}$	
`q_f_fs = q_f_fs_u.marginalise([u])`	- We select the appropriate entries of parameters of $q(\boldsymbol{f}, \boldsymbol{f}_*, \boldsymbol{u})$.
4. Multiplication: $q(\boldsymbol{f}, \boldsymbol{f}_*, \boldsymbol{y}) = p(\boldsymbol{y}\|\boldsymbol{f})q(\boldsymbol{f}, \boldsymbol{f}_*)$	
`q_f_fs_y = p_ygf*q_f_fs`	- We solve using the same method as in 1. but the fact that the prior is a block matrix means that we have to invert a square matrix which complicates the math slightly and makes it less clean.
5. Marginalisation: $q(\boldsymbol{f}_*, \boldsymbol{y}) = \text{marg}_{\boldsymbol{f}}\{q(\boldsymbol{f}, \boldsymbol{f}_*, \boldsymbol{y})\}$	
`q_fs_y = q_f_fs_y.marginalise([f])`	- Select appropriate entries of parameters of $q(\boldsymbol{f}, \boldsymbol{f}_*, \boldsymbol{y})$.
6. Conditioning: $q(\boldsymbol{f}_*\|\boldsymbol{y}) = \text{cond}_{\boldsymbol{f}}\{q(\boldsymbol{f}_*, \boldsymbol{y})\}$	
`q_fs_g_y = q_fs_y.condition([y])`	- Apply conditioning formulae to parameters of $q(\boldsymbol{f}_*, \boldsymbol{y})$.

Table 3.2: *Sequence of operations to obtain recursive equations for FITC GP approximation using SymGP and hand calculations.*

We can then reach the predictive distribution in six lines of code, as shown in Table 3.2, which is much less work than what would need to be done by hand where we would need

to perform two lengthy $p(a|b) * p(b)$ multiplication operations. In this case, we can arrive at the predictive distribution by simply extracting the terms involving $\boldsymbol{f}_*$ from the exponent of $q(\boldsymbol{f}, \boldsymbol{f}_*, \boldsymbol{y})$ in step four and completing the square to obtain the parameters of $p(\boldsymbol{f}_*|\boldsymbol{y})$. However, this may be just as tedious as completing the square requires us to find the mean and covariance that allow the square to be formed by inspection.

The final equations for the parameters of $q(\boldsymbol{f}_*|\boldsymbol{y})$ (using utils.matLatex) are then:

$$q_{FITC}\left(\mathbf{f}_*|\mathbf{y}\right) = \mathcal{N}\left(\mathbf{f}_*; \mathbf{m}_{\mathbf{f}_*|\mathbf{y}}, \mathbf{\Sigma}_{\mathbf{f}_*|\mathbf{y}}\right)$$

$$\mathbf{m}_{\mathbf{f}_*|\mathbf{y}} = \mathbf{K}_{\mathbf{f}_*,\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{K}_{\mathbf{u},\mathbf{f}}\left(\sigma_y^2\mathbf{I} + \text{diag}[\mathbf{K}_{\mathbf{f},\mathbf{f}} - \mathbf{K}_{\mathbf{f},\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{K}_{\mathbf{u},\mathbf{f}}] + \mathbf{K}_{\mathbf{f},\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{K}_{\mathbf{u},\mathbf{f}}\right)^{-1}\mathbf{y}$$

$$\begin{aligned}\mathbf{\Sigma}_{\mathbf{f}_*|\mathbf{y}} = \mathbf{K}_{\mathbf{f}_*,\mathbf{f}_*} - \mathbf{K}_{\mathbf{f}_*,\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{K}_{\mathbf{u},\mathbf{f}}\left(\sigma_y^2\mathbf{I} + \text{diag}[\mathbf{K}_{\mathbf{f},\mathbf{f}} - \mathbf{K}_{\mathbf{f},\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{K}_{\mathbf{u},\mathbf{f}}] + \mathbf{K}_{\mathbf{f},\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{K}_{\mathbf{u},\mathbf{f}}\right)^{-1} \\ \mathbf{K}_{\mathbf{f},\mathbf{u}}\mathbf{K}_{\mathbf{u},\mathbf{u}}^{-1}\mathbf{K}_{\mathbf{u},\mathbf{f}_*}\end{aligned}$$

We can then apply our utils.simplify function to obtain LaTeX code that can be compiled to give a form of the distribution that matches the one given in 3.4:

$$\begin{aligned}q_{FITC}\left(\mathbf{f}_*|\mathbf{y}\right) &= \mathcal{N}\left(\mathbf{f}_*; \mathbf{m}_{\mathbf{f}_*|\mathbf{y}}, \mathbf{\Sigma}_{\mathbf{f}_*|\mathbf{y}}\right) \\ \mathbf{m}_{\mathbf{f}_*|\mathbf{y}} &= \mathbf{Q}_{\mathbf{f}_*,\mathbf{f}}\left(\mathbf{Q}_{\mathbf{f},\mathbf{f}} + \sigma_y^2\mathbf{I} + \text{diag}[\mathbf{K}_{\mathbf{f},\mathbf{f}} - \mathbf{Q}_{\mathbf{f},\mathbf{f}}]\right)^{-1}\mathbf{y} \\ \mathbf{\Sigma}_{\mathbf{f}_*|\mathbf{y}} &= \mathbf{K}_{\mathbf{f}_*,\mathbf{f}_*} - \mathbf{Q}_{\mathbf{f}_*,\mathbf{f}}\left(\mathbf{Q}_{\mathbf{f},\mathbf{f}} + \sigma_y^2\mathbf{I} + \text{diag}[\mathbf{K}_{\mathbf{f},\mathbf{f}} - \mathbf{Q}_{\mathbf{f},\mathbf{f}}]\right)^{-1}\mathbf{Q}_{\mathbf{f},\mathbf{f}_*}\end{aligned}$$

Chapter 4

Extra Features

In this chapter, we describe some extra features of the library that allow the its output to be

4.1 Code Generation/Numerical Evaluation

We implement numerical evaluation of expressions using the utils.evaluate_expr function which takes in the expression and a dictionary matching the *name* (and not the Python handle) of the symbols in the expression to a numerical matrix. This matrix can be a SymPy Matrix () or a NumPy ndarray although we convert all numerical matrices to numpy.ndarrays so as to leverage NumPy's linear algebra tools.

For example, for the Kalman Filter example given in section 3.1, we can evalute the parameters of $p(x_t|y_{1:t})$ (equations 3.2 and 3.3) as such:

```
d = {'A': Matrix([[1, 0, 1, 0],[0, 1, 0, 1],[0, 0, 1, 0],[0, 0, 0, 1]]),
       'C': Matrix([[1, 0, 0, 0],[0, 1, 0, 0]]),
       'Q': 0.001*Matrix(Identity(4)),
       'R': Matrix(Identity(2)),
       'm_{t-1|t-1}': Matrix([[8],[10],[1],[0]]),
       'P_{t-1|t-1}': Matrix(Identity(4)),
       'b_t': Matrix([[0],[0],[0],[0]]),
       'd_t': Matrix([[0],[0]])}

m_post = utils.evaluate_expr(p_xt_update.mean.to_full_expr(), d)
cov_post = utils.evaluate_expr(p_xt_update.cov.to_full_expr(), d)
```

The values given are from a 2-D tracking example given in [Murphy, 2012] (Figure 18.1 and Section 18.2.1). Also d['y_t'] is specified later when we sample from the state-space for $t = 0$.

We can see that this feature is relatively easy to use as we only need to specify the dictionary once and it can then be used as many times as possible. Also, the conversion to numpy.ndarrays allows for faster operations than can be perform with SymPy's Matrix class.

4.2 Generation of LaTeX expressions

As already mentioned above, we can generate LaTeX expressions by calling the utils.matLatex function. This takes in a SuperMat expression, general SymPy expressions or an MVG object and returns a full LaTeX expression that can then be compiled directly.

Taking the Kalman Filter example, we can generate the LaTeX code for p_xt_predict ($p(x_t|y_{1:t-1})$) using utils.matLatex(p_xt_predict) to give

```
\begin{align*}
p\left(\mathbf{x_t}|\mathbf{y_{1:t-1}}\right)&= \mathcal{N}\left(\mathbf{x_t};
\mathbf{m}_{\mathbf{x_t}|\mathbf{y_{1:t-1}}},\mathbf{\Sigma}_{\mathbf{x_t}|\mathbf{y_{1:t
    -1}}}\right)\\
\mathbf{m}_{\mathbf{x_t}|\mathbf{y_{1:t-1}}} &= \mathbf{A} \mathbf{m_{t-1|t-1}} + \mathbf{b_
    t}\\
\mathbf{\Sigma}_{\mathbf{x_t}|\mathbf{y_{1:t-1}}} &= \mathbf{Q} + \mathbf{A} \mathbf{P_{t-1|
    t-1}} \mathbf{A}^T\\
\end{align*}
```

which is then compiled to give

$$
\begin{aligned}
p\left(\mathbf{x_t}|\mathbf{y_{1:t-1}}\right) &= \mathcal{N}\left(\mathbf{x_t}; \mathbf{m}_{\mathbf{x_t}|\mathbf{y_{1:t-1}}}, \mathbf{\Sigma}_{\mathbf{x_t}|\mathbf{y_{1:t-1}}}\right)\\
\mathbf{m}_{\mathbf{x_t}|\mathbf{y_{1:t-1}}} &= \mathbf{A}\mathbf{m_{t-1|t-1}} + \mathbf{b_t}\\
\mathbf{\Sigma}_{\mathbf{x_t}|\mathbf{y_{1:t-1}}} &= \mathbf{Q} + \mathbf{A}\mathbf{P_{t-1|t-1}}\mathbf{A}^T
\end{aligned}
$$

This allows the MVGs we produce to be rendered into an easily readable format. This allows us to check for errors in the MVG by simply calling one line. For example, we can perform a derivation in a Jupyter notebook where we can print the LaTeX code then compile and display it using one line: display(Latex(utils.matLatex(p_xt_predict))).

4.3 Expression simplification

As highlighted in the Kalman Filter equation above, once we obtain the expressions for the parameters of $p(x_t|y_{1:t})$, we need to apply some substitutions to obtain the final recursive equations. Therefore, the first step to creating this functionality for simplification is to write functions that allow for collecting matrix terms and for arbitrary sub-expressions in an expression to be replaced with a shorter one. We do this with the utility functions utils.replace, which takes in a SymPy/SymGP expression and a dictionary matching sub-expressions to their corresponding substitution and utils.collect which takes in an expression, a list of sub-expressions to collect and a list specifying whether the sub-expressions should be collected on the left or right if that is a possibility.

To give an example of their use, we can first collect $Q + AP_{t-1|t-1}A^T$ in the covariance expression of $p(x_t|y_{1:t})$ (p_xt_update.covar) and then replace it with automatically created name for the covariance of $p(x_t|y_{1:t-1})$ as they are the same expression:

```
# Collect Q + A*P_{t-1|t-1}*A' on the right
post_cov = utils.collect(p_xt_update.covar.to_full_expr(),[Q+A*F_prev*A.T],'right')
# >> (I + (-1)*(Q + A*P_{t-1|t-1}*A')*C'*(R + C*(Q + A*P_{t-1|t-1}*A')*C')^-1*C)* ...
#  (Q + A*P_{t-1|t-1}*A')

# Replace Q + A*P_{t-1|t-1}*A' with S_{x_t|y_{1:t-1}}, the covariance of p_xt_predict
repl_dict = {Q + A*F_prev*A.T: p_xt_predict.covar}
new_cov_1 = utils.replace(post_cov, repl_dict).doit()
# >> (I + (-1)*S_{x_t|y_{1:t-1}}*C'*(R + C*S_{x_t|y_{1:t-1}}*C')^-1*C)*S_{x_t|y_{1:t-1}}
```

We can then keep on applying replacements to obtain an expression for the covariance of $p(x_t|y_{1:t})$ similar to that in 1.4:

```
# Replace R + C*S_{x_t|y_{1:t-1}}*C' with S_{t}
S_t = Constant('S_{t}',R.shape[0],R.shape[1], full_expr=R + C*p_xt_predict.covar*C.T)
repl_dict = {R + C*p_xt_predict.covar*C.T: S_t}
new_cov_2 = utils.replace(new_cov_1, repl_dict).doit()
# >> (I + (-1)*S_{x_t|y_{1:t-1}}*C'*S_{t}^-1*C)*S_{x_t|y_{1:t-1}}

# Replace S_{x_t|y_{1:t-1}}*C'*S_{t}^{-1} with K_{t}
K_t = Constant('K_{t}', x_t.shape[0], C.shape[0], full_expr=p_xt_predict.covar*C.T*S_t.I)
repl_dict = {p_xt_predict.covar*C.T*S_t.I: K_t}
new_cov_3 = utils.replace(new_cov_2, repl_dict).doit()
# >> (I + (-1)*K_{t}*C)*S_{x_t|y_{1:t-1}}
```

This last expression can be turned into LaTeX to give:

$$\mathbf{\Sigma_{x_t|y_{1:t-1},y_t}} = (\mathbf{I} - \mathbf{K_t C})\,\mathbf{S_{x_t|y_{1:t-1}}} \tag{4.1}$$

where $\mathbf{S_{x_t|y_{1:t-1}}}$ is the same as $P_{t|t-1}$ in 1.4. This is very similar to 1.4 thus illustrating the success of our method and its reliability in allowing us to obtain any suitable simplified expression with the replacement names we desire.

However, this process can become tedious as many steps may need to be taken to get to the final expression. A way to alleviate this would be to automate this process but it comes with the problem of knowing which simplifications to make at each stage. One way to attack this is to make substitutions from the lowest level in the expression tree upwards. By this we mean that we substitute for sub-expressions at the lowest level in the tree then using the shortest simplified expression from these substitutions, make more substitutions at the second lowest level and repeat the process up until the highest level. We select the shortest expression based on the number of individuals symbols in the expression tree i.e. the number of leaves on the tree. We also collect terms to create shorter expressions. All this is implemented in the utils.simplify function which works as in Algorithm 1

Algorithm 1 Expression Simplification

```
1: function SIMPLIFY(expr, d)
2:     Initialise as empty a list of candidate simplified expressions (simps), dictionary of substitutions that will be made (subs)
   and a list of expressions that have been substituted thus far (used_subs).
3:     depth ← GETMAXDEPTH(expr)                                   ▷ The maximum depth of tree
4:     min_expr ← expr
5:     for i ← 0, depth do
6:         d ← depth − i
7:         Get sub-expressions at all depths up to depth and store in exprs_by_depth
8:         sub_exprs ← exprs_by_depth[d]                           ▷ sub-expressions at depth d
9:         for s in sub_exprs do
10:            if s is suitable for replacement and s is not in used_subs then   ▷ See text for details
11:                Update used_subs with s
12:                Replace (utils.replace) s with new SuperMatSymbol, r, and store in simp_expr
13:                Append simp_expr to simps
14:                Add s (key) and r (value) to subs if it doesn't exist
15:                if simp_expr has fewer individual matrix symbols than min_expr then
16:                    min_expr ← simp_expr
17:                if simp_expr is a MatAdd then
18:                    Find terms in simp_expr that can be collected.
19:                    Apply utils.collect to simp_expr for each of these terms. Store in simped.
20:                    if simped is not in simps then
21:                        Append simped to simps
       return simps, subs
```

In Algorithm 1, on line 10, we consider a sub-expression (s) to be a candidate for replacement if:

- The number of *other* occurrences of s in the expression tree are greater than zero **or** s matches the special case $E^{-1}F(H - GE^{-1}F)^{-1}$. This corresponds to a form of matrix expression that can be obtained in derivations of some models. For example, this is found in the Kalman Filter equation, $K_t = P_{t|t-1}C^T(CP_{t|t-1}C^T + R)^{-1}$ (1.2) and so it helps to detect this expression and substitute for it. We check for this second case using the utility function acceptInvLemma due to the expressions relation to the matrix inversion lemma (see Section 4.3.4.2 in [Murphy, 2012]).

- The expression s has not been substituted before.

As an example of the application of this algorithm, we apply the function to the full expression for the covariance of $p(x_t|y_{1:t})$ obtained from the final operation in SymGP:

```
p_xt_update = p_yt_xt.condition([y_t])
print(p_xt_update.covar.to_full_expr())
# >> Q + A*P_{t-1|t-1}*A' + (-1)*(Q + A*P_{t-1|t-1}*A')*C'*
#    (R + C*(Q + A*P_{t-1|t-1}*A')*C')^-1*C*(Q + A*P_{t-1|t-1}*A')

simps, subs = utils.simplify(p_xt_update.covar.to_full_expr())
# >> simps = [(I + (-1)*A_{2}*C)*A_{1},
#             (-1)*A_{2}*C*A_{1} + A_{1},
#             A_{1}*(I + (-1)*C'*(R + C*A_{1}*C')^-1*C*A_{1}),
#             (I + (-1)*A_{1}*C'*(R + C*A_{1}*C')^-1*C)*A_{1},
#             (-1)*A_{1}*C'*(R + C*A_{1}*C')^-1*C*A_{1} + A_{1},
#             (Q + A_{0})*(I + (-1)*C'*(R + C*(Q + A_{0})*C')^-1*C*(Q + A_{0})),
#             (I + (-1)*(Q + A_{0})*C'*(R + C*(Q + A_{0})*C')^-1*C)*(Q + A_{0}),
```

```
#             Q + (-1)*(Q + A_{0})*C'*(R + C*(Q + A_{0})*C')^-1*C*(Q + A_{0}) + A_{0}]
#
# >> subs = {Q + A_{0}: A_{1}
#            A_{1}*C'*(R + C*A_{1}*C')^-1: A_{2}
#            A*P_{t-1|t-1}*A': A_{0}}
```

The function produces a list of simplifications ordered by the number of individual symbols (MatrixSymbol, Transpose, Inverse, etc.) in the expression. We can make a number of observations from the output:

- The same collection of symbols is performed at different levels of substitutions e.g. A_1 and the expression it substitutes, Q + A_0, are collected similarly, on the left and the right at different levels of the tree (see last six outputs of simps). This is because when we reset *min_expr* in the code, we don't use the simplified expressions obtained from collecting symbols and so the same collection is performed at different levels of the tree. We do this because the initial **for** loop works on the depth of the original expression tree and using a expression that has a different structure could cause the function to fail.

- The simplest expression we obtain ((I + (-1)*A_2*C)*A_1) is similar to the expression for the posterior covariance in 1.4 with A_2 corresponding to K_t, A_1 to $P_{t|t-1}$ and A_0 to $AP_{t-1|t-1}A'$.

Note that we can obtain more simplified expressions by relaxing the constraint that the number of occurrences in an expression has to be greater than one. We can substitute for single-occurring expressions if it is more convenient.

These results indicate the benefits of using an automatic simplification algorithm because we can obtain useful simplification by applying a simple set of rules. This was motivated from noticing that the sub-expressions in the mean and covariance expressions obtained after operations between multivariate Gaussians generally take the forms (upper case corresponds to matrices, lower case to vectors):

- $Ax + b$
- AFA'
- $E + FGF'$
- $(E - FH^{-1}G)^{-1}FH^{-1}/E^{-1}F(H - GE^{-1}F)^{-1}$ (Matrix inversion lemma, see Section 4.3.4.2 in [Murphy, 2012])

In SymPy terms, these generally correspond to MatMuls and MatAdds and so a good strategy would be to simply replace expressions of these forms that appear repeatedly. The last case can then be detected by parsing matrix expressions to apply their special simplifications.

Overall, this strategy provides a good starting point for more sophisticated simplification algorithms that can better determine the simplification a user requires.

Chapter 5

Conclusions & Future Work

In this report, we have outlined a solution to the problem of tedious hand calculations involved in the derivation of inference and prediction equations for linear Gaussian models. The solution has taken the form of a software library that uses a computer algebra system to manipulate and output symbolic expressions. The library was made to be flexible by having the multivariate Gaussian representations work with any number of variables. It is able to abstract the details that a symbol may represent by using classes to create this abstraction. It was made easy to use by only requiring a few lines of code as opposed to many more lines of handwritten calculations to produce the same equations. The output from the library (from the multivariate Gaussian objects) can be easily used for various other functions such as LaTeX code generation and numerical evaluation.

Working on the library has brought out several ideas as to where the project can go next:

- **Extension of utilities**. We can extend our current utility functions and develop new ones. The numerical evaluation function can be improved to produce code that can be run in different programming environments i.e. we can produce a function and then pass our arguments to it in several programming languages. This can be done in a similar manner to the way we print LaTeX whereby we write a printer that outputs code in other languages.

  The simplification algorithm can also be improved by using better measurements of the desirability of a simplified expression and by using a more intelligent sequence of substitutions so as to get to a required result. For example, we could allow the user to specify examples of simplified expressions that they desire and the program could learn to transform unsimplified expressions similarly.

- **More MVG operations**. The range of models that can be supported can be increased by adding operations such as the summation of Gaussians which could allow the li-

brary to support models such as mixture of Gaussians or Linear Discriminant Analysis (LDA) [Murphy, 2012] or by introducing some non-Gaussian distributions such as a multinomial likelihood.

- **GUI**. Having to use the Python API to use the library may be not be ideal for people who want to use the software but are not versed in Python. Therefore, a better way to provide interaction would be to use a graphical user interface (GUI) that would be much easier to understand and manipulate. It can also offer a visual representation of the Gaussian distributions which eases interpretability of the results.

Bibliography

[Meurer et al., 2017] Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A. (2017). Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103.

[Murphy, 2012] Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press.

[Quiñonero Candela and Rasmussen, 2005] Quiñonero Candela, J. and Rasmussen, C. E. (2005). A unifying view of sparse approximate gaussian process regression. *J. Mach. Learn. Res.*, 6:1939–1959.

[Rasmussen and Williams, 2005] Rasmussen, C. E. and Williams, C. K. I. (2005). *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

[Tucker, 1958] Tucker, L. (1958). An inter-battery method of factor analysis. *Psychometrika*, 23(2):111–136.

[Wiskott and Sejnowski, 2002] Wiskott, L. and Sejnowski, T. J. (2002). Slow feature analysis: Unsupervised learning of invariances. *Neural Comput.*, 14(4):715–770.