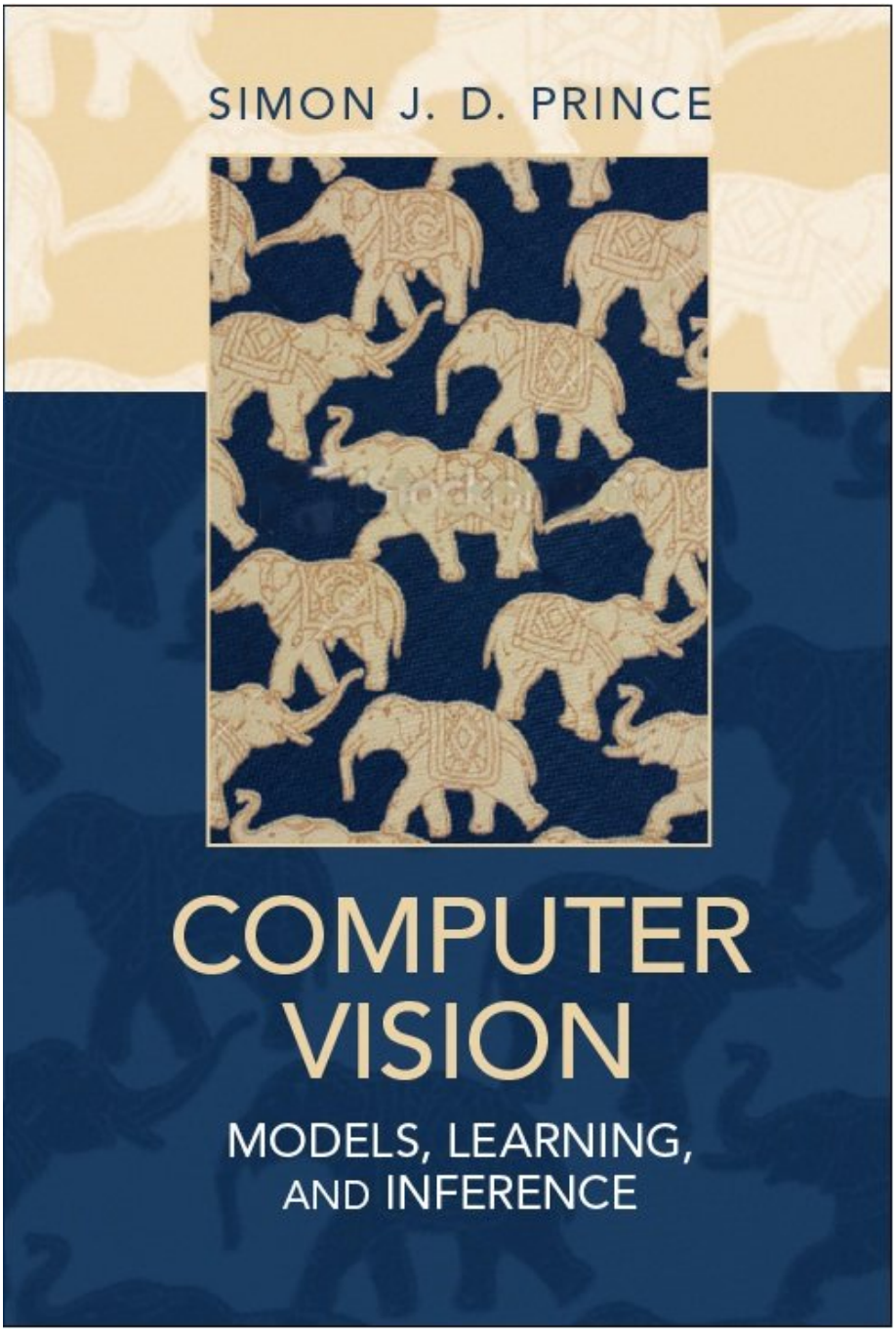

Algorithms booklet

November 2, 2013

Algorithms booklet

This document accompanies the book "Computer vision: models, learning, and inference" by Simon J.D. Prince. It contains concise descriptions of almost all of the models and algorithms in the book. The goal is to provide sufficient information to implement a naive version of each method. This information was published separately from the main book because (i) it would have impeded the clarity of the main text and (ii) on-line publishing means that I can update the text periodically and eliminate any mistakes.

In the main, this document uses the same notation as the main book (see Appendix A for a summary). In addition, we also use the following conventions:

- When two matrices are concatenated horizontally, we write $\mathbf{C} = [\mathbf{A}, \mathbf{B}]$.
- When two matrices are concatenated vertically, we write $\mathbf{C} = [\mathbf{A}; \mathbf{B}]$.
- The function $\text{argmin}_{\mathbf{x}} f[\mathbf{x}]$ returns the value of the argument $\mathbf{x}$ that minimizes $f[\mathbf{x}]$. If $\mathbf{x}$ is discrete then this should be done by exhaustive search. If $\mathbf{x}$ is continuous, then it should be done by gradient descent and I usually supply the gradient and Hessian of the function to help with this.
- The function $\delta[\mathbf{x}]$ for discrete x returns 1 when the argument x is 0 and returns 0 otherwise.
- The function $\textbf{diag}[\mathbf{A}]$ returns a column vector containing the elements on the diagonal of matrix $\mathbf{A}$.
- The function $\textbf{zeros}[I, J]$ creates an $I \times J$ matrix that is full of zeros.

As a final note, I should point out that this document has not yet been checked very carefully. I'm looking for volunteers to help me with this. There are two main ways you can help. First, please mail me at s.prince@cs.ucl.ac.uk if you manage to successfully implement one of these methods. That way I can be sure that the description is sufficient. Secondly, please also mail me if you if you have problems getting any of these methods to work. It's possible that I can help, and it will help me to identify ambiguities and errors in the descriptions.

Simon Prince

List of Algorithms

Algorithm 4.1: Maximum likelihood learning of normal distribution

The univariate normal distribution is a probability density model suitable for describing continuous data x in one dimension. It has pdf

$$Pr(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-0.5(x-\mu)^2/\sigma^2\right],$$

where the parameter μ denotes the mean and σ^2 denotes the variance.

Algorithm 4.1: Maximum likelihood learning for normal distribution

Input : Training data $\{x_i\}_{i=1}^I$
Output: Maximum likelihood estimates of parameters $\boldsymbol{\theta} = \{\mu, \sigma^2\}$
begin
  `// Set mean parameter`
  $\mu = \sum_{i=1}^I x_i / I$
  `// Set variance`
  $\sigma^2 = \sum_{i=1}^I (x_i - \hat{\mu})^2 / I$
end

Algorithm 4.2: MAP learning of univariate normal parameters

The conjugate prior to the normal distribution is the normal-scaled inverse gamma distribution which has pdf

$$Pr(\mu, \sigma^2) = \frac{\sqrt{\gamma}}{\sigma\sqrt{2\pi}} \frac{\beta^\alpha}{\Gamma(\alpha)} \left(\frac{1}{\sigma^2}\right)^{\alpha+1} \exp\left[-\frac{2\beta + \gamma(\delta-\mu)^2}{2\sigma^2}\right],$$

with hyperparameters $\alpha, \beta, \gamma > 0$ and $\delta \in [-\infty, \infty]$.

Algorithm 4.2: MAP learning for normal distribution with conjugate prior

Input : Training data $\{x_i\}_{i=1}^I$, Hyperparameters $\alpha, \beta, \gamma, \delta$
Output: MAP estimates of parameters $\boldsymbol{\theta} = \{\mu, \sigma^2\}$
begin
  `// Set mean parameter`
  $\mu = (\sum_{i=1} x_i + \gamma\delta)/(I + \gamma)$
  `// Set variance`
  $\sigma^2 = (\sum_{i=1}^I (x_i - \mu)^2 + 2\beta + \gamma(\delta - \mu)^2)/(I + 3 + 2\alpha)$
end

Algorithm 4.3: Bayesian approach to univariate normal distribution

In the Bayesian approach to fitting the univariate normal distribution we again use a normal-scaled inverse gamma prior. In the learning stage we compute a normal inverse gamma distribution over the mean and variance parameters. The predictive distribution for a new datum is computed by integrating the predictions for a given set of parameters weighted by the probability of those parameters being present.

Algorithm 4.3: Bayesian approach to normal distribution

Input : Training data $\{x_i\}_{i=1}^{I}$, Hyperparameters $\alpha, \beta, \gamma, \delta$, Test data x^*
Output: Posterior parameters $\{\tilde{\alpha}, \tilde{\beta}, \tilde{\gamma}, \tilde{\delta}\}$, predictive distribution $Pr(x^*|x_{1...I})$
begin
`// Compute normal inverse gamma posterior over normal parameters`
$\tilde{\alpha} = \alpha + I/2$
$\tilde{\beta} = \sum_i x_i^2/2 + \beta + \gamma\delta^2/2 - (\gamma\delta + \sum_i x_i)^2/(2\gamma + 2I)$
$\tilde{\gamma} = \gamma + I$
$\tilde{\delta} = (\gamma\delta + \sum_i x_i)/(\gamma + I)$
`// Compute intermediate parameters`
$\breve{\alpha} = \tilde{\alpha} + 1/2$
$\breve{\beta} = x^{*2}/2 + \tilde{\beta} + \tilde{\gamma}\tilde{\delta}^2/2 - (\tilde{\gamma}\tilde{\delta} + x^*)^2/(2\tilde{\gamma} + 2)$
$\breve{\gamma} = \tilde{\gamma} + 1$
`// Evaluate new datapoint under predictive distribution`
$Pr(x^*|\mathbf{x}_{1...I}) = \sqrt{\tilde{\gamma}}\tilde{\beta}^{\tilde{\alpha}}\Gamma[\breve{\alpha}] / \left(\sqrt{2\pi}\sqrt{\breve{\gamma}}\breve{\beta}^{\breve{\alpha}}\Gamma[\tilde{\alpha}]\right)$
end

Algorithm 4.4: ML learning of categorical parameters

The categorical distribution is a probability density model suitable for describing discrete multivalued data $x \in \{1, 2, \ldots K\}$. It has pdf

$$Pr(x = k) = \lambda_k,$$

where the parameter λ_k denotes the probability of observing category k.

Algorithm 4.4: Maximum likelihood learning for categorical distribution

Input : Multi-valued training data $\{x_i\}_{i=1}^{I}$
Output: ML estimate of categorical parameters $\boldsymbol{\theta} = \{\lambda_1 \ldots \lambda_k\}$
begin
  for *k=1* **to** K **do**
    $\lambda_k = \sum_{i=1}^{I} \delta[\mathbf{x}_i - k]/I$
  end
end

Algorithm 4.5: MAP learning of categorical parameters

For MAP learning of the categorical parameters, we need to define a prior and to this end, we choose the Dirichlet distribution:

$$Pr(\lambda_1 \ldots \lambda_K) = \frac{\Gamma[\sum_{k=1}^K \alpha_k]}{\prod_{k=1}^K \Gamma[\alpha_k]} \prod_{k=1}^K \lambda_k^{\alpha_k - 1},$$

where $\Gamma[\bullet]$ is the Gamma function and $\{\alpha_k\}_{k=1}^K$ are hyperparameters.

Algorithm 4.5: MAP learning for categorical distribution with conjugate prior

Input : Binary training data $\{x_i\}_{i=1}^I$, Hyperparameters $\{\alpha_k\}_{k=1}^K$
Output: MAP estimates of parameters $\boldsymbol{\theta} = \{\lambda_k\}_{k=1}^K$
begin
  for *k=1* **to** K **do**
    $N_k = \sum_{i=1}^I \delta[\mathbf{x}_i - k])$
    $\lambda_k = (N_k - 1 + \alpha_k)/(I - K + \sum_{k=1}^K \alpha_k)$
  end
end

Algorithm 4.6: Bayesian approach to categorical distribution

In the Bayesian approach to fitting the categorical distribution we again use a Dirichlet prior. In the learning stage we compute a probability distribution over K categorical parameters, which is also a Dirichlet distribution. The predictive distribution for a new datum is based on a weighted sum of the predictions for all possible parameter values where the weights used are based on the Dirichlet distribution computed in the learning stage.

Algorithm 4.6: Bayesian approach to categorical distribution

Input : Categorical training data $\{x_i\}_{i=1}^I$, Hyperparameters $\{\alpha_k\}_{k=1}^K$
Output: Posterior parameters $\{\tilde{\alpha}_k\}_{k=1}^K$, predictive distribution $Pr(x^*|\mathbf{x}_{1\ldots I})$
begin
  `// Compute categsorical posterior over` λ
  for *k=1* **to** K **do**
    $\tilde{\alpha}_k = \alpha_k + \sum_{i=1}^I \delta[\mathbf{x}_i - k]$
  end
  `// Evaluate new datapoint under predictive distribution`
  for *k=1* **to** K **do**
    $Pr(x^* = k|\mathbf{x}_{1\ldots I}) = \tilde{\alpha_k}/(\sum_{m=1}^K \tilde{\alpha_m})$
  end
end

Algorithm 6.1: Basic generative classifier

Consider the situation where we wish to assign a label $w \in \{1, 2, \ldots K\}$ based on an observed multivariate measurement vector $\mathbf{x}_i$. We model the class conditional density functions as normal distributions so that

$$Pr(\mathbf{x}_i|w_i = k) = \text{Norm}_{\mathbf{x}_i}[\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k],$$

with prior probabilities over the world state defined by

$$Pr(w_i) = \text{Cat}_{w_i}[\boldsymbol{\lambda}].$$

In the learning phase, we fit the parameters $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k^2$ of the k^{th} class conditional density function $Pr(\mathbf{x}_i|w_i = k)$ from just the subset of data $\mathcal{S}_k = \{x_i : w_i = k\}$ where the k^{th} state was observed. We learn the prior parameter $\boldsymbol{\lambda}$ from the training world states $\{w_i\}_{i=1}^I$. Here we have used the maximum likelihood approach in both cases.

The inference algorithm takes new datum $\mathbf{x}^*$ and returns the posterior $Pr(w^*|\mathbf{x}^*, \boldsymbol{\theta})$ over the world state w^* using Bayes' rule,

$$Pr(w^*|\mathbf{x}^*) = \frac{Pr(\mathbf{x}^*|w^*)Pr(w^*)}{\sum_{w^*=1}^K Pr(\mathbf{x}^*|w^*)Pr(w^*)}.$$

Algorithm 6.1: Basic Generative classifier

Input : Training data $\{\mathbf{x}_i, w_i\}_{i=1}^I$, new data example $\mathbf{x}^*$
Output: ML parameters $\boldsymbol{\theta} = \{\lambda_{1\ldots K}, \boldsymbol{\mu}_{1\ldots K}, \boldsymbol{\Sigma}_{1\ldots K}\}$, posterior probability $Pr(w^*|\mathbf{x}^*)$
begin
  `// For each training class`
  for k=*1* **to** K **do**
    `// Set mean`
    $\boldsymbol{\mu}_k = (\sum_{i=1}^I \mathbf{x}_i \delta[w_i - k])/(\sum_{i=1}^I \delta[w_i - k])$
    `// Set variance`
    $\boldsymbol{\Sigma}_k = (\sum_{i=1}^I (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T \delta[w_i - k])/(\sum_{i=1}^I \delta[w_i - k])$
    `// Set prior`
    $\lambda_k = \sum_{i=1}^I \delta[w_i - k]/I$
  end
  `// Compute likelihoods for each class for a new datapoint`
  for k=*1* **to** K **do**
    $l_k = \text{Norm}_{\mathbf{x}^*}[\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k]$
  end
  `// Classify new datapoint using Bayes' rule`
  for k=*1* **to** K **do**
    $Pr(w^* = k|\mathbf{x}^*) = l_k\lambda_k / \sum_{m=1}^K l_m\lambda_m$
  end
end

Algorithm 7.1: Fitting mixture of Gaussians

The mixture of Gaussians (MoG) is a probability density model suitable for data $\mathbf{x}$ in D dimensions. The data is described as a weighted sum of K normal distributions

$$Pr(\mathbf{x}|\boldsymbol{\theta}) = \sum_{k=1}^{K} \lambda_k \text{Norm}_{\mathbf{x}}[\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k],$$

where $\boldsymbol{\mu}_{1\ldots K}$ and $\boldsymbol{\Sigma}_{1\ldots K}$ are the means and covariances of the normal distributions and $\lambda_{1\ldots K}$ are positive valued weights that sum to one.

The MoG is fit using the EM algorithm. In the E-step, we compute the posterior distribution over a hidden variable h_i for each observed data point $\mathbf{x}_i$. In the M-step, we iterate through the K components, updating the mean $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ for each and also update the weights $\{\lambda_k\}_{k=1}^{K}$.

Algorithm 7.1: Maximum likelihood learning for mixtures of Gaussians

Input : Training data $\{\mathbf{x}_i\}_{i=1}^{I}$, number of clusters K
Output: ML estimates of parameters $\boldsymbol{\theta} = \{\lambda_{1\ldots K}, \boldsymbol{\mu}_{1\ldots K}, \boldsymbol{\Sigma}_{1\ldots K}\}$
begin
  Initialize $\boldsymbol{\theta} = \boldsymbol{\theta}_0$ [a]
  repeat
    `// Expectation Step`
    for i=*1* **to** I **do**
      for k=*1* **to** K **do**
        $l_{ik} = \lambda_k \text{Norm}_{\mathbf{x}_i}[\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k]$ `// numerator of Bayes' rule`
      end
      `// Compute posterior (responsibilities) by normalizing`
      for k=*1* **to** K **do**
        $r_{ik} = l_{ik}/(\sum_{k=1}^{K} l_{ik})$
      end
    end
    `// Maximization Step` [b]
    for k=*1* **to** K **do**
      $\lambda_k^{[t+1]} = (\sum_{i=1}^{I} r_{ik})/(\sum_{k=1}^{K}\sum_{i=1}^{I} r_{ik})$
      $\boldsymbol{\mu}_k^{[t+1]} = (\sum_{i=1}^{I} r_{ik}\mathbf{x}_i)/(\sum_{i=1}^{I} r_{ik})$
      $\boldsymbol{\Sigma}_k^{[t+1]} = (\sum_{i=1}^{I} r_{ik}(\mathbf{x}_i - \boldsymbol{\mu}_k^{[t+1]})(\mathbf{x}_i - \boldsymbol{\mu}_k^{[t+1]})^T)/(\sum_{i=1}^{I} r_{ik})$.
    end
    `// Compute Data Log Likelihood and EM Bound`
    $L = \sum_{i=1}^{I} \log\left[\sum_{k=1}^{K} \lambda_k \text{Norm}_{\mathbf{x}_i}[\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k]\right]$
    $B = \sum_{i=1}^{I}\sum_{k=1}^{K} r_{ik} \log\left[\lambda_k \text{Norm}_{\mathbf{x}_i}[\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k]/r_{ik}\right]$
  until *No further improvement in L*
end

[a] One possibility is to set the weights $\lambda_\bullet = 1/K$, the means $\boldsymbol{\mu}_\bullet$ to the values of K randomly chosen datapoints and the variances $\boldsymbol{\Sigma}_\bullet$ to the variance of the whole dataset.
[b] For a diagonal covariance retain only the diagonal of the $\boldsymbol{\Sigma}_k$ update.

Algorithm 7.2: Fitting the t-distribution

The t-distribution is a robust (long-tailed) distribution with pdf

$$Pr(\mathbf{x}) = \frac{\Gamma\left(\frac{\nu+D}{2}\right)}{(\nu\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}\Gamma\left(\frac{\nu}{2}\right)}\left(1+\frac{(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})}{\nu}\right)^{-(\nu+D)/2},$$

where $\boldsymbol{\mu}$ is the mean of the distribution $\boldsymbol{\Sigma}$ is a matrix that controls the spread, ν is the degrees of freedom, and D is the dimensionality of the input data.
We use the EM algorithm to fit the parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\Sigma}, \nu\}$. In the E-step, we compute the gamma-distributed posterior over the hidden variable h_i for each observed data point $\mathbf{x}_i$. In the M-step we update the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ in closed form, but must perform an explicit line search to update ν using the criterion:

$$\text{tCost}\left[\nu, \{E[h_i], E[\log[h_i]]\}_{i=1}^I\right] = -\sum_{i=1}^I \frac{\nu}{2}\log\left[\frac{\nu}{2}\right] + \log\left[\Gamma\left[\frac{\nu}{2}\right]\right] - \left(\frac{\nu}{2}-1\right)E[\log[h_i]] + \frac{\nu}{2}E[h_i].$$

Algorithm 7.2: Maximum likelihood learning for t-distribution

Input : Training data $\{\mathbf{x}_i\}_{i=1}^I$
Output: Maximum likelihood estimates of parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\Sigma}, \nu\}$
begin
  Initialize $\boldsymbol{\theta} = \boldsymbol{\theta}_0$ [a]
  repeat
    `// Expectation step`
    for *i=1* **to** I **do**
      $\delta_i = (\mathbf{x}_i - \boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu})$
      $\mathsf{E}[h_i] = (\nu + D)/(\nu + \delta_i)$
      $\mathsf{E}[\log[h_i] = \Psi[\nu/2 + D/2] - \log[\nu/2 + \delta_i/2]$
    end
    `// Maximization step`
    $\boldsymbol{\mu} = (\sum_{i=1}^I E[h_i]\mathbf{x}_i)/(\sum_{i=1}^I E[h_i])$
    $\boldsymbol{\Sigma} = (\sum_{i=1}^I E[h_i](\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T)/(\sum_{i=1}^I E[h_i])$
    $\nu = \text{argmin}_\nu[\mathsf{tCost}[\nu, \{E[h_i], E[\log[h_i]]\}_{i=1}^I]]$
    `// Compute data log Likelihood`
    for *i=1* **to** I **do**
      $\delta_i = (\mathbf{x}_i - \boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu})$
    end
    $L = I\log[\Gamma[(\nu + D)/2]] - ID\log[\nu\pi]/2 - I\log[|\boldsymbol{\Sigma}|]/2 - I\log[\Gamma[\nu/2]]$
    $L = L - (\nu + D)\sum_{i=1}^I \log[1 + \delta_i/\nu]/2$
  until *No further improvement in L*
end

[a] One possibility is to initialize the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ to the mean and variance of the data and set the initial degrees of freedom to a large value say $\nu = 1000$.

Algorithm 7.3: Fitting a factor analyzer

The factor analyzer is a probability density model suitable for data $\mathbf{x}$ in D dimensions. It has pdf

$$Pr(\mathbf{x}|\boldsymbol{\theta}) = \text{Norm}_{\mathbf{x}}[\boldsymbol{\mu}, \boldsymbol{\Phi}\boldsymbol{\Phi}^T + \boldsymbol{\Sigma}],$$

where $\boldsymbol{\mu}$ is a $D \times 1$ mean vector, $\boldsymbol{\Phi}$ is a $D \times K$ matrix containing the K factors $\{\boldsymbol{\phi}_k\}_{k=1}^K$ in its columns and $\boldsymbol{\Sigma}$ is a diagonal matrix of size $D \times D$.

The factor analyzer is fit using the EM algorithm. In the E-step, we compute the posterior distribution over the hidden variable $\mathbf{h}_i$ for each data example $\mathbf{x}_i$ and extract the expectations $E[\mathbf{h}_i]$ and $E[\mathbf{h}_i\mathbf{h}_i^T]$. In the M-step, we use these distributions in closed-form updates for the basis function matrix $\boldsymbol{\Phi}$ and the diagonal noise term $\boldsymbol{\Sigma}$.

Algorithm 7.3: Maximum likelihood learning for factor analyzer

Input : Training data $\{\mathbf{x}_i\}_{i=1}^I$, number of factors K
Output: Maximum likelihood estimates of parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\Phi}, \boldsymbol{\Sigma}\}$
begin
  Initialize $\boldsymbol{\theta} = \boldsymbol{\theta}_0$ [a]
  `// Set mean`
  $\boldsymbol{\mu} = \sum_{i=1}^I \mathbf{x}_i / I$
  repeat
    `// Expectation Step`
    for *i=1* **to** I **do**
      $\mathsf{E}[\mathbf{h}_i] = (\boldsymbol{\Phi}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\Phi} + \mathbf{I})^{-1}\boldsymbol{\Phi}^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu})$
      $\mathsf{E}[\mathbf{h}_i\mathbf{h}_i^T] = (\boldsymbol{\Phi}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\Phi} + \mathbf{I})^{-1} + \mathsf{E}[\mathbf{h}_i]\mathsf{E}[\mathbf{h}_i]^T$
    end
    `// Maximization Step`
    $\boldsymbol{\Phi} = \left(\sum_{i=1}^I (\mathbf{x}_i - \boldsymbol{\mu})\mathsf{E}[\mathbf{h}_i]^T\right)\left(\sum_{i=1}^I \mathsf{E}[\mathbf{h}_i\mathbf{h}_i^T]\right)^{-1}$
    $\boldsymbol{\Sigma} = \textbf{diag}\left[\sum_{i=1}^I (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T - \boldsymbol{\Phi}\mathsf{E}[\mathbf{h}_i](\mathbf{x}_i - \boldsymbol{\mu})^T\right]/I$
    `// Compute Data Log Likelihood`[b]
    $L = \sum_{i=1}^I \log\left[\mathsf{Norm}_{\mathbf{x}_i}[\boldsymbol{\mu}, \boldsymbol{\Phi}\boldsymbol{\Phi}^T + \boldsymbol{\Sigma}]\right]$
  until *No further improvement in L*
end

[a] It is usual to initialize $\boldsymbol{\Phi}$ to random values. The D diagonal elements of $\boldsymbol{\Sigma}$ can be initialized to the variances of the D data dimensions.

[b] In high dimensions it is worth reformulating the covariance of this matrix using the Sherman-Morrison-Woodbury relation (matrix inversion lemma) .

Algorithm 8.1: ML fitting of linear regression model

The linear regression model describes the world w as a normal distribution. The mean of this distribution is a linear function $\phi_0 + \boldsymbol{\phi}^T\mathbf{x}$ and the variance is constant. In practice we add a 1 to the start of every data vector $\mathbf{x}_i \leftarrow [1 \quad \mathbf{x}_i^T]^T$ and attach the y-intercept ϕ_0 to the start of the gradient vector $\boldsymbol{\phi} \leftarrow [\phi_0 \quad \boldsymbol{\phi}^T]^T$ and write

$$Pr(w_i|\mathbf{x}_i, \boldsymbol{\theta}) = \text{Norm}_{w_i}\left[\boldsymbol{\phi}^T\mathbf{x}_i, \sigma^2\right].$$

In the learning algorithm, we work with the matrix $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2 \ldots \mathbf{x}_I]$ which contains all of the training data examples in its columns and the world vector $\mathbf{w} = [w_1, w_2 \ldots w_I]^T$ which contains the training world states.

Algorithm 8.1: Maximum likelihood learning for linear regression

Input : $(D+1)\times I$ data matrix $\mathbf{X}$, $I\times 1$ world vector $\mathbf{w}$
Output: Maximum likelihood estimates of parameters $\boldsymbol{\theta} = \{\boldsymbol{\phi}, \sigma^2\}$
begin
  `// Set gradient parameter`
  $\boldsymbol{\phi} = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{w}$
  `// Set variance parameter`
  $\sigma^2 = (\mathbf{w} - \mathbf{X}^T\boldsymbol{\phi})^T(\mathbf{w} - \mathbf{X}^T\boldsymbol{\phi})/I$
end

Algorithm 8.2: Bayesian linear regression

In Bayesian linear regression we define a normal prior over the parameters $\boldsymbol{\phi}$

$$Pr(\boldsymbol{\phi}) = \text{Norm}_{\boldsymbol{\phi}}[\mathbf{0}, \sigma_p^2\mathbf{I}],$$

which contains one hyperparameter σ_p^2 which determines the prior variance. We compute a distribution over possible parameters $\boldsymbol{\phi}$ and use this to evaluate the mean $\mu_{w^*|\mathbf{x}^*}$ and variance $\sigma^2_{w^*|\mathbf{x}^*}$ of the predictive distribution for new data $\mathbf{x}^*$.

As in the previous algorithm, we add a 1 to the start of every data vector $\mathbf{x}_i \leftarrow [1 \quad \mathbf{x}_i^T]^T$ and then work with the matrix $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2 \ldots \mathbf{x}_I]$ which contains all of the training data examples in its columns and the world vector $\mathbf{w} = [w_1, w_2 \ldots w_I]^T$ which contains the training world states.

The choice of approach depends on whether the number of data examples I is greater or less than the dimensionality D of the data. Depending on which case which situation we are in we move to a situation where we invert the $(D+1) \times (D+1)$ matrix $\mathbf{X}\mathbf{X}^T$ or the $I \times I$ matrix $\mathbf{X}^T\mathbf{X}$.

Algorithm 8.2: Bayesian formulation of linear regression.

Input : $(D+1)\times I$ data matrix $\mathbf{X}$, $I\times 1$ world vector $\mathbf{w}$, Hyperparameter σ_p^2,
Output: Distribution $Pr(w^*|\mathbf{x}^*)$ over world given new data example $\mathbf{x}^*$
begin
  `// If dimensions` D `less than number of data examples` I
  if $D < I$ **then**
    `// Fit variance parameter` σ^2 `with line search`
    $\sigma^2 = \text{argmin}_{\sigma^2}\left[-\log[\text{Norm}_{\mathbf{w}}[\mathbf{0}, \sigma_p^2\mathbf{X}^T\mathbf{X} + \sigma^2\mathbf{I}]]\right]$ [a]
    `// Compute inverse variance of posterior distribution over` $\boldsymbol{\phi}$
    $\mathbf{A}^{-1} = (\mathbf{X}\mathbf{X}^T/\sigma^2 + \mathbf{I}/\sigma_p^2)^{-1}$
  else
    `// Fit variance parameter` σ^2 `with line search`
    $\sigma^2 = \text{argmin}_{\sigma^2}\left[-\log[\text{Norm}_{\mathbf{w}}[\mathbf{0}, \sigma_p^2\mathbf{X}^T\mathbf{X} + \sigma^2\mathbf{I}]]\right]$
    `// Compute inverse variance of posterior distribution over` $\boldsymbol{\phi}$
    $\mathbf{A}^{-1} = \sigma_p^2\mathbf{I} - \sigma_p^2\mathbf{X}\left(\mathbf{X}^T\mathbf{X} + (\sigma^2/\sigma_p^2)\mathbf{I}\right)^{-1}\mathbf{X}^T$
  end
  `// Compute mean of prediction for new example` $\mathbf{x}^*$
  $\mu_{w^*|\mathbf{x}^*} = \mathbf{x}^{*T}\mathbf{A}^{-1}\mathbf{X}\mathbf{w}/\sigma^2$
  `// Compute variance of prediction for new example` $\mathbf{x}^*$
  $\sigma^2_{w^*|\mathbf{x}^*} = \mathbf{x}^{*T}\mathbf{A}^{-1}\mathbf{x}^* + \sigma^2$
end

[a] To compute this cost function when the dimensions $D < I$ we need to compute both the inverse and determinant of the covariance matrix. It is inefficient to implement this directly as the covariance is $I \times I$. To compute the inverse, the covariance should be reformulated using the matrix inversion lemma and the determinant calculated using the matrix determinant lemma.

Algorithm 8.3: Gaussian process regression

To compute a non-linear fit to a set of data, we first transform the data $\mathbf{x}$ by a non-linear function $\mathbf{f}[\bullet]$ to create a new variable $\mathbf{z} = \mathbf{f}[\mathbf{x}_i]$. We then proceed as normal with the Bayesian approach, but using the transformed data.

In practice, we exploit the fact that the Bayesian non-linear regression fitting and prediction algorithms can be described in terms of inner products $\mathbf{z}^T\mathbf{z}$ of the transformed data. We hence directly define a single *kernel function* $\mathrm{k}[\mathbf{x}_i, \mathbf{x}_j]$ as a replacement for the operation $\mathbf{f}[\mathbf{x}_i]^T\mathbf{f}[\mathbf{x}_j]$. For many transformations $\mathbf{f}[\bullet]$ it is more efficient to evaluate the kernel function directly than to transform the variables separately and then compute the dot product. It is further possible to choose kernel functions that correspond to projection to very high or even infinite dimensional spaces without ever having to explicitly compute this transformation.

As usual we add a 1 to the start of every data vector $\mathbf{x}_i \leftarrow [1 \quad \mathbf{x}_i^T]^T$ and then work with the matrix $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2 \ldots \mathbf{x}_I]$ which contains all of the training data examples in its columns and the world vector $\mathbf{w} = [w_1, w_2 \ldots w_I]^T$ which contains the training world states. In this algorithm, we use the notation $\mathbf{K}[\mathbf{A}, \mathbf{B}]$ to denote the $D_A \times D_B$ matrix containing all of the inner products of the D_A columns of A with the D_B columns of B.

Algorithm 8.3: Gaussian process regression.

Input : $(D+1)\times I$ data matrix $\mathbf{X}$, $I\times 1$ world vector $\mathbf{w}$, hyperparameter σ_p^2
Output: Normal distribution $Pr(w^*|\mathbf{x}^*)$ over world given new data example $\mathbf{x}^*$
begin

```
// Fit variance parameter σ^2 with line search
```
$\sigma^2 = \text{argmin}_{\sigma^2}\left[-\log[\text{Norm}_{\mathbf{w}}[\mathbf{0}, \sigma_p^2\mathbf{K}[\mathbf{X},\mathbf{X}] + \sigma^2\mathbf{I}]]\right]$
```
// Compute inverse term
```
$\mathbf{A}^{-1} = \left(\mathbf{K}[\mathbf{X},\mathbf{X}] + (\sigma^2/\sigma_p^2)\mathbf{I}\right)^{-1}$
```
// Compute mean of prediction for new example x*
```
$\mu_{w^*|\mathbf{x}^*} = (\sigma_p^2/\sigma^2)\mathbf{K}[\mathbf{x}^*,\mathbf{X}]\mathbf{w} - (\sigma_p^2/\sigma^2)\mathbf{K}[\mathbf{x}^*,\mathbf{X}]\mathbf{A}^{-1}\mathbf{K}[\mathbf{X},\mathbf{X}]\mathbf{w}$
```
// Compute variance of prediction for new example x*
```
$\sigma^2_{w^*|\mathbf{x}^*} = \sigma_p^2\mathbf{K}[\mathbf{x}^*,\mathbf{x}^*] - \sigma_p^2\mathbf{K}[\mathbf{x}^*,\mathbf{X}]\mathbf{A}^{-1}\mathbf{K}[\mathbf{X},\mathbf{x}^*] + \sigma^2$

end

Algorithm 8.4: Sparse linear regression

In the sparse linear regression model, we replace the normal prior over the parameters with a prior that is a product of t-distributions. This favours solutions where most of the regression parameters are effectively zero. In practice, the t-distribution corresponding to the d^{th} dimension of the data is represented as a marginalization of a joint distribution with a hidden variable h_d.

The algorithm is iterative and alternates between updating the hidden variables in closed form and performing a line search for the noise parameters σ^2. After the system has converged, we prune the model to remove dimensions where the hidden variable was large (>1000 is a reasonable criterion); these dimensions contribute very little to the final prediction.

Algorithm 8.4: Sparse linear regression.

Input : $(D+1)\times I$ data matrix $\mathbf{X}$, $I\times 1$ world vector $\mathbf{w}$, degrees of freedom ν
Output: Distribution $Pr(w^*|\mathbf{x}^*)$ over world given new data example $\mathbf{x}^*$
begin
  `// Initialize variables`
  $\mathbf{H} = \mathbf{diag}[1, 1, \ldots 1]$
  repeat
    `// Maximize marginal likelihood w.r.t. variance parameter`
    $\sigma^2 = \text{argmin}_{\sigma^2}\left[-\log[\text{Norm}_{\mathbf{w}}[\mathbf{0}, \mathbf{X}^T\mathbf{H}^{-1}\mathbf{X} + \sigma^2\mathbf{I}]]\right]$
    `// Maximize marginal likelihood w.r.t. relevance parameters H`
    $\boldsymbol{\Sigma} = \sigma^2(\mathbf{X}\mathbf{X}^T + \mathbf{H})^{-1}$
    $\boldsymbol{\mu} = \boldsymbol{\Sigma}\mathbf{X}\mathbf{w}/\sigma^2$
    `// For each dimension except the first (the constant)`
    for d=2 **to** $D+1$ **do**
      `// Update the diagonal entry of H`
      $h_{dd} = (1 - h_{dd}\Sigma_{dd} + \nu)/(\mu_d^2 + \nu)$
    end
  until *No further improvement*
  `// Remove columns of X and rows and columns of H`
  `// where value` h_{dd} `on the diagonal of H is large`
  $[\mathbf{H}, \mathbf{X}, \mathbf{w}] = \mathbf{prune}[\mathbf{H}, \mathbf{X}, \mathbf{w}]$
  `// Compute variance of posterior over` $\boldsymbol{\Phi}$
  $\mathbf{A}^{-1} = \mathbf{H}^{-1} - \mathbf{H}^{-1}\mathbf{X}\left(\mathbf{X}^T\mathbf{H}^{-1}\mathbf{X} + \sigma^2\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{H}^{-1}$
  `// Compute mean of prediction for new example` $\mathbf{x}^*$
  $\mu_{w^*|\mathbf{x}^*} = \mathbf{x}^{*T}\mathbf{A}^{-1}\mathbf{X}\mathbf{w}/\sigma^2$
  `// Compute variance of prediction for new example` $\mathbf{x}^*$
  $\sigma^2_{w^*|\mathbf{x}^*} = \mathbf{x}^{*T}\mathbf{A}^{-1}\mathbf{x}^* + \sigma^2$
end

Algorithm 8.5: Dual Bayesian linear regression

In dual linear regression, we formulate the weight vector as a sum of the observed data examples $\mathbf{X}$ so that

$$\boldsymbol{\phi} = \mathbf{X}\boldsymbol{\psi}$$

and then solve for the dual parameters $\boldsymbol{\psi}$. To this end we place a normally distributed prior on $\boldsymbol{\psi}$ with a uniform covariance matrix with magnitude σ_p.

Algorithm 8.5: Dual formulation of linear regression.

Input : $(D+1)\times I$ data matrix $\mathbf{X}$, $I\times 1$ world vector $\mathbf{w}$, Hyperparameter σ_p^2,
Output: Distribution $Pr(w^*|\mathbf{x}^*)$ over world given new data example $\mathbf{x}^*$
begin

```
// Fit variance parameter σ^2 with line search
```
$\sigma^2 = \text{argmin}_{\sigma^2}\left[-\log[\text{Norm}_{\mathbf{w}}[0, \sigma_p^2\mathbf{X}^T\mathbf{X}\mathbf{X}^T\mathbf{X} + \sigma^2\mathbf{I}]]\right]$
```
// Compute inverse variance of posterior over Φ
```
$\mathbf{A} = \mathbf{X}^T\mathbf{X}\mathbf{X}^T\mathbf{X}/\sigma^2 + \mathbf{I}/\sigma_p^2$
```
// Compute mean of prediction for new example x*
```
$\mu_{w^*|\mathbf{x}^*} = \mathbf{x}^{*T}\mathbf{X}\mathbf{A}^{-1}\mathbf{X}^T\mathbf{X}\mathbf{w}^*/\sigma^2$
```
// Compute variance of prediction for new example x*
```
$\sigma^2_{w^*|\mathbf{x}^*} = \mathbf{x}^{*T}\mathbf{X}\mathbf{A}^{-1}\mathbf{X}^T\mathbf{x}^* + \sigma^2$

end

Algorithm 8.6: Dual Gaussian process regression

The dual algorithm relies only on inner products of the form $\mathbf{x}^T\mathbf{x}$ and so can be kernelized to form a non-linear regression method. As previously, we use the notation $\mathbf{K}[\mathbf{A}, \mathbf{B}]$ to denote the $D_A \times D_B$ matrix containing all of the inner products of the D_A columns of A with the D_B columns of B.

Algorithm 8.6: Dual Gaussian process regression.

Input : $(D+1)\times I$ data matrix $\mathbf{X}$, $I\times 1$ world vector $\mathbf{w}$, Hyperparameter σ_p^2, Kernel Function $\mathbf{K}[\bullet, \bullet]$
Output: Distribution $Pr(w^*|\mathbf{x}^*)$ over world given new data example $\mathbf{x}^*$
begin

```
// Fit variance parameter σ^2 with line search
```
$\sigma^2 = \text{argmin}_{\sigma^2}\left[-\log[\text{Norm}_{\mathbf{w}}[\mathbf{0}, \sigma_p^2\mathbf{K}[\mathbf{X},\mathbf{X}]\mathbf{K}[\mathbf{X},\mathbf{X}] + \sigma^2\mathbf{I}]]\right]$
```
// Compute inverse term
```
$\mathbf{A} = \mathbf{K}[\mathbf{X},\mathbf{X}]\mathbf{K}[\mathbf{X},\mathbf{X}]/\sigma^2 + \mathbf{I}/\sigma_p^2$
```
// Compute mean of prediction for new example x*
```
$\mu_{w^*|\mathbf{x}^*} = \mathbf{K}[\mathbf{x}^*,\mathbf{X}]\mathbf{A}^{-1}\mathbf{K}[\mathbf{X},\mathbf{X}]\mathbf{w}^*/\sigma^2$
```
// Compute variance of prediction for new example x*
```
$\sigma^2_{w^*|\mathbf{x}^*} = \mathbf{K}[\mathbf{x}^*,\mathbf{X}]\mathbf{A}^{-1}\mathbf{K}[\mathbf{X},\mathbf{x}^*] + \sigma^2$

end

Algorithm 8.7: Relevance vector regression

Relevance vector regression is simply sparse linear regression applied in the dual situation; we encourage the dual parameters $\boldsymbol{\psi}$ to be sparse using a prior that is a product of t-distributions. Since there is one dual parameter for each of the I training examples, we introduce I hidden variables h_i which control the tendency to be zero for each dimension.

The algorithm is iterative and alternates between updating the hidden variables in closed form and performing a line search for the noise parameter σ^2. After the system has converged, we prune the model to remove dimensions where the hidden variable was large (>1000 is a reasonable criterion); these dimensions contribute very little to the final prediction.

Algorithm 8.7: Relevance vector regression.

Input : $(D+1)\times I$ data matrix $\mathbf{X}$, $I\times 1$ world vector $\mathbf{w}$, kernel $\mathbf{K}[\bullet,\bullet]$, degrees of freedom ν
Output: Distribution $Pr(w^*|\mathbf{x}^*)$ over world given new data example $\mathbf{x}^*$
begin
  `// Initialize variables`
  $\mathbf{H} = \mathbf{diag}[1, 1, \ldots 1]$
  repeat
    `// Maximize marginal likelihood wrt variance parameter` σ^2
    $\sigma^2 = \text{argmin}_{\sigma^2}\left[-\log[\text{Norm}_{\mathbf{w}}[\mathbf{0}, \mathbf{K}[\mathbf{X},\mathbf{X}]\mathbf{H}^{-1}\mathbf{K}[\mathbf{X},\mathbf{X}] + \sigma^2\mathbf{I}]]\right]$
    `// Maximize marginal likelihood wrt relevance parameters` $\mathbf{H}$
    $\boldsymbol{\Sigma} = (\mathbf{K}[\mathbf{X},\mathbf{X}]\mathbf{K}[\mathbf{X},\mathbf{X}]/\sigma^2 + \mathbf{H})^{-1}$
    $\boldsymbol{\mu} = \boldsymbol{\Sigma}\mathbf{K}[\mathbf{X},\mathbf{X}]\mathbf{w}/\sigma^2$
    `// For each dual parameter`
    for *i=1* **to** I **do**
      `// Update diagonal entry of` $\mathbf{H}$
      $h_{dd} = (1 - h_{dd}\Sigma_{ii} + \nu)/(\mu_i^2 + \nu)$
    end
  until *No further improvement*
  `// Remove cols of` $\mathbf{X}$`, rows of` $\mathbf{w}$`, rows and cols of` $\mathbf{H}$ `where` h_{dd} `is large`
  $[\mathbf{H}, \mathbf{X}, \mathbf{w}] = \mathbf{prune}[\mathbf{H}, \mathbf{X}, \mathbf{w}]$
  `// Compute inverse term`
  $\mathbf{A} = \mathbf{K}[\mathbf{X},\mathbf{X}]\mathbf{K}[\mathbf{X},\mathbf{X}]/\sigma^2 + \mathbf{H}$
  `// Compute mean of prediction for new example` $\mathbf{x}^*$
  $\mu_{w^*|\mathbf{x}^*} = \mathbf{K}[\mathbf{x}^*,\mathbf{X}]\mathbf{A}^{-1}\mathbf{K}[\mathbf{X},\mathbf{X}]\mathbf{w}/\sigma^2$
  `// Compute variance of prediction for new example` $\mathbf{x}^*$
  $\sigma^2_{w^*|\mathbf{x}^*} = \mathbf{K}[\mathbf{x}^*,\mathbf{X}]\mathbf{A}^{-1}\mathbf{K}[\mathbf{X},\mathbf{x}^*] + \sigma^2$
end

Algorithm 9.1: MAP Logistic regression

The logistic regression model is defined as

$$Pr(w|\mathbf{x},\boldsymbol{\phi}) = \text{Bern}_w\left[\frac{1}{1+\exp[-\boldsymbol{\phi}^T\mathbf{x}]}\right],$$

where as usual, we have attached a 1 to the start of each data example $\mathbf{x}_i$. We now perform a non-linear minimization over the negative log binomial probability with respect to the parameter vector $\boldsymbol{\phi}$:

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\text{argmin}}\left[-\sum_{i=1}^{I}\log\left[\text{Bern}_{w_i}\left[\frac{1}{1+\exp[-\boldsymbol{\phi}^T\mathbf{x}_i]}\right]\right] - \log\left[\text{Norm}_{\boldsymbol{\phi}}[\mathbf{0},\sigma_p^2\mathbf{I}]\right]\right],$$

where we have also added a prior over the parameters $\boldsymbol{\phi}$. The MAP solution is superior to the maximum likelihood approach in that it encourages the function to be smooth even when the classes are completely separable. A typical approach would be to use a second order optimization method such as the Newton method (e.g., using Matlab's fminunc function). The optimization method will need to compute the cost function and it's derivative and Hessian with respect to the parameter $\boldsymbol{\phi}$.

Algorithm 9.1: Cost and derivatives for MAP logistic regression

Input : Binary world state $\{w_i\}_{i=1}^I$, observed data $\{\mathbf{x}_i\}_{i=1}^I$, parameters $\boldsymbol{\phi}$
Output: cost L, gradient $\mathbf{g}$, Hessian $\mathbf{H}$
begin
  `// Initialize cost, gradient, Hessian`
  $L = (D+1)\log[2\pi\sigma_p^2]/2 + \boldsymbol{\phi}^T\boldsymbol{\phi}/(2\sigma_p^2)$
  $\mathbf{g} = \boldsymbol{\phi}/\sigma_p^2$
  $\mathbf{H} = \mathbf{1}/\sigma_p^2$ `// Matrix of ones divided by prior variance`
  `// For each data point`
  for *i=1* **to** I **do**
    `// Compute prediction` y
    $y_i = 1/(1+\exp[-\boldsymbol{\phi}^T\mathbf{x}_i])$
    `// Add term to log likelihood`
    if $w_i == 1$ **then**
      $L = L - \log[y_i]$
    else
      $L = L - \log[1-y_i]$
    end
    `// Add term to gradient`
    $\mathbf{g} = \mathbf{g} + (y_i - w_i)\mathbf{x}_i$
    `// Add term to Hessian`
    $\mathbf{H} = \mathbf{H} + y_i(1-y_i)\mathbf{x}_i\mathbf{x}_i^T$
  end
end

Algorithm 9.2: Bayesian logistic regression

In Bayesian logistic regression, we aim to compute the predictive distribution $Pr(w^*|\mathbf{x}^*)$ over the binary world state w^* for a new data example $\mathbf{x}^*$. This takes the form of a Bernoulli distribution and is hence summarized by the single $\lambda^* = Pr(w^* = 1|\mathbf{x}^*)$.

The method works by first finding the MAP solution (using the cost function in the previous algorithm). It then builds a Laplace approximation based on this result and the Hessian at the MAP solution. Using the mean and variance of the Laplace approximation we can compute a probability distribution over the activation. We then use a further approximation to compute the integral over this distribution.

As usual, we assume that we have added a one to the start of every data vector so that $\mathbf{x}_i \leftarrow [1, \mathbf{x}_i^T]^T$ to model the offset parameter elegantly.

Algorithm 9.2: Bayesian logistic regression

Input : Binary world state $\{w_i\}_{i=1}^I$, observed data $\{\mathbf{x}_i\}_{i=1}^I$, new data $\mathbf{x}^*$
Output: Predictive distribution $Pr(w^*|\mathbf{x}^*)$
begin
  `// Optimization using cost function of algorithm 9.1`
  $\boldsymbol{\phi} = \text{argmin}_{\boldsymbol{\phi}}\left[-\sum_{i=1}^I \log\left[\mathsf{Bern}_{w_i}[1/(1+\exp[-\boldsymbol{\phi}^T\mathbf{x}_i])]\right] - \log\left[\mathsf{Norm}_{\boldsymbol{\phi}}[\mathbf{0}, \sigma_p^2\mathbf{I}]\right]\right]$
  `// Compute Hessian at peak`
  $\mathbf{H} = \mathbf{1}/\sigma_p^2$
  for *i=1* **to** I **do**
    $y_i = 1/(1+\exp[-\boldsymbol{\phi}^T\mathbf{x}_i])$ `// Compute prediction` y
    $\mathbf{H} = \mathbf{H} + y_i(1-y_i)\mathbf{x}_i\mathbf{x}_i^T$ `// Add term to Hessian`
  end
  `// Set mean and variance of Laplace approximation`
  $\boldsymbol{\mu} = \boldsymbol{\phi}$
  $\boldsymbol{\Sigma} = -\mathbf{H}^{-1}$
  `// Compute mean and variance of activation`
  $\mu_a = \boldsymbol{\mu}^T\mathbf{x}^*$
  $\sigma_a^2 = \mathbf{x}^{*T}\boldsymbol{\Sigma}\mathbf{x}^*$
  `// Approximate integral to get Bernoulli parameter`
  $\lambda^* = 1/(1+\exp[-\mu_a/\sqrt{1+\pi\sigma_a^2/8}])$
  `// Compute predictive distribution`
  $Pr(w^*|\mathbf{x}^*) = \mathsf{Bern}_{w^*}[\lambda^*]$
end

Algorithm 9.3: MAP dual logistic regression

The dual logistic regression model is the same as the logistic regression model, but now we represent the parameters $\boldsymbol{\phi}$ as a weighted sum $\boldsymbol{\phi} = \mathbf{X}\boldsymbol{\psi}$ of the original data points, where $\mathbf{X}$ is a matrix containing all of the training data giving the prediction:

$$Pr(w|\boldsymbol{\psi}, \mathbf{x}) = \text{Bern}_w\left[\frac{1}{1+\exp[-\boldsymbol{\psi}^T\mathbf{X}^T\mathbf{x}]}\right].$$

We place a normal prior on the dual parameters $\boldsymbol{\psi}$ and optimize them using the criterion:

$$\hat{\boldsymbol{\psi}} = \underset{\boldsymbol{\psi}}{\text{argmin}}\left[-\sum_{i=1}^{I}\log\left[\text{Bern}_{w_i}\left[\frac{1}{1+\exp[-\boldsymbol{\psi}^T\mathbf{X}^T\mathbf{x}_i]}\right]\right] - \log\left[\text{Norm}_{\boldsymbol{\psi}}[\mathbf{0}, \sigma_p^2\mathbf{I}]\right]\right].$$

A typical approach would be to use a second order optimization method such as the Newton method (e.g., using Matlabs fminunc function). The optimization method will need to compute the cost function and its derivative and Hessian with respect to the parameter $\boldsymbol{\psi}$, and the calculations for these are given in the algorithm below.

Algorithm 9.3: Cost and derivatives for MAP dual logistic regression

Input : Binary world state $\{w_i\}_{i=1}^I$, observed data $\{\mathbf{x}_i\}_{i=1}^I$, parameters $\boldsymbol{\psi}$
Output: cost L, gradient $\mathbf{g}$, Hessian $\mathbf{H}$
begin
  `// Initialize cost, gradient, Hessian`
  $L = -I\log[2\pi\sigma^2]/2 - \boldsymbol{\psi}^T\boldsymbol{\psi}/(2\sigma_p^2)$
  $\mathbf{g} = -\boldsymbol{\psi}/\sigma_p^2$
  $\mathbf{H} = -\mathbf{1}/\sigma_p^2$
  `// Form compound data matrix`
  $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_I]$
  `// For each data point`
  for *i=1* **to** I **do**
    `// Compute prediction` y
    $y_i = 1/(1+\exp[-\boldsymbol{\psi}^T\mathbf{X}^T\mathbf{x}_i])$
    `// Update log likelihood, gradient and Hessian`
    if $w_i == 1$ **then**
      $L = L - \log[y_i]$
    else
      $L = L - \log[1-y_i]$
    end
    $\mathbf{g} = \mathbf{g} + (y_i - w_i)\mathbf{X}^T\mathbf{x}_i$
    $\mathbf{H} = \mathbf{H} + y_i(1-y_i)\mathbf{X}^T\mathbf{x}_i\mathbf{x}_i^T\mathbf{X}$
  end
end

Algorithm 9.4a: Dual Bayesian logistic regression

In dual Bayesian logistic regression, we aim to compute the predictive distribution $Pr(w^*|\mathbf{x}^*)$ over the binary world state w^* for a new data example $\mathbf{x}^*$. This takes the form of a Bernoulli distribution and is hence summarized by the single $\lambda^* = Pr(w^* = 1|\mathbf{x}^*)$.

The method works by first finding the MAP solution to the dual problem(using the cost function in the previous algorithm). It then builds a Laplace approximation based on this result and the Hessian at the MAP solution. Using the mean and variance of the Laplace approximation we can compute a probability distribution over the activation. We then use a further approximation to compute the integral over this distribution.

As usual, we assume that we have added a one to the start of every data vector so that $\mathbf{x}_i \leftarrow [1, \mathbf{x}_i^T]^T$ to model the offset parameter elegantly.

Algorithm 9.4: Dual Bayesian logistic regression

Input : Binary world state $\{w_i\}_{i=1}^I$, observed data $\{\mathbf{x}_i\}_{i=1}^I$, new data $\mathbf{x}^*$
Output: Bernoulli parameter λ^* from $Pr(w^*|\mathbf{x}^*)$ for new data $\mathbf{x}^*$
begin
  `// Optimization using cost function of algorithm 9.3`
  $\boldsymbol{\psi} = \text{argmin}_{\boldsymbol{\psi}} \left[-\sum_{i=1}^I \log \left[\mathsf{Bern}_{w_i}[1/(1+\exp[-\boldsymbol{\psi}^T\mathbf{X}^T\mathbf{x}_i])] \right] - \log \left[\mathsf{Norm}_{\boldsymbol{\psi}}[\mathbf{0}, \sigma_p^2\mathbf{I}] \right] \right]$
  `// Compute Hessian at peak`
  $\mathbf{H} = \mathbf{1}/\sigma_p^2$
  for *i=1* **to** I **do**
    $y_i = 1/(1+\exp[-\boldsymbol{\phi}^T\mathbf{X}^T\mathbf{x}_i])$ `// Compute prediction` y
    $\mathbf{H} = \mathbf{H} + y_i(1-y_i)\mathbf{X}^T\mathbf{x}_i\mathbf{x}_i^T\mathbf{X}$ `// Add term to Hessian`
  end
  `// Set mean and variance of Laplace approximation`
  $\boldsymbol{\mu} = \boldsymbol{\psi}$
  $\boldsymbol{\Sigma} = -\mathbf{H}^{-1}$
  `// Compute mean and variance of activation`
  $\mu_a = \boldsymbol{\mu}^T\mathbf{X}^T\mathbf{x}^*$
  $\sigma_a^2 = \mathbf{x}^{*T}\mathbf{X}\boldsymbol{\Sigma}\mathbf{X}^T\mathbf{x}^*$
  `// Compute approximate prediction`
  $\lambda^* = 1/(1+\exp[-\mu_a/\sqrt{1+\pi\sigma_a^2/8}])$
end

Algorithm 9.4b: Gaussian process classification

Notice that algorithm 9.4a and algorithm 9.3, which it uses, are defined entirely in terms of inner products of the form $\mathbf{x}_i^T\mathbf{x}_j$, which usually occur in matrix multiplications like $\mathbf{X}^T\mathbf{x}^*$. This means they is amenable to kernelization. When we replace all of the inner products in algorithm 9.4a with a kernel function $\mathbf{K}[\bullet,\bullet]$, the resulting algorithm is called Gaussian process classification or kernel logistic regression.

Algorithm 9.5: Relevance vector classification

Relevance vector classification is a version of the kernel logistic regression (Gaussian process classification) that encourages the dual parameters $\boldsymbol{\psi}$ to be sparse using a prior that is a product of t-distributions. Since there is one dual parameter for each of the I training examples, we introduce I hidden variables h_i which control the tendency to be zero for each dimension.

The algorithm is iterative and alternates between updating the hidden variables in closed form and finding the resulting MAP solutions. After the system has converged, we prune the model to remove dimensions where the hidden variable was large (> 1000 is a reasonable criterion); these dimensions contribute very little to the final prediction.

Algorithm 9.5: Relevance vector classification

Input : $(D+1)\times I$ data $\mathbf{X}$, $I\times 1$ binary world vector $\mathbf{w}$, degrees of freedom ν, kernel $K[\bullet,\bullet]$
Output: Bernoulli parameter λ^* from $Pr(w^*|\mathbf{x}^*)$ for new data $\mathbf{x}^*$
begin
  `// Initialize I hidden variables to reasonable values`
  $\mathbf{H} = \mathbf{diag}[1, 1, \ldots 1]$
  repeat
    `// Find MAP solution using kernelized version of algorithm 9.3`
    $\boldsymbol{\psi} =$
    $\text{argmin}_{\boldsymbol{\psi}} \left[-\sum_{i=1}^{I} \log\left[\text{Bern}_{w_i}[1/(1+\exp[-\boldsymbol{\psi}^T\mathbf{K}[\mathbf{X},\mathbf{x}_i]])]\right] - \log\left[\text{Norm}_{\boldsymbol{\psi}}[\mathbf{0},\mathbf{H}^{-1}]\right]\right]$
    `// Compute Hessian S at peak` [a]
    $\mathbf{S} = \mathbf{H}$
    for *i=1* **to** I **do**
      $y_i = 1/(1+\exp[-\boldsymbol{\psi}^T\mathbf{K}[\mathbf{X},\mathbf{x}_i]])$ `// Compute prediction y`
      $\mathbf{S} = \mathbf{S} + y_i(1-y_i)\mathbf{K}[\mathbf{X},\mathbf{x}_i]\mathbf{K}[\mathbf{x}_i,\mathbf{X}]$ `// Add term to Hessian`
    end
    `// Set mean and variance of Laplace approximation`
    $\boldsymbol{\mu} = \boldsymbol{\psi}$
    $\boldsymbol{\Sigma} = -\mathbf{S}^{-1}$
    `// For each data example`
    for *i=1* **to** I **do**
      `// Update the diagonal entry of H`
      $h_{ii} = (1 - h_{ii}\Sigma_{ii} + \nu)/(\mu_i^2 + \nu)$
    end
  until *No further improvement*
  `// Remove rows of` $\boldsymbol{\mu}$`, cols of` $\mathbf{X}$`, rows and cols of` $\boldsymbol{\Sigma}$ `where` h_{dd} `is large`
  $[\boldsymbol{\mu}, \boldsymbol{\Sigma}, \mathbf{X}] = \mathbf{prune}[\boldsymbol{\mu}, \boldsymbol{\Sigma}, \mathbf{X}]$
  `// Compute mean and variance of activation`
  $\mu_a = \boldsymbol{\mu}^T\mathbf{K}[\mathbf{X},\mathbf{x}^*]$
  $\sigma_a^2 = \mathbf{K}[\mathbf{x}^*,\mathbf{X}]\boldsymbol{\Sigma}\mathbf{K}[\mathbf{X},\mathbf{x}^*]$
  `// Compute approximate prediction`
  $\lambda^* = 1/(1+\exp[-\mu_a/\sqrt{1+\pi\sigma_a^2/8}])$
end

[a] Notice that I have used $\mathbf{S}$ to represent the Hessian here, so that it's not confused with the diagonal matrix $\mathbf{H}$ containing the hidden variables.

Algorithm 9.6: Incremental fitting for logistic regression

The incremental fitting approach applies to the non-linear model

$$Pr(w|\boldsymbol{\phi}, \mathbf{x}) = \text{Bern}_w\left[\frac{1}{1+\exp[-\phi_0 - \sum_{k=1}^{K}\phi_k \text{f}[\mathbf{x}_i, \boldsymbol{\xi}_k]]}\right].$$

The method initializes the weights $\{\phi_k\}_{k=1}^K$ to zero and then optimizes them one by one. At the first stage we optimize ϕ_0, ϕ_1 and $\boldsymbol{\xi}_1$. Then we optimize ϕ_0, ϕ_2 and $\boldsymbol{\xi}_2$ and so on.

Algorithm 9.6: Incremental logistic regression

Input : Binary world state $\{w_i\}_{i=1}^I$, observed data $\{\mathbf{x}_i\}_{i=1}^I$
Output: ML parameters $\phi_0, \{\phi_k, \boldsymbol{\xi}_k\}_{k=1}^K$
begin
  `// Initialize parameters`
  $\phi_0 = 0$
  `// Initialize activation for each data point (sum of first k-1 functions)`
  for *i=1* **to** I **do**
    $a_i = 0$
  end
  for *k=1* **to** K **do**
    `// Reset offset parameter` ϕ_0
    for *i=1* **to** I **do**
      $a_i = a_i - \phi_0$
    end
    $[\phi_0, \phi_k, \boldsymbol{\xi}_k] = \text{argmin}_{\phi_0,\phi_k,\boldsymbol{\xi}_k}\left[-\sum_{i=1}^{I}\log\left[\text{Bern}_{w_i}\left[1/(1+\exp[-a_i-\phi_0-\phi_k\text{f}[\mathbf{x}_i,\boldsymbol{\xi}_k]])\right]\right]\right]$
    for *i=1* **to** I **do**
      $a_i = a_i + \phi_0 + \phi_k \text{f}[\mathbf{x}_i, \boldsymbol{\xi}_k]$
    end
  end
end

Obviously, the derivatives for the optimization algorithm depend on the choice of non-linear function. For example, if we use the function $\text{f}[\mathbf{x}_i, \boldsymbol{\xi}_k] = \arctan[\boldsymbol{\xi}_k^T \mathbf{x}_i]$ where we have added a 1 to the start of each data vector $\mathbf{x}_i$, then the first derivatives of the cost function L are:

$$\begin{aligned}
\frac{\partial \text{L}}{\partial \phi_0} &= \sum_{i=1}^{I}(y_i - w_i) \\
\frac{\partial \text{L}}{\partial \phi_k} &= \sum_{i=1}^{I}(y_i - w_i)\text{atan}[\boldsymbol{\xi}_k^T \mathbf{x}_i] \\
\frac{\partial \text{L}}{\partial \xi} &= \sum_{i=1}^{I}(y_i - w_i)\phi_k\left(\frac{1}{1+(\boldsymbol{\xi}_k^T \mathbf{x}_i)^2}\right)\mathbf{x}_i
\end{aligned}$$

where $y_i = 1/(1+\exp[-a_i - \phi_0 - \phi_k \text{f}[\mathbf{x}_i, \boldsymbol{\xi}_k]]$ is the current prediction for the i^{th} data point.

Algorithm 9.7: Logitboost

Logitboost is a special case of non-linear logistic regression, with heaviside step functions:

$$Pr(w|\boldsymbol{\phi}, \mathbf{x}) = \text{Bern}_w \left[\frac{1}{1+\exp[-\phi_0 - \sum_{k=1}^{K} \phi_k \text{heaviside}[f[\mathbf{x}, \xi_{c_k}]]} \right].$$

One interpretation is that we are combining a set of 'weak classifiers' which decide on the class based on whether it is to the left or the right of the step in the step function.

The step functions do not have smooth derivatives, so at the k^{th} stage, the algorithm exhaustively considers a set of possible step functions $\{\text{heaviside}[f[\mathbf{x}, \xi_m]]\}_{m=1}^{M}$, choosing the index $c_k \in \{1, 2, \ldots M\}$ that is best, and simultaneously optimizes the weights ϕ_0 and ϕ_k.

Algorithm 9.7: Logitboost

Input : Binary world state $\{w_i\}_{i=1}^{I}$, observed data $\{\mathbf{x}_i\}_{i=1}^{I}$, functions $\{f_m[\mathbf{x}, \xi_m]\}_{m=1}^{M}$
Output: ML parameters $\phi_0, \{\phi_k\}_{k=1}^{K}$, $\{c_k\} \in \{1 \ldots M\}$
begin
  `// Initialize activations`
  for *i=1* **to** I **do**
    $a_i = 0$
  end
  `// Initialize parameters`
  for *k=1* **to** K **do**
    `// Find best weak classifier by looking at magnitude of gradient`
    $c_k = \max_m[(\sum_{i=1}^{I}(a_i - w_i)\mathsf{f}[x_i, \xi_m])^2]$
    `// Remove effect of offset parameters`
    for *i=1* **to** I **do**
      $a_i = a_i - \phi_0$
    end
    $\phi_0 = 0$
    `// Perform optimization`
    $[\phi_0, \phi_k] = \text{argmin}_{\phi_0,\phi_k} \left[\sum_{i=1}^{I} -\log\left[\mathsf{Bern}_{w_i}\left[1/(1+\exp[-a_i - \phi_0 - \phi_k \mathsf{f}[\mathbf{x}_i, \boldsymbol{\xi}_{c_k}]]) \right] \right] \right]$
    `// Compute new activation`
    for *i=1* **to** I **do**
      $a_i = a_i + \phi_0 + \phi_k \mathsf{f}[\mathbf{x}_i, \boldsymbol{\xi}_{c_k}]$
    end
  end
end

The derivatives for the optimization are given by

$$\begin{aligned} \frac{\partial \text{L}}{\partial \phi_0} &= \sum_{i=1}^{I}(y_i - w_i) \\ \frac{\partial \text{L}}{\partial \phi_k} &= \sum_{i=1}^{I}(y_i - w_i)\mathsf{f}[\mathbf{x}_i, \boldsymbol{\xi}_{c_k}] \end{aligned}$$

where $y_i = 1/(1+\exp[-a_i - \phi_0 - \phi_k \mathsf{f}[\mathbf{x}_i, \boldsymbol{\xi}_{c_k}]]$ is the current prediction for the i^{th} data point.

Algorithm 9.8: Multi-class logistic regression

The multi-class logistic regression model is defined as

$$Pr(w|\boldsymbol{\phi}, \mathbf{x}) = \text{Cat}_w\left[\textbf{softmax}[\boldsymbol{\phi}_1^T\mathbf{x}, \boldsymbol{\phi}_2^T\mathbf{x}, \ldots \boldsymbol{\phi}_N^T\mathbf{x}]\right].$$

where we have prepended a 1 to the start of each data vector $\mathbf{x}$. This is a straightforward optimization problem over the negative log probability with respect to the parameter vector $\boldsymbol{\phi} = [\boldsymbol{\phi}_1; \boldsymbol{\phi}_2; \ldots; \boldsymbol{\phi}_N]$. We need to compute this value, and the derivative and Hessian with respect to the parameters $\{\boldsymbol{\phi}\}_m$.

Algorithm 9.8: Cost function, derivative and Hessian for multi-class logistic regression

Input : World state $\{w_i\}_{i=1}^I$, observed data $\{\mathbf{x}_i\}_{i=1}^I$, parameters $\{\boldsymbol{\phi}\}_{n=1}^N$
Output: cost L, gradient $\mathbf{g}$, Hessian $\mathbf{H}$

begin
  // Initialize cost, gradient, Hessian
  $L = 0$
  for n=*1* **to** N **do**
    $\mathbf{g}_n = \mathbf{0}$ // Part of gradient relating to $\boldsymbol{\phi}_n$
    for m=*1* **to** N **do**
      $\mathbf{H}_{mn} = \mathbf{0}$ // Portion of Hessian relating $\boldsymbol{\phi}_n$ and $\boldsymbol{\phi}_m$
    end
  end
  // For each data point
  for i=*1* **to** I **do**
    // Compute prediction $\mathbf{y}$
    $\mathbf{y}_i = \textbf{softmax}[\boldsymbol{\phi}_1^T\mathbf{x}_i, \boldsymbol{\phi}_2^T\mathbf{x}_i, \ldots \boldsymbol{\phi}_N^T\mathbf{x}_i]$
    // Update log likelihood
    $L = L - \log[y_{i,w_i}]$ // Take w_i^{th} element of $\mathbf{y}_i$
    // Update gradient and Hessian
    for n=*1* **to** N **do**
      $\mathbf{g}_n = \mathbf{g}_n + (y_{in} - \delta[w_i - n])\mathbf{x}_i$
      for m=*1* **to** N **do**
        $\mathbf{H}_{mn} = \mathbf{H}_{mn} + y_{im}(\delta[m-n] - y_{in})\mathbf{x}_i\mathbf{x}_i^T$
      end
    end
  end
  // Assemble final gradient vector
  $\mathbf{g} = [\mathbf{g}_1; \mathbf{g}_2; \ldots \mathbf{g}_N]$
  // Assemble final Hessian
  for n=*1* **to** N **do**
    $\mathbf{H}_n = [\mathbf{H}_{n1}, \mathbf{H}_{n2}, \ldots \mathbf{H}_{nN}]$
  end
  $\mathbf{H} = [\mathbf{H}_1; \mathbf{H}_2; \ldots \mathbf{H}_N]$
end

Algorithm 9.9: Multi-class logistic classification tree

Here, we present a deterministic multi-class classification tree. At the j^{th} branching point, it selects the index $c_j \in \{1, 2, \ldots, M\}$ indicating which of a pre-determined set of classifiers $\{g[\mathbf{x}, \omega_m]\}_{m=1}^{M}$ should be chosen.

Algorithm 9.9: Multiclass classification tree

Input : World state $\{w_i\}_{i=1}^{I}$, data $\{\mathbf{x}_i\}_{i=1}^{I}$, classifiers $\{g[\mathbf{x}, \omega_m]\}_{m=1}^{M}$
Output: Categorical params at leaves $\{\boldsymbol{\lambda}_p\}_{p=1}^{J+1}$, Classifier indices $\{c_j\}_{j=1}^{J}$
begin
  enqueue$[\mathbf{x}_{1\ldots I}, w_{1\ldots I}]$ // Store data and class labels
  // For each node in tree
  for $j = 1$ **to** J **do**
    $[\mathbf{x}_{1\ldots I}, w_{1\ldots I}] =$ dequeue$[\,]$ // Retrieve data and class labels
    for $m = 1$ **to** M **do**
      // Count frequency for k^{th} class in left and right branches
      for $k = 1$ **to** K **do**
        $n_k^{(l)} = \sum_{i=1}^{I} \delta[g[\mathbf{x}_i, \omega_m] - 0]\delta[w_i - k]$
        $n_k^{(r)} = \sum_{i=1}^{I} \delta[g[\mathbf{x}_i, \omega_m] - 1]\delta[w_i - k]$
      end
      // Compute log likelihood
      $l_m = \sum_{k=1}^{K} \log[n_k^{(l)} / \sum_{q=1}^{K} n_q^{(l)}]$ // Contribution from left branch
      $l_m = l_m + \sum_{k=1}^{K} \log[n_k^{(r)} / \sum_{q=1}^{K} n_q^{(r)}]$ // Contribution from right branch
    end
    // Store index of best classifier
    $c_j = \text{argmax}_m [l_m]$
    // Partition into two sets
    $\mathcal{S}_l = \{\}; \mathcal{S}_r = \{\}$
    for i=1 **to** I **do**
      if $g[\mathbf{x}_i, \omega_{c_j}] == 0$ **then**
        $\mathcal{S}_L = \mathcal{S}_l \cup i$
      else
        $\mathcal{S}_R = \mathcal{S}_r \cup i$
      end
    end
    // Add to queue of nodes to process next
    enqueue$[\mathbf{x}_{\mathcal{S}_l}, w_{\mathcal{S}_l}]$
    enqueue$[\mathbf{x}_{\mathcal{S}_r}, w_{\mathcal{S}_r}]$
  end
  // Recover categorical parameters at $J + 1$ leaves
  for $p = 1$ **to** $J + 1$ **do**
    $[\mathbf{x}_{1\ldots I}, w_{1\ldots I}] =$ dequeue$[\,]$
    for k=1 **to** K **do**
      $n_k = \sum_{i=1}^{I} \delta[w_i - k]$ // Frequency of class k at the p^{th} leaf
    end
    $\boldsymbol{\lambda}_p = \mathbf{n} / \sum_{k=1}^{K} n_k$ // ML solution for categorical parameter
  end
end

Algorithm 10.1: Gibbs' sampling from a discrete undirected model

This algorithm generates samples from an undirected model with distribution

$$Pr(x_{1\ldots D}) = \frac{1}{Z}\prod_{c=1}^{C}\phi_c[\mathcal{S}_c],$$

where the c^{th} function $\phi_c[\mathcal{S}_c]$ operates on a subset $\mathcal{S}_c \subset \{x_1, x_2, \ldots, x_D\}$ of the D variables and returns a positive number. For this algorithm, we assume that each variable $\{x_d\}_{d=1}^{D}$ is discrete and takes values $x_d \in \{1, 2, \ldots, K\}$.

In Gibbs' sampling, we choose each variable in turn and update by sampling from its marginal posterior distribution. Since, the variables are discrete, the marginal distribution is a categorical distribution (a histogram), so we can sample from it by partitioning the range 0 to 1 according to the probabilities, drawing a uniform sample between 0 and 1, and seeing which partition it falls into.

Algorithm 10.1: Gibbs' sampling from undirected model

Input : Potential functions $\{\phi_c[\mathcal{S}_c]\}_{c=1}^{C}$
Output: Samples $\{\mathbf{x}_t\}_1^T$
begin
  `// Initialize first sample in chain`
  $\mathbf{x}_0 = \mathbf{x}^{(0)}$
  `// For each time sample`
  for t=*1* **to** T **do**
    $\mathbf{x}_t = \mathbf{x}_{t-1}$
    `// For each dimension`
    for d=*1* **to** D **do**
      `// For each possible value of the` d`th variable`
      for k=*1* **to** K **do**
        `// Set the variable to` k
        $x_{td} = k$
        `// Compute the unnormalized marginal probability`
        $\lambda_k = 1$
        for c *s.t.* $x_d \in \mathcal{S}_c$ **do**
          $\lambda_k = \lambda_k \cdot \phi_c[\mathcal{S}_c]$
        end
      end
      `// Normalize the probabilities`
      $\boldsymbol{\lambda} = \boldsymbol{\lambda}/\sum_{k=1}^{K}\lambda_k$
      `// Draw from categorical distribution`
      $x_{td} = \mathsf{Sample}\left[\mathsf{Cat}_{x_{td}}[\boldsymbol{\lambda}]\right]$
    end
  end
end

It is normal to discard the first few thousand entries so that the initial conditions are forgotten. Then entries are chosen that are spaced apart to avoid correlation between the samples.

Algorithm 10.2: Contrastive divergence for learning undirected models

The contrastive divergence algorithm is used to learn the parameters $\boldsymbol{\theta}$ of an undirected model of the form

$$Pr(x_{1\ldots D}, \boldsymbol{\theta}) = \frac{1}{Z[\boldsymbol{\theta}]} f(\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{Z[\boldsymbol{\theta}]} \prod_{c=1}^{C} \phi_c[\mathcal{S}_c, \boldsymbol{\theta}].$$

where the c^{th} function $\phi_c[\mathcal{S}_c]$ operates on a subset $\mathcal{S}_c \subset \{x_1, x_2, \ldots, x_D\}$ of the D variables and returns a positive number. It is generally not possible to maximize log likelihood either in closed form or via a non-linear optimization algorithm, because we cannot compute the denominator $Z[\boldsymbol{\theta}]$ that normalizes the distribution and which also depends on the parameters.

The contrastive divergence algorithm gets around this problem by computing the approximate gradient by means of generating J samples $\{\mathbf{x}_j^*\}_{j=1}^J$ and then using this approximate gradient to perform gradient descent. The approximate gradient is computed as

$$\frac{\partial L}{\partial \boldsymbol{\theta}} \approx -\frac{I}{J} \sum_{j=1}^{J} \frac{\partial \log[f(\mathbf{x}_j^*, \boldsymbol{\theta})]}{\partial \boldsymbol{\theta}} + \sum_{i=1}^{I} \frac{\partial \log[f(\mathbf{x}_i, \boldsymbol{\theta})]}{\partial \boldsymbol{\theta}}.$$

In the algorithm below, the function gradient$[\mathbf{x}, \boldsymbol{\theta}]$ represents the derivative of the unnormalized log likelihood (i.e. the two terms on the right hand side). We've also made the simplifying assumption that there is one sample $\mathbf{x}_i^*$ for each training example $\mathbf{x}_i$ (i.e., $I = J$). In practice, computing valid samples is a burden, so in this algorithm we generate the i^{th} sample $\mathbf{x}_i^*$ by taking a single Gibbs' sample step from the i^{th} training example.

Algorithm 10.2: Contrastive divergence learning of undirected model

Input : Data $\{\mathbf{x}\}_{i=1}^I$, learning rate α
Output: ML Parameters $\boldsymbol{\theta}$
begin
  `// Initialize parameters`
  $\boldsymbol{\theta} = \boldsymbol{\theta}^{(0)}$
  `// For each time sample`
  repeat
    for i=*1* **to** I **do**
      `// Take a single Gibbs' sample step from the` i`th data point`
      $\mathbf{x}_i^* = \text{Sample}[\mathbf{x}_i, \boldsymbol{\theta}]$
    end
    `// Update parameters`
    `// Function gradient[•,•] returns derivative of log of unnormalized probability`
    $\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha \sum_{i=1}^{I} (\text{gradient}[\mathbf{x}_i, \theta] - \text{gradient}[\mathbf{x}_i^*, \boldsymbol{\theta}])$
  until *No further average change in* $\boldsymbol{\theta}$
end

Algorithm 11.1: Dynamic programming for chain model

This algorithm computes the maximum a posteriori solution for a chain model. The directed chain model has a likelihood and prior that factorize as

$$\begin{aligned} Pr(\mathbf{x}|\mathbf{w}) &= \prod_{n=1}^{N} Pr(\mathbf{x}_n|w_n) \\ Pr(\mathbf{w}) &= \prod_{n=2}^{N} Pr(w_n|w_{n-1}), \end{aligned}$$

respectively. To find the MAP solution, we minimize the negative log posterior:

$$\begin{aligned} \hat{w}_{1\ldots N} &= \underset{w_{1\ldots N}}{\operatorname{argmin}} \left[-\sum_{n=1}^{N} \log\left[Pr(\mathbf{x}_n|w_n)\right] - \sum_{n=2}^{N} \log\left[Pr(w_n|w_{n-1})\right] \right] \\ &= \underset{w_{1\ldots N}}{\operatorname{argmin}} \left[\sum_{n=1}^{N} U_n(w_n) + \sum_{n=2}^{N} P_n(w_n, w_{n-1}) \right]. \end{aligned}$$

This cost function can be optimized using dynamic programming. We pass from variables w_1 to w_N, computing the minimum cost to reach each point, and caching the route. We find the overall minimum at w_N and retrieve the cached route. Here, denote the unary cost $U_n(w_n = k)$ for the n^{th} variable taking value k by $U_{n,k}$, and the pairwise cost $P_n(w_n = k, w_{n-1} = l)$ for the n^{th} variable taking value k and the $(n-1)^{th}$ variable taking value l by $P_{n,k,l}$.

Algorithm 11.1: Dynamic programming in chain

Input : Unary costs $\{U_{n,k}\}_{n=1,k=1}^{N,K}$, Pairwise costs $\{P_{n,k,l}\}_{n=2,k=1,l=1}^{N,K,K}$
Output: Minimum cost path $\{w_n\}_{n=1}^{N}$
begin
  `// Initialize cumulative sums` $S_{1,k}$
  for k=*1* **to** K **do**
    $S_{1,k} = U_{1,k}$
  end
  `// Work forward through chain`
  for n=*2* **to** N **do**
    `// Find minimum cost to get to this node`
    $S_{n,k} = U_{n,k} + \min_l[S_{n-1,l} + P_{n,k,l}]$
    `// Store route by which we got here`
    $R_{n,k} = \text{argmin}_l[S_{n-1,l} + P_{n,k,l}]$
  end
  `// Find node` w_N `with overall minimum cost`
  $w_N = \text{argmin}_k[S_{N,k}]$
  `// Trace back to retrieve route`
  for n=N**to** 2 **do**
    $w_{n-1} = R_{n,w_n}$
  end
end

Algorithm 11.2: Dynamic programming for tree model

This algorithm can be used to compute the MAP solution for a directed or undirected model which has the form of a tree. As such, it generalizes algorithm 11.2 which is specialized for chains. As for the simpler case, the algorithm proceeds by working through the nodes, computing the minimum possible cost to reach this position and caching the route by which we reached this point. At the last node we compute the overall minimum cost and then trace back the route using the cached information.

Here, denote the unary cost $U_n(w_n = k)$ for the n^{th} variable taking value k by $U_{n,k}$. We denote the higher order cost for assigning value K to the n^{th} variable based on its children $\text{ch}[n]$ as $H_{n,k}[\text{ch}[n]]$. This might consist of pairwise, three-wise, or higher costs depending on the number of children in the graph.

Algorithm 11.2: Dynamic programming in tree

Input : Unary costs $\{U_{n,k}\}_{n=1,k=1}^{N,K}$, higher order cost function $\{H_{n,k}[\text{ch}[n]]\}_{n=1,k=1}^{N,K}$
Output: Minimum cost path $\{w_n\}_{n=1}^{N}$
begin
  repeat
    `// Retrieve nodes in an order so children always come before parents`
    $n = \text{GetNextNode}[\,]$
    `// For each possible value of this node`
    for k=*1* **to** K **do**
      `// Compute the minimum cost for reaching here`
      $S_{n,k} = U_{n,k} + \min\left[S_{\text{ch}_{[n]}} + H_{n,k}[\text{ch}[n]]\right]$ [a]
      `// Cache the route for reaching here (store` $|\text{ch}[n]|$ `values)`
      $R_{n,k} = \text{argmin}\left[H_{n,k}[S_{\text{ch}_{[n]}} + \text{ch}[n]]\right]$ [a]
    end
    `// Push node index onto stack`
    $\text{push}[n]$
    `// Until no more parents`
  until $pa[w_n] = \{\}$
  `// Find node` w_N `with overall minimum cost`
  $w_n = \text{min}_k[S_{n,k}]$
  `// Trace back to retrieve route`
  for c=*1* **to** N **do**
    $n = \text{pop}[\,]$
    if $ch[n] \neq \{\}$ **then**
      $w_{\text{ch}_{[n]}} = R_{n,w_n}$
    end
  end
end

[a] This minimization is done over all the values of all of the children variables. With a pairwise term, this would be a single minimization over the single previous variable that fed into this one. With a three-wise term is would be a joint minimization over both children variables etc.

Algorithm 11.3: Forward-backward algorithm

This algorithm computes the marginal posterior distributions $Pr(w_n|\mathbf{x}_{1...N})$ for a chain model. To find the marginal posteriors we perform a forward recursion and a backward recursion and multiply these two quantities together.

Here, we use the term $u_{n,k}$ to represent the likelihood $Pr(x_n|w_n = k)$ of the data x_n at the n^{th} node taking label k and the term $p_{n,k,l}$ to represent the prior term $Pr(w_n = k|w_{n-1} = l)$ when the n^{th} variable takes value k and the $n-1^{th}$ variable takes value l. Note that $u_{n,k}$ and $p_{n,k,l}$ are probabilities, and are not the same as the unary and pairwise costs in the dynamic programming algorithms.

Algorithm 11.3: Forward backward algorithm

Input : Likelihoods $\{l_{nk}\}_{n=1,k=1}^{N,K}$, prior terms $\{P_{n,k,l}\}_{n=2,k=1,l=11}^{N,K,K}$
Output: Marginal probability distributions $\{q_n[w_n]\}_{n=1}^{N}$
begin
  `// Initialize forward vector to likelihood of first variable`
  for k=*1* **to** K **do**
    $\mathbf{f}_{1,k} = u_{1k}$
  end
  `// For each state of each subsequent variable`
  for n=*2* **to** N **do**
    for k=*1* **to** K **do**
      `// Forward recursion`
      $f_{n,k} = u_{n,k} \sum_{l=1}^{K} p_{n-1,k,l} f_{n-1,l}$
    end
  end
  `// Initialize vector for backward pass`
  for k=*1* **to** K **do**
    $b_{N,k} = 1/K$
  end
  `// For each state of each previous variable`
  for n= N **to** 2 **do**
    for k=*1* **to** K **do**
      `// Backward recursion`
      $b_{n-1,k} = \sum_{l=1}^{K} u_{n,l} p_{n,l,k} b_{n,l}$
    end
  end
  `// Compute marginal posteriors`
  for n= 1 **to** N **do**
    for k=*1* **to** K **do**
      `// Take product of forward and backward messages and normalize`
      $q_n[w_n = k] = f_{n,k} b_{n,k} / (\sum_{l=1}^{k} f_{n,l} b_{n,l})$
    end
  end
end

Algorithm 11.4: Sum product belief propagation

The sum product algorithm proceeds in two phases: a forward pass and a backward pass. The forward pass distributes evidence through the graph and the backward pass collates this evidence. Both the distribution and collation of evidence are accomplished by passing messages from node to node in the factor graph. Every edge in the graph is connected to exactly one variable node, and each message is defined over the domain of this variable.

In the description of the algorithm below, we denote the edges by $\{\mathbf{e}_n\}_{n=1}^N$, which joins node e_{n1} to e_{n2}. The edges are processed in such an order that all incoming edges to a function are processed before the outgoing message $\mathbf{m}_n$ is passed. We first discuss the distribute phase.

Algorithm 11.4: Sum product: distribute

Input : Observed data $\{\mathbf{z}_n^*\}_{n\in\mathcal{S}_{obs}\}}$, functions $\{\phi_k[\mathcal{C}_k]\}_{k=1}^K$, edges $\{\mathbf{e}_n\}_{n=1}^N$
Output: Forward messages $\mathbf{m}_n$ on each of the n edges $\mathbf{e}_n$
begin
  repeat
    `// Retrieve edges in any valid order`
    $\mathbf{e}_n = \text{GetNextEdge}[\,]$
    `// Test for type of edge - returns 1 if` e_{n2} `is a function, else 0`
    $t = \text{isEdgeToFunction}[\mathbf{e}_n]$
    if t **then**
      `// If this data was observed`
      if $e_{n1} \in \mathcal{S}_{obs}$ **then**
        $\mathbf{m}_n = \delta[\mathbf{z}^*_{e_{n1}}]$
      else
        `// Find set of edges that are incoming to start of this edge`
        $\mathcal{S} = \{k : e_{n1} == e_{k2}\}$
        `// Take product of messages`
        $\mathbf{m}_n = \prod_{k\in\mathcal{S}} \mathbf{m}_k$
        `// Add edge to stack`
        $\text{push}[\mathbf{e}_n]$
      end
    else
      `// Find set of edges incoming to start of this edge`
      $\mathcal{S} = \{k : e_{n1} == e_{k2}\}$
      `// Find all variables connected to this function`
      $\mathcal{V} = e_{\mathcal{S}1} \cup e_{n2}$
      `// Take product of messages`
      $\mathbf{m}_n = \sum_{\mathbf{y}\in\mathcal{S}} \phi_n[\mathbf{y}_\mathcal{V}] \prod_{k\in\mathcal{S}} \mathbf{m}_k$
      `// Add edge to stack`
      $\text{push}[n]$
    end
  until $pa[e_n] = \{\}$
end

This algorithm continues overleaf...

Algorithm 11.4b: Collate and compute marginal distributions

After the distribute stage is complete (one message has been passed along each edge in the graph) we commence t the second pass through the variables. This happens in the opposite order to the first stage (accomplished by popping edges off the stack). Now, we collate the evidence and compute the normalized distributions at each node.

Algorithm 11.4b: Sum product: collate and compute marginal distributions

Input : Observed data $\{\mathbf{z}_n^*\}_{n\in\mathcal{S}_{obs}}$, functions $\{\phi_k[\mathcal{C}_k]\}_{k=1}^K$, edges $\{\mathbf{e}_n\}_{n=1}^N$
Output: Marginal probability distributions $\{q_n[y_n]\}_{n=1}^N$
begin
  `// Collate evidence`
  repeat
    `// Retrieve edges in opposite order`
    $n = \text{pop}[\,]$
    `// Test for type of edge - returns 1 if` e_{n2} `is a function, else 0`
    $t = \text{isEdgeToFunction}[\mathbf{e}_n]$
    `// Test for type of edge`
    if t **then**
      `// Find set of edges incoming to function node`
      $\mathcal{S} = \{k : e_{n2} == e_{k1}\}$
      `// Find all variables connected to this function`
      $\mathcal{V} = e_{\mathcal{S}2} \cup e_{n1}$
      `// Take product of messages`
      $\mathbf{b}_n = \sum_{\mathbf{y}\in mathcalS} \phi_n[\mathbf{y}_\mathcal{S}] \prod_{k\in\mathcal{S}} \mathbf{b}_k$
    else
      `// Find set of edges that are incoming to data node`
      $\mathcal{S} = \{k : e_{n2} == e_{k1}\}$
      `// Take product of messages`
      $\mathbf{b}_n = \prod_{k\in\mathcal{S}} \mathbf{b}_k$
    end
  until *stack empty*
  `// Compute distributions at nodes`
  for k=*1* **to** K **do**
    `// Find set of edges that are incoming to data node`
    $\mathcal{S}_1 = \{n : e_{n2} == k\}$
    $\mathcal{S}_2 = \{n : e_{n1} == k\}$
    $\mathbf{q}_k = \prod_{n\in\mathcal{S}_1} \mathbf{m}_n \prod_{n\in\mathcal{S}_2} \mathbf{b}_n$
  end
end

Algorithm 12.1: Binary graph cuts

This algorithm assumes that we have N variables each of which takes a binary value. Their connections are indicated by a series of flags $\{E_{mn}\}_{n,m=1}^{N,N}$ which are set to one if the variables are connected (and have an associated pairwise term) or zero otherwise. This algorithm sets up the graph but doesn't find the min-cut solution. Consult a standard algorithms text for details of how to do this.

Algorithm 12.1: Binary graph cuts

Input : Unary costs $\{U_n(k)\}_{n,k=1}^{N,K}$, pairwise costs $\{P_{n,m}(k,l)\}_{n,m,k,l=1}^{N,N,K,K}$,flags $\{e_{mn,}\}_{n=1,m=1}^{N,N}$
Output: Label assignments w_n
begin
  `// Initialize graph to empty`
  $\mathcal{G} = \{\}$
  for n=*1* **to** N **do**
    `// Create edges from source and to sink and set capacity to zero`
    $\mathcal{G} = \mathcal{G} \cup \{s, n\}; c_{sn} = 0$
    $\mathcal{G} = \mathcal{G} \cup \{n, t\}; c_{nt} = 0$
    `// If edge between` m `and` n `is desired`
    if $e_{m,n} = 1$ **then**
      $\mathcal{G} = \mathcal{G} \cup \{m, n\}; c_{nm} = 0$
      $\mathcal{G} = \mathcal{G} \cup \{n, m\}; c_{mn} = 0$
    end
  end
  `// Add costs to edges`
  for n=*1* **to** N **do**
    $c_{sn} = c_{sn} + U_n(0)$ $c_{nt} = c_{nt} + U_n(1)$ **for** m=*1* **to** $n-1$ **do**
      if $e_{m,n} = 1$ **then**
        $c_{nm} = c_{nm} + P_{mn}(1,0) - P_{mn}(1,1) - P_{mn}(0,0)$
        $c_{mn} = c_{mn} + P_{mn}(1,0)$
        $c_{sm} = c_{sm} + P_{mn}(0,0)$
        $c_{nt} = c_{nt} + P_{mn}(1,1)$
      end
    end
  end
  $\mathbf{C}$ = Reparameterize[$\mathbf{C}$] `// Ensures all capacities are positive (see overleaf)`
  $\mathcal{G}$ = ComputeMinCut[$\mathcal{G}, \mathbf{C}$] `// Augmenting paths or similar`
  `// Read off world state values based on new (cut) graph`
  for n=*1* **to** N **do**
    if $\{s, n\} \in \mathcal{G}$ **then**
      $w_n = 1$
    else
      $w_n = 0$
    end
  end
end

Algorithm 12.2: Reparameterization for graph cuts

The previous algorithm relies on a max-flow / min cut algorithm such as augmenting paths or push-relabel. For these algorithms to converge, it is critical that all of the capacities are non-negative. The process of making them non-negative is called re-parameterization. It is only possible in certain special cases, and here the problem is known as submodular. Cost functions in vision tend to encourage smoothing and are submodular.

Algorithm 12.2: Reparameterization for binary graph cut

Input : Edge flags $\{e_{mn}\}_{m,n=1}^{N,N}$, capacities $\{c_{mn}\} : e_{m,n} = 1$
Output: Modified graph with non-negative capacities
begin
  `// For each node pair`
  for n=*1* **to** N **do**
    for m=*1* **to** $n-1$ **do**
      `// If an edge between the nodes exist`
      if $e_{m,n} = 1$ **then**
        `// Test if submodular and return error code if not`
        if $c_{nm} < 0$ && $c_{mn} < -c_{nm}$ **then**
          return[-1]
        end
        if $c_{mn} < 0$ && $c_{nm} < -c_{mn}$ **then**
          return[-1]
        end
        `// Handle links between source and sink`
        if $c_{nm} < 0$ **then**
          $\beta = c_{nm}$
        end
        if $c_{mn} < 0$ **then**
          $\beta = -c_{mn}$
        end
        $c_{nm} = c_{nm} - \beta$
        $c_{mn} = c_{mn} + \beta$
        $c_{sm} = c_{sm} + \beta$
        $c_{mt} = c_{mt} + \beta$
      end
    end
    `// Handle links between source and sink`
    $\alpha = \min[c_{sn}, c_{nt}]$
    $c_{sn} = c_{sn} - \alpha$
    $c_{nt} = c_{nt} - \alpha$
  end
end

Algorithm 12.3: Multi-label graph cuts

This algorithm assumes that we have N variables each of which takes one of K values. Their connections are indicated by a set of flags $\{e_{mn}\}_{n,m=1}^{N,N}$ which are set to one if the variables are connected (and have an associated pairwise term) or zero otherwise. We construct a graph that has $N \cdot (K+1)$ nodes where the first $K+1$ nodes pertain to the first variable and so on.

Algorithm 12.3: Multilabel graph cuts

Input : Unary costs $\{U_n(k)\}_{n,k=1}^{N,K}$, pairwise costs $\{P_{n,m}(k,l)\}_{n,m,k,l=1}^{N,N,K,K}$, flags $\{e_{mn,}\}_{n=1,m=1}^{N,N}$
Output: Label assignments w_n
begin
  $\mathcal{G} = \{\}$ `// Initialize graph to empty`
  for n=*1* **to** N **do**
    `// Create edges from source and to sink and set costs`
    $\mathcal{G} = \mathcal{G} \cup \{s, (n-1)(K+1)+1\}; c_{s,(n-1)(K+1)+1} = \infty$
    $\mathcal{G} = \mathcal{G} \cup \{n(K+1), t\}; c_{,n(K+1)t} = \infty$
    `// Create edges within columns and set costs`
    for k=*1* **to** K **do**
      $\mathcal{G} = \mathcal{G} \cup \{(n-1)(K+1)+k, (n-1)(K+1)+k+1\}$
      $c_{(n-1)(K+1)+k,(n-1)(K+1)+k+1} = U_{(n-1)(K+1)+k,k}$
      $\mathcal{G} = \mathcal{G} \cup \{(n-1)(K+1)+k+1, (n-1)(K+1)+k\}$
      $c_{(n-1)(K+1)+k+1,(n-1)(K+1)+k} = \infty$
    end
    `// Create edges between columns and set costs`
    for m=*1* **to** $n-1$ **do**
      if $e_{m,n} = 1$ **then**
        for k=*1* **to** K **do**
          for L=*2* **to** $K+1$ **do**
            $\mathcal{G} = \mathcal{G} \cup \{(n-1)(K+1)+k, (m-1)(K+1)+l\}$
            $c_{(n-1)(K+1)+k,(m-1)(K+1)+l} =$
            $P_{n,m}(k,l-1) + P_{n,m}(k-1,l) - P_{n,m}(k,l) - P_{n,m}(k-1,l-1)$
          end
        end
      end
    end
  end
  $\mathbf{C} =$ Reparameterize$[\mathbf{C}]$ `// Ensures all capacities are positive (see book)`
  $\mathcal{G} =$ ComputeMinCut$[\mathcal{G}, \mathbf{C}]$ `// Augmenting paths or similar`
  tcpRead off values **for** n=*1* **to** N **do**
    $w_n = 1$
    for k=*1* **to** K **do**
      if $\{(n-1)(K+1)+k, (n-1)(K+1)+k\} \in \mathcal{G}]$ **then**
        $w_n = w_n + 1$
      end
    end
  end
end

Algorithm 12.4: Alpha-expansion algorithm

The *alpha-expansion* algorithm works by breaking the solution down into a series of binary problems, each of which can be solved exactly. At each iteration, we choose one of the K label values α, and for each pixel, we consider either retaining the current label, or switching it to α. The name alpha-expansion derives from the fact that the space occupied by label α in the solution expands at each iteration. The process is iterated until no choice of α causes any change. Each expansion move is guaranteed to lower the overall objective function, although the final result is not guaranteed to be the global minimum.

Algorithm 12.4: Alpha expansion algorithm (main loop)

Input : Unary costs $\{U_n(k)\}_{n,k=1}^{N,K}$, pairwise costs $\{P_{n,m}(k,l)\}_{n,m,k,l=1}^{N,N,K,K}$, flags $\{e_{mn},\}_{n=1,m=1}^{N,N}$
Output: Label assignments $\{w_n\}_{n=1}^{N}$
begin
  `// Initialize labels in some way - perhaps to minimize unary costs`
  $\mathbf{w} = \mathbf{w}^0$
  `// Compute log likelihood`
  $L = \sum_{n=1}^{N} U_n(w_n) + \sum_{n=1}^{N}\sum_{m=1}^{M} e_{mn} P_{n,m}(w_n, w_m)$
  repeat
    `// Store initial log likelihood`
    $L_0 = L$
    `// For each label in turn`
    for *k=1* **to** K **do**
      `// Try to expand this label (see overleaf)`
      $\mathbf{w} = \text{AlphaExpand}[\mathbf{w}, k]$
    end
    `// Compute new log likelihood`
    $L = \sum_{n=1}^{N} U_n(w_n) + \sum_{n=1}^{N}\sum_{m=1}^{M} E_{mn} P_{n,m}(w_n, w_m)$
  until $L = L_0$
end

In the alpha-expansion graph construction, there is one vertex associated with each pixel. Each of these vertices is connected to the source (representing keeping the original label or $\overline{\alpha}$) and the sink (representing the label α). To separate source from sink, we must cut one of these two edges at each pixel. The choice of edge will determine whether we keep the original label or set it to α. Accordingly, we associate the unary costs for each edge being set to α or its original label with the two links from each pixel. If the pixel already has label α, then we set the cost of being set to $\overline{\alpha}$ to ∞.

The remaining structure of the graph is dynamic: it changes at each iteration depending on the choice of α and the current labels. There are four possible relationships between adjacent pixels:

- They can both already be set to alpha.
- One can be set to alpha and the other to another value β.
- Both can be set to the same other value β .
- They can be set to two other values β and γ.

Algorithm 12.4b: Alpha expansion (expand)

Algorithm 12.4b: Alpha expansion (expand)

Input : Costs $\{U_n(k)\}_{n,k=1}^{N,K}$, $\{P_{n,m}(k,l)\}_{n,m,k,l=1}^{N,N,K,K}$, expansion label k, states $\{w_n\}_{n=1}^{N}$
Output: New label assignments $\{w_n\}_{n=1}^{N}$
begin
  $\mathcal{G} = \{\}$ // Initialize graph to empty
  $z = N$ // Counter for new nodes added to graph
  for n=*1* **to** N **do**
    $\mathcal{G} = \mathcal{G} \cup \{s, n\}; c_{sn} = U_n(k)$ // Connect pixel nodes to source and set cost
    if $w_n = k$ **then**
      $\mathcal{G} = \mathcal{G} \cup \{n, t\}; c_{nt} = \infty$ // Connect pixel nodes to sink and set cost
    else
      $\mathcal{G} = \mathcal{G} \cup \{n, t\}; c_{nt} = U_n(w_n)$ // Connect pixel nodes to sink and set cost
    end
    for m=*1* **to** n **do**
      if $e_{m,n} == 1$ **then**
        if $(w_n == k \;||\; w_m == k)$ **then**
          if $w_n != k$ **then**
            $\mathcal{G} = \mathcal{G} \cup \{n, m\}; c_{nm} = P_{n,m}(w_m, w_n)$ // Case 2a
          end
          if $w_m != k$ **then**
            $\mathcal{G} = \mathcal{G} \cup \{m, n\}; c_{mn} = P_{n,m}(w_n, w_m)$ // Case 2b
          end
        else
          if $w_n == w_m$ **then**
            $\mathcal{G} = \mathcal{G} \cup \{n, m\}; c_{nm} = P_{n,m}(k, w_n)$ // Case 3
            $\mathcal{G} = \mathcal{G} \cup \{m, n\}; c_{mn} = P_{n,m}(w_n, k)$
          else
            $z = z + 1$ // Increment new node counter
            $\mathcal{G} = \mathcal{G} \cup \{n, z\}; c_{nz} = P_{n,m}(k, w_n); c_{zn} = \infty$ // Case 4
            $\mathcal{G} = \mathcal{G} \cup \{m, z\}; c_{mz} = P_{n,m}(w_m, k) c_{zm} = \infty$
            $\mathcal{G} = \mathcal{G} \cup \{z, t\}; c_{zt} = P_{n,m}(w_m, w_n)$
          end
        end
      end
    end
  end
  $\mathbf{C} = \text{Reparameterize}[\mathbf{C}]$ // Ensures all capacities are positive
  $\mathcal{G} = \text{ComputeMinCut}[\mathcal{G}, \mathbf{C}]$ // Augmenting paths or similar
  // Read off values
  for n=*1* **to** N **do**
    if $\{n, t\} \in \mathcal{G}$ **then**
      $w_n = k$
    end
  end
end

Algorithm 13.1: Principal components analysis

The goal of PCA is to approximate a set of multivariate data $\{\mathbf{x}_i\}_{i=1}^I$ with a second set of variables of reduced size $\{\mathbf{h}_i\}_{i=1}^I$, so that

$$\mathbf{x}_i \approx \boldsymbol{\mu} + \boldsymbol{\Phi}\mathbf{h}_i,$$

where $\boldsymbol{\Phi}$ is a rectangular matrix where the columns are unit length and orthogonal to one another so that $\boldsymbol{\Phi}^T\boldsymbol{\Phi} = \mathbf{I}$.

This formulation assumes that the number of original data dimensions D is higher than the number of training examples I and so works by taking the singular value decomposition of the $I \times I$ matrix $\mathbf{X}^T\mathbf{X}$ to compute the dual principal components $\boldsymbol{\Psi}$ before recovering the original principal components $\boldsymbol{\Phi}$.

Algorithm 13.1: Principal components analysis (dual)

Input : Training data $\{\mathbf{x}_i\}_{i=1}^I$, number of components K
Output: Mean $\boldsymbol{\mu}$, PCA basis functions $\boldsymbol{\Phi}$, low dimensional data $\{\mathbf{h}_i\}_{i=1}^I$
begin
  // Estimate mean
  $\boldsymbol{\mu} = \sum_{i=1}^I \mathbf{x}_i / I$
  // Form mean zero data matrix
  $\mathbf{X} = [\mathbf{x}_1 - \boldsymbol{\mu}, \mathbf{x}_2 - \boldsymbol{\mu}, \ldots \mathbf{x}_I - \boldsymbol{\mu}]$
  // Do spectral decomposition and compute dual components
  $[\boldsymbol{\Psi}, \mathbf{L}, \boldsymbol{\Psi}] = \mathbf{svd}[\mathbf{X}^T\mathbf{X}]$
  // Compute principal components
  $\boldsymbol{\Phi} = \mathbf{X}\boldsymbol{\Psi}\mathbf{L}^{-1/2}$
  // Retain only the first K columns
  $\boldsymbol{\Phi} = [\boldsymbol{\phi}_1, \boldsymbol{\phi}_2, \ldots, \boldsymbol{\phi}_K]$
  // Convert data to low dimensional representation
  for i=*1* **to** I **do**
    $\mathbf{h}_i = \boldsymbol{\Phi}^T(\mathbf{x}_i - \boldsymbol{\mu})$
  end
  // Reconstruct data
  for i = *1* **to** I **do**
    $\tilde{\mathbf{x}}_i = \boldsymbol{\mu} + \boldsymbol{\Phi}\mathbf{h}_i$
  end
end

Algorithm 13.2: k-means algorithm

The goal of the k-means algorithm is to partition a set of data $\{\mathbf{x}_i\}_{i=1}^{I}$ into K clusters. It can be thought of as approximating each data point with the associated cluster mean $\boldsymbol{\mu}_k$, so that

$$\mathbf{x}_i \approx \boldsymbol{\mu}_{h_i},$$

where $h_i \in \{1, 2, \ldots K\}$ is a discrete variable that indicates which cluster the ith point belongs to. The algorithm works by alternately (i) assigning data points to the nearest cluster center and (ii)

Algorithm 13.2: K-means algorithm

Input : Data $\{\mathbf{x}_i\}_{i=1}^{I}$, number of clusters K, data dimension D
Output: Cluster means $\{\boldsymbol{\mu}_k\}_{k=1}^{K}$, cluster assignment indices, $\{h_i\}_{i=1}^{I}$
begin
  `// Initialize cluster means (one of many heuristics)`
  $\boldsymbol{\mu} = \sum_{i=1}^{I} \mathbf{x}_i / I$ `// Compute overall mean`
  $\boldsymbol{\Sigma} = \sum_{i=1}^{I} (\mathbf{x}_i - \boldsymbol{\mu})^T (\mathbf{x}_i - \boldsymbol{\mu}) / I$ `// Compute overall covariance`
  for k=*1* **to** K **do**
    $\boldsymbol{\mu}_k = \boldsymbol{\mu} + \boldsymbol{\Sigma}^{1/2}\mathbf{randn}[D, 1]$ `// Randomly draw from normal model`
  end
  `// Main loop`
  repeat
    `// Compute distance from data points to cluster means`
    for i=*1* **to** I **do**
      for k=*1* **to** K **do**
        $\mathbf{d}_{ik} = (\mathbf{x}_i - \boldsymbol{\mu}_k)^T (\mathbf{x}_i - \boldsymbol{\mu}_k)$
      end
      `// Update cluster assignments based on closest cluster`
      $h_i = \text{argmin}_k [\mathbf{d}_{ik}]$
    end
    `// Update cluster means from data that was assigned to this cluster`
    for k=*1* **to** K **do**
      $\boldsymbol{\mu}_k = (\sum_{i=1}^{I} \delta[h_i - k]\mathbf{x}_i) / (\sum_{i=1}^{I} \delta[h_i - k])$
    end
  until *No further change in* $\{\boldsymbol{\mu}_k\}_{k=1}^{K}$
end

Algorithm 14.1: ML learning of camera extrinsic parameters

Given a known object, with I distinct three-dimensional points $\{\mathbf{w}_i\}_{i=1}^{I}$ points, their corresponding projections in the image $\{\mathbf{x}_i\}_{i=1}^{I}$ and known camera parameters $\mathbf{\Lambda}$, estimate the geometric relationship between the camera and the object determined by the rotation $\boldsymbol{\Omega}$ and the translation $\boldsymbol{\tau}$.

The solution to this problem is to minimize:

$$\hat{\boldsymbol{\Omega}}, \hat{\boldsymbol{\tau}} = \underset{\boldsymbol{\Omega},\boldsymbol{\tau}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} \left(\mathbf{x}_i - \mathbf{pinhole}[\mathbf{w}_i, \mathbf{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}]\right)^T \left(\mathbf{x}_i - \mathbf{pinhole}[\mathbf{w}_i, \mathbf{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}]\right) \right]$$

where $\mathbf{pinhole}[\mathbf{w}_i, \mathbf{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}]$ represents the action of the pinhole camera (equation 14.8 from the book. The bulk of this algorithm consists of finding a good initial starting point for this minimization. This optimization should be carried out while enforcing the constraint that $\boldsymbol{\Omega}$ remains a valid rotation matrix.

Algorithm 14.1: ML learning of extrinsic parameters

Input : Intrinsic matrix $\mathbf{\Lambda}$, pairs of points $\{\mathbf{x}_i, \mathbf{w}_i\}_{i=1}^{I}$
Output: Extrinsic parameters: rotation $\boldsymbol{\Omega}$ and translation $\boldsymbol{\tau}$
begin
  for i=1 **to** I **do**
    `// Convert to normalized camera coordinates`
    $\mathbf{x}'_i = \mathbf{\Lambda}^{-1}[x_i; y_i; 1]$
    `// Compute linear constraints`
    $a_{1i} = [u_i, v_i, w_i, 1, 0, 0, 0, 0, -u_i x'_i, -v_i x'_i, -w_i x'_i, -x'_i]$
    $a_{2i} = [0, 0, 0, 0, u_i, v_i, w_i, 1, -u_i y'_i, -v_i y'_i, -w_i y'_i, -y'_i]$
  end
  `// Stack linear constraints`
  $\mathbf{A} = [a_{11}; a_{21}; a_{12}; a_{22}; \ldots a_{1I}; a_{2I}]$
  `// Solve with SVD`
  $[\mathbf{U}, \mathbf{L}, \mathbf{V}] = \mathsf{svd}[\mathbf{A}]$
  $\mathbf{b} = \mathbf{v}_{12}$ `// extract last column of` $\mathbf{V}$
  `// Extract estimates up to unknown scale`
  $\tilde{\boldsymbol{\Omega}} = [b_1, b_2, b_3; b_5, b_6, b_7; b_9; b_{10}, b_{11}]$
  $\tilde{\boldsymbol{\tau}} = [b_4; b_8; b_{12}]$
  `// Find closest rotation using Procrustes method`
  $[\mathbf{U}, \mathbf{L}, \mathbf{V}] = \mathsf{svd}[\tilde{\boldsymbol{\Omega}}]$
  $\boldsymbol{\Omega} = \mathbf{U}\mathbf{V}^T$
  `// Rescale translation`
  $\boldsymbol{\tau} = \tilde{\boldsymbol{\tau}} \sum_{i=1}^{3} \sum_{j=1}^{3} (\boldsymbol{\Omega}_{ij} / \tilde{\boldsymbol{\Omega}}_{ij})/9$
  `// Use these parameters as initial conditions in non-linear optimization`
  $[\boldsymbol{\Omega}, \boldsymbol{\tau}] = \operatorname{argmin}_{\boldsymbol{\Omega},\boldsymbol{\tau}} \left[\sum_{i=1}^{I} \left(\mathbf{x}_i - \mathbf{pinhole}[\mathbf{w}_i, \mathbf{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}]\right)^T \left(\mathbf{x}_i - \mathbf{pinhole}[\mathbf{w}_i, \mathbf{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}]\right) \right]$
end

Algorithm 14.2: ML learning of intrinsic parameters (camera calibration)

Given a known object, with I distinct 3D points $\{\mathbf{w}_i\}_{i=1}^{I}$ points and their corresponding projections in the image $\{\mathbf{x}_i\}_{i=1}^{I}$, establish the camera parameters $\boldsymbol{\Lambda}$. In order to do this we need also to estimate the extrinsic parameters. We use the following criterion

$$\hat{\boldsymbol{\Lambda}} = \underset{\boldsymbol{\Lambda}}{\operatorname{argmin}} \left[\min_{\boldsymbol{\Omega},\boldsymbol{\tau}} \left[\sum_{i=1}^{I} (\mathbf{x}_i - \mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}])^T (\mathbf{x}_i - \mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}]) \right] \right]$$

where $\mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}]$ represents the action of the pinhole camera (equation 14.8 from the book).

This algorithm consists of an alternating approach in which the extrinsic parameters are found using the previous algorithm and then the intrinsic parameters are found in closed form. Finally, these estimates should form the starting point for a non-linear optimization process over all of the unknown parameters.

Algorithm 14.2: ML learning of intrinsic parameters

Input : World points $\{\mathbf{w}_i\}_{i=1}^{I}$, image points $\{\mathbf{x}_i\}_{i=1}^{I}$, initial $\boldsymbol{\Lambda}$
Output: Intrinsic parameters $\boldsymbol{\Lambda}$
begin
  `// Main loop for alternating optimization`
  for t=*1* **to** T **do**
    `// Compute extrinsic parameters`
    $[\boldsymbol{\Omega}, \boldsymbol{\tau}] = \mathbf{calcExtrinsic}[\boldsymbol{\Lambda}, \{\mathbf{w}_i, \mathbf{x}_i\}_{i=1}^{I}]$
    `// Compute intrinsic parameters`
    for i=*1* **to** I **do**
      `// Compute matrix` $\mathbf{A}_i$
      $a_i = (\boldsymbol{\omega}_{1\bullet}\mathbf{w}_i + \boldsymbol{\tau}_x)/(\boldsymbol{\omega}_{3\bullet}\mathbf{w}_i + \boldsymbol{\tau}_z)$ `//` $\boldsymbol{\omega}_{k\bullet}$ `is` k^{th} `row of` $\boldsymbol{\Omega}$
      $b_i = (\boldsymbol{\omega}_{2\bullet}\mathbf{w}_i + \boldsymbol{\tau}_y)/(\boldsymbol{\omega}_{3\bullet}\mathbf{w}_i + \boldsymbol{\tau}_z)$
      $\mathbf{A}_i = [a_i, b_i, 1, 0, 0; 0, 0, 0, b_i, 1]$
    end
    `// Concatenate matrices and data points`
    $\mathbf{x} = [\mathbf{x}_1; \mathbf{x}_2; \ldots \mathbf{x}_I]$
    $\mathbf{A} = [\mathbf{A}_1; \mathbf{A}_2; \ldots \mathbf{A}_I]$
    `// Compute parameters`
    $\boldsymbol{\theta} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{x}$
    $\boldsymbol{\Lambda} = [\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3; 0, \boldsymbol{\theta}_4, \boldsymbol{\theta}_5; 0, 0, 1]$
  end
  `// Refine parameters with non-linear optimization`
  $\boldsymbol{\Lambda} = \operatorname{argmin}_{\boldsymbol{\Lambda}} \left[\min_{\boldsymbol{\Omega},\boldsymbol{\tau}} \left[\sum_{i=1}^{I} (\mathbf{x}_i - \mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}])^T (\mathbf{x}_i - \mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}]) \right] \right]$
end

Algorithm 14.3: Inferring 3D world points (reconstruction)

Given J calibrated cameras in known positions (i.e. cameras with known $\boldsymbol{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}$), viewing the same three-dimensional point $\mathbf{w}$ and knowing the corresponding projections in the images $\{\mathbf{x}_j\}_{j=1}^J$, establish the position of the point in the world.

As for the previous algorithms the final solution depends on a non-linear minimization of the reprojection error between $\mathbf{w}$ and the observed data $\mathbf{x}_j$,

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\text{argmin}} \left[\sum_{j=1}^{J} \left(\mathbf{x}_j - \mathbf{pinhole}[\mathbf{w}, \boldsymbol{\Lambda}_j, \boldsymbol{\Omega}_j, \boldsymbol{\tau}_j]\right)^T \left(\mathbf{x}_j - \mathbf{pinhole}[\mathbf{w}, \boldsymbol{\Lambda}_j, \boldsymbol{\Omega}_j, \boldsymbol{\tau}_j]\right) \right]$$

The algorithm below finds a good approximate initial conditions for this minimization using a closed-form least-squares solution.

Algorithm 14.3: Inferring 3D world position

Input : Image points $\{\mathbf{x}_j\}_{j=1}^J$, camera parameters $\{\boldsymbol{\Lambda}_j, \boldsymbol{\Omega}_j, \boldsymbol{\tau}_j\}_{j=1}^J$
Output: 3D world point $\mathbf{w}$
begin
  for j=*1* **to** J **do**
    `// Convert to normalized camera coordinates`
    $\mathbf{x}'_j = \boldsymbol{\Lambda}_j^{-1}[x_j, y_j, 1]^T$
    `// Compute linear constraints`
    $a_{1j} = [\omega_{31j}x'_j - \omega_{11j}, \omega_{32j}x'_j - \omega_{12j}, \omega_{33j}x'_j - \omega_{13j}]$
    $a_{2j} = [\omega_{31j}y'_j - \omega_{21j}, \omega_{32j}y'_j - \omega_{22j}, \omega_{33j}y'_j - \omega_{23j}]$
    $b_j = [\tau_{xj} - \tau_{zj}x'_j; \tau_{yj} - \tau_{zj}y'_j]$
  end
  `// Stack linear constraints`
  $\mathbf{A} = [a_{11}; a_{21}; a_{12}; a_{22}; \ldots a_{1J}; a_{2J}]$
  $\mathbf{b} = [b_1; b_2; \ldots b_J]$
  `// LS solution for parameters`
  $\mathbf{w} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$
  `// Refine parameters with non-linear optimization`
  $\hat{\mathbf{w}} = \text{argmin}_{\mathbf{w}} \left[\sum_{j=1}^{J} \left(\mathbf{x}_j - \mathbf{pinhole}[\mathbf{w}, \boldsymbol{\Lambda}_j, \boldsymbol{\Omega}_j, \boldsymbol{\tau}_j]\right)^T \left(\mathbf{x}_j - \mathbf{pinhole}[\mathbf{w}, \boldsymbol{\Lambda}_j, \boldsymbol{\Omega}_j, \boldsymbol{\tau}_j]\right) \right]$
end

Algorithm 15.1: ML learning of Euclidean transformation

The Euclidean transformation model maps one set of 2D points $\{\mathbf{w}_i\}_{i=1}^I$ to another set $\{\mathbf{x}_i\}_{i=1}^I$ with a rotation $\boldsymbol{\Omega}$ and a translation $\boldsymbol{\tau}$. To recove these parameters we use the criterion

$$\hat{\boldsymbol{\Omega}}, \hat{\boldsymbol{\tau}} = \underset{\boldsymbol{\Omega},\boldsymbol{\tau}}{\operatorname{argmin}} \left[-\sum_{i=1}^{I} \log \left[\text{Norm}_{\mathbf{x}_i} \left[\boldsymbol{\Omega}\mathbf{w}_i + \boldsymbol{\tau}, \sigma^2 \mathbf{I} \right] \right] \right]$$

where $\boldsymbol{\Omega}$ is constrained to be a rotation matrix so that $\boldsymbol{\Omega}^T\boldsymbol{\Omega} = \mathbf{I}$ and $\det[\boldsymbol{\Omega}] = 1$.

Algorithm 15.1: Maximum likelihood learning of Euclidean transformation

Input : Training data pairs $\{\mathbf{x}_i, \mathbf{w}_i\}_{i=1}^I$
Output: Rotation $\boldsymbol{\Omega}$, translation $\boldsymbol{\tau}$, variance, σ^2
begin
  `// Compute mean of two data sets`
  $\boldsymbol{\mu}_w = \sum_{i=1}^I \mathbf{w}_i / I$
  $\boldsymbol{\mu}_x = \sum_{i=1}^I \mathbf{x}_i / I$
  `// Concatenate data into matrix form`
  $\mathbf{W} = [\mathbf{w}_1 - \boldsymbol{\mu}_w, \mathbf{w}_2 - \boldsymbol{\mu}_w, \ldots, \mathbf{w}_I - \boldsymbol{\mu}_w]$
  $\mathbf{X} = [\mathbf{x}_1 - \boldsymbol{\mu}_x, \mathbf{x}_2 - \boldsymbol{\mu}_x, \ldots, \mathbf{x}_I - \boldsymbol{\mu}_x]$
  `// Solve for rotation`
  $[\mathbf{U}, \mathbf{L}, \mathbf{V}] = \mathbf{svd}[\mathbf{W}\mathbf{X}^T]$
  $\boldsymbol{\Omega} = \mathbf{V}\mathbf{U}^T$
  `// Solve for translation`
  $\boldsymbol{\tau} = \sum_{i=1}^I (\mathbf{x}_i - \boldsymbol{\Omega}\mathbf{w}_i)/I$
end

Algorithm 15.2: ML learning of similarity transformation

The similarity transformation model maps one set of 2D points $\{\mathbf{w}_i\}_{i=1}^I$ to another set $\{\mathbf{x}_i\}_{i=1}^I$ with a rotation $\boldsymbol{\Omega}$, a translation $\boldsymbol{\tau}$ and a scaling factor ρ. To recover these parameters we use the criterion:

$$\hat{\boldsymbol{\Omega}}, \hat{\boldsymbol{\tau}}, \hat{\rho} = \underset{\boldsymbol{\Omega}, \boldsymbol{\tau}, \rho}{\operatorname{argmin}} \left[-\sum_{i=1}^{I} \log \left[\text{Norm}_{\mathbf{x}_i} \left[\rho \boldsymbol{\Omega} \mathbf{w}_i + \boldsymbol{\tau}, \sigma^2 \mathbf{I} \right] \right] \right]$$

where $\boldsymbol{\Omega}$ is constrained to be a rotation matrix so that $\boldsymbol{\Omega}^T\boldsymbol{\Omega} = \mathbf{I}$ and $\det[\boldsymbol{\Omega}] = 1$.

Algorithm 15.2: Maximum likelihood learning of similarity transformation

Input : Training data pairs $\{\mathbf{x}_i, \mathbf{w}_i\}_{i=1}^I$
Output: Rotation $\boldsymbol{\Omega}$, translation $\boldsymbol{\tau}$, scale ρ, variance σ^2
begin
  `// Compute mean of two data sets`
  $\boldsymbol{\mu}_w = \sum_{i=1}^I \mathbf{w}_i / I$
  $\boldsymbol{\mu}_x = \sum_{i=1}^I \mathbf{x}_i / I$
  `// Concatenate data into matrix form`
  $\mathbf{W} = [\mathbf{w}_1 - \boldsymbol{\mu}_w, \mathbf{w}_2 - \boldsymbol{\mu}_w, \ldots, \mathbf{w}_I - \boldsymbol{\mu}_w]$
  $\mathbf{X} = [\mathbf{x}_1 - \boldsymbol{\mu}_x, \mathbf{x}_2 - \boldsymbol{\mu}_x, \ldots, \mathbf{x}_I - \boldsymbol{\mu}_x]$
  `// Solve for rotation`
  $[\mathbf{U}, \mathbf{L}, \mathbf{V}] = \mathbf{svd}[\mathbf{W}\mathbf{X}^T]$
  $\boldsymbol{\Omega} = \mathbf{V}\mathbf{U}^T$
  `// Solve for scaling`
  $\rho = (\sum_{i=1}^I (\mathbf{x}_i - \boldsymbol{\mu}_x)^T \boldsymbol{\Omega} (\mathbf{w}_i - \boldsymbol{\mu}_w)) / (\sum_{i=1}^I (\mathbf{w}_i - \boldsymbol{\mu}_w)^T (\mathbf{w} - \boldsymbol{\mu}_w))$
  `// Solve for translation`
  $\boldsymbol{\tau} = \sum_{i=1}^I (\mathbf{x}_i - \rho \boldsymbol{\Omega} \mathbf{w}_i) / I$
end

Algorithm 15.3: ML learning of affine transformation

The affine transformation model maps one set of 2D points $\{\mathbf{w}_i\}_{i=1}^{I}$ to another set $\{\mathbf{x}_i\}_{i=1}^{I}$ with a linear transformation $\boldsymbol{\Phi}$ and an offset $\boldsymbol{\tau}$. To recover these parameters we use the criterion

$$\hat{\boldsymbol{\Phi}}, \hat{\boldsymbol{\tau}} = \underset{\boldsymbol{\Phi},\boldsymbol{\tau}}{\operatorname{argmin}} \left[-\sum_{i=1}^{I} \log \left[\text{Norm}_{\mathbf{x}_i} \left[\boldsymbol{\Phi}\mathbf{w}_i + \boldsymbol{\tau}, \sigma^2\mathbf{I} \right] \right] \right].$$

Algorithm 15.3: Maximum likelihood learning of affine transformation

Input : Training data pairs $\{\mathbf{x}_i, \mathbf{w}_i\}_{i=1}^{I}$
Output: Linear transformation $\boldsymbol{\Phi}$, offset $\boldsymbol{\tau}$, variance σ^2
begin
  `// Compute intermediate` 2×6 `matrices` $\mathbf{A}_i$
  for i=*1* **to** I **do**
    $\mathbf{A}_i = [\mathbf{w}_i^T, 1, \mathbf{0}^T; \mathbf{0}^T, \mathbf{w}_i^T, 1]$
  end
  `// Concatenate matrices` $\mathbf{A}_i$ `into` $2I\times6$ `matrix` $\mathbf{A}$
  $\mathbf{A} = [\mathbf{A}_1; \mathbf{A}_2; \ldots \mathbf{A}_I]$
  `// Concatenate output points into` $2I \times 1$ `vector` $\mathbf{c}$
  $\mathbf{c} = [\mathbf{x}_1; \mathbf{x}_2; \ldots \mathbf{x}_I]$
  `// Solve for linear transformation`
  $\boldsymbol{\phi} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{c}$
  `// Extract parameters`
  $\boldsymbol{\Phi} = [\phi_1, \phi_2; \phi_4, \phi_5]$
  $\boldsymbol{\tau} = [\phi_3; \phi_6]$
  `// Solve for variance`
  $\sigma^2 = \sum_{i=1}^{I}(\mathbf{x}_i - \boldsymbol{\phi}\mathbf{w}_i - \boldsymbol{\tau})^T(\mathbf{x}_i - \boldsymbol{\phi}\mathbf{w}_i - \boldsymbol{\tau})/2I$
end

Algorithm 15.4: ML learning of projective transformation (homography)

The projective transformation model maps one set of 2D points $\{\mathbf{w}_i\}_{i=1}^I$ to another set $\{\mathbf{x}_i\}_{i=1}^I\}$ with a non-linear transformation with 3×3 parameter matrix $\boldsymbol{\Phi}$. To recover this matrix we use the criterion

$$\hat{\boldsymbol{\Phi}} = \underset{\boldsymbol{\Phi}}{\text{argmin}}\left[-\sum_{i=1}^{I}\log\left[\text{Norm}_{\mathbf{x}_i}\left[\mathbf{proj}[\mathbf{w}_i,\boldsymbol{\Phi}],\sigma^2\mathbf{I}\right]\right]\right].$$

where the function $\mathbf{proj}[\mathbf{w}_i,\boldsymbol{\Phi}]$ applies the homography to point $\mathbf{w}_i$ and is defined as

$$\mathbf{proj}[\mathbf{w}_i,\boldsymbol{\Phi}] = \begin{bmatrix}\frac{\phi_{11}u+\phi_{12}v+\phi_{13}}{\phi_{31}u+\phi_{32}v+\phi_{33}} & \frac{\phi_{21}u+\phi_{22}v+\phi_{23}}{\phi_{31}u+\phi_{32}v+\phi_{33}}\end{bmatrix}^T.$$

Unlike the previous three transformations, it is not possible to minimize this criterion in closed form. The best that we can do is to get an approximate solution and use this to start a non-linear minimization process.

Algorithm 15.4: Maximum likelihood learning of projective transformation

Input : Training data pairs $\{\mathbf{x}_i,\mathbf{w}_i\}_{i=1}^I$
Output: Parameter matrix $\boldsymbol{\Phi}$,, variance σ^2
begin
  `// Convert data to homogeneous representation`
  for *i=1* **to** I **do**
    $\tilde{\mathbf{x}}_i = [\mathbf{x}_i; 1]$
  end
  `// Compute intermediate` 2×9 `matrices` $\mathbf{A}_i$
  for *i=1* **to** I **do**
    $\mathbf{A}_i = [\mathbf{0}, \tilde{\mathbf{w}}_i; -\tilde{\mathbf{w}}_i, \mathbf{0}; y_i\tilde{\mathbf{w}}_i, -x_i\tilde{\mathbf{w}}_i]^T$
  end
  `// Concatenate matrices` $\mathbf{A}_i$ `into` $2I\times9$ `matrix` $\mathbf{A}$
  $\mathbf{A} = [\mathbf{A}_1; \mathbf{A}_2; \ldots \mathbf{A}_I]$
  `// Solve for approximate parameters`
  $[\mathbf{U},\mathbf{L},\mathbf{V}] = \mathbf{svd}[\mathbf{A}]$
  $\boldsymbol{\Phi}_0 = [v_{19}, v_{29}, v_{39}; v_{49}, v_{59}, v_{69}; v_{79}, v_{89}, v_{99}]$
  `// Refine parameters with non-linear optimization`
  $\hat{\boldsymbol{\Phi}} = \text{argmin}_{\boldsymbol{\Phi}}\left[-\sum_{i=1}^{I}\log\left[\text{Norm}_{\mathbf{x}_i}\left[\mathbf{proj}[\mathbf{w}_i,\boldsymbol{\Phi}],\sigma^2\mathbf{I}\right]\right]\right]$.
end

Algorithm 15.5: ML Inference for transformation models

Consider a transformation model maps one set of 2D points $\{\mathbf{w}_i\}_{i=1}^I$ to another set $\{\mathbf{x}_i\}_{i=1}^I$ so that

$$Pr(\mathbf{x}_i|\mathbf{w}_i, \boldsymbol{\Phi}) = \text{Norm}_{\mathbf{x}_i}\left[\mathbf{trans}[\mathbf{w}_i, \boldsymbol{\Phi}], \sigma^2\mathbf{I}\right].$$

In inference we wish are given a new data point $\mathbf{x} = [x, y]$ and wish to compute the most likely point $\mathbf{w} = [u, v]$ that was responsible for it. To make progress, we consider the transformation model $\mathbf{trans}[\mathbf{w}_i, \boldsymbol{\Phi}]$ in homogeneous form

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \phi_{11} & \phi_{12} & \phi_{13} \\ \phi_{21} & \phi_{22} & \phi_{23} \\ \phi_{31} & \phi_{32} & \phi_{33} \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix},$$

or $\tilde{\mathbf{x}} = \boldsymbol{\Phi}\tilde{\mathbf{w}}$. The Euclidean, similarity, affine and projective transformations can all be expressed as a 3×3 matrix of this kind.

Algorithm 15.5: Maximum likelihood inference for transformation models

Input : Transformation parameters $\boldsymbol{\Phi}$, new point $\mathbf{x}$
Output: point $\mathbf{w}$
begin
  `// Convert data to homogeneous representation`
  $\tilde{\mathbf{x}} = [\mathbf{x}; 1]$
  `// Apply inverse transform`
  $\mathbf{a} = \boldsymbol{\Phi}^{-1}\tilde{\mathbf{x}}$
  `// Convert back to Cartesian coordinates`
  $\mathbf{w} = [a_1/a_3, a_2/a_3]$
end

Algorithm 15.6: Learning extrinsic parameters (planar scene)

Consider a calibrated camera with known parameters $\boldsymbol{\Lambda}$ viewing a planar. We are given a set of 2D positions on the plane $\{\mathbf{w}_{i=1}^{I}\}$ (measured in real world units like cm) and their corresponding 2D pixel positions $\mathbf{x}_{i-1}^{I}$. The goal of this algorithm is to learn the 3D rotation $\boldsymbol{\Omega}$ and translation $\boldsymbol{\tau}$ that maps a point in the frame of reference of the plane $\mathbf{w} = [u, v, w]^T$ ($w = 0$ on the plane) into the frame of reference of the camera.

This goal is accomplished by minimizing the following criterion:

$$\hat{\boldsymbol{\Omega}}, \hat{\boldsymbol{\tau}} = \underset{\boldsymbol{\Omega},\boldsymbol{\tau}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} \left(\mathbf{x}_i - \mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}]\right)^T \left(\mathbf{x}_i - \mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}]\right) \right]$$

This optimization should be carried out while enforcing the constraint that $\boldsymbol{\Omega}$ remains a valid rotation matrix. The bulk of this algorithm consists of computing a good initialization point for this minimization procedure.

Algorithm 15.6: ML learning of extrinsic parameters (planar scene)

Input : Intrinsic matrix $\boldsymbol{\Lambda}$, pairs of points $\{\mathbf{x}_i, \mathbf{w}_i\}_{i=1}^{I}$
Output: Extrinsic parameters: rotation $\boldsymbol{\Omega}$ and translation $\boldsymbol{\tau}$
begin
  `// Compute homography between pairs of points`
  $\boldsymbol{\Phi} = \mathbf{LearnHomography}[\{\mathbf{x}_i\}_{i=1}^{I}, \{\mathbf{w}_i\}_{i=1}^{I}]$
  `// Eliminate effect of intrinsic parameters`
  $\boldsymbol{\Phi} = \boldsymbol{\Lambda}^{-1}\boldsymbol{\Phi}$
  `// Compute SVD of first two columns of` $\boldsymbol{\Phi}$
  $[\mathbf{ULV}] = \mathbf{svd}[\boldsymbol{\phi}_1, \boldsymbol{\phi}_2]$
  `// Estimate first two columns of rotation matrix`
  $[\boldsymbol{\omega}_1, \boldsymbol{\omega}_2] = [\mathbf{u}_1, \mathbf{u}_2] * \mathbf{V}^T$
  `// Estimate third column by taking cross product`
  $\boldsymbol{\omega}_3 = \boldsymbol{\omega}_1 \times \boldsymbol{\omega}_2$
  $\boldsymbol{\Omega} = [\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \boldsymbol{\omega}_3]$
  `// Check that determinant is not minus 1`
  if $|\boldsymbol{\Omega}| < 0$ **then**
    $\boldsymbol{\Omega} = [\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, -\boldsymbol{\omega}_3]$
  end
  `// Compute scaling factor for translation vector`
  $\lambda = (\sum_{i=1}^{3}\sum_{j=1}^{2} \boldsymbol{\omega}_{ij}/\boldsymbol{\phi}_{ij})/6$
  `// Compute translation`
  $\boldsymbol{\tau} = \lambda\boldsymbol{\phi}_3$
  `// Refine parameters with non-linear optimization`
  $\hat{\boldsymbol{\Omega}}, \hat{\boldsymbol{\tau}} = \operatorname{argmin}_{\boldsymbol{\Omega},\boldsymbol{\tau}} \left[\sum_{i=1}^{I} \left(\mathbf{x}_i - \mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}]\right)^T \left(\mathbf{x}_i - \mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}, \boldsymbol{\tau}]\right)\right]$
end

Algorithm 15.7: Learning intrinsic parameters (planar scene)

This is also known as camera calibration from a plane. The camera is presented with J views of a plane with unknown pose $\{\boldsymbol{\Omega}_j, \boldsymbol{\tau}_j\}$. For each image we know I points $\{\mathbf{w}_i\}_{i=1}^{I}$ where $\mathbf{w}_i = [u_i, v_i, 0]$ and we know their imaged positions $\{\mathbf{x}_{ij}\}_{i=1,j=1}^{I,J}$ in each of the J scenes. The goal is to compute the intrinsic matrix $\boldsymbol{\Lambda}$. To this end, we use the criterion:

$$\hat{\boldsymbol{\Lambda}} = \underset{\boldsymbol{\Lambda}}{\text{argmin}} \left[\sum_{j=1}^{J} \min_{\boldsymbol{\Omega}_j, \boldsymbol{\tau}_j} \left[\sum_{i=1}^{I} (\mathbf{x}_{ij} - \mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}_j, \boldsymbol{\tau}_j])^T (\mathbf{x}_{ij} - \mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}_j, \boldsymbol{\tau}_j]) \right] \right]$$

where again, the minimization must be carried out while ensuring that $\boldsymbol{\Omega}$ is a valid rotation matrix. The strategy is to alternately estimate the extrinsic parameters using the previous algorithm and compute the intrinsic parameters in closed form. After several iterations we use the resulting solution as initial conditions for a non-linear optimization procedure.

Algorithm 15.7: ML learning of intrinsic parameters (planar scene)

Input : World points $\{\mathbf{w}_i\}_{i=1}^{I}$, image points $\{\mathbf{x}_{ij}\}_{i=1,j=1}^{I,J}$, initial $\boldsymbol{\Lambda}$
Output: Intrinsic parameters $\boldsymbol{\Lambda}$
begin
  `// Main loop for alternating optimization`
  for k=*1* **to** K **do**
    `// Compute extrinsic parameters for each image`
    for j=*1* **to** J **do**
      $[\boldsymbol{\Omega}_j, \boldsymbol{\tau}_j] = \mathbf{calcExtrinsic}[\boldsymbol{\Lambda}, \{\mathbf{w}_i, \mathbf{x}_{ij}\}_{i=1}^{I}]$
    end
    `// Compute intrinsic parameters`
    for i=*1* **to** I **do**
      for j=*1* **to** J **do**
        `// Compute matrix` $\mathbf{A}_{ij}$
        $a_{ij} = (\boldsymbol{\omega}_{1\bullet j}^T \mathbf{w}_i + \boldsymbol{\tau}_{xj}) / (\boldsymbol{\omega}_{3\bullet j}^T \mathbf{w}_i + \boldsymbol{\tau}_{zj})$ `//` $\boldsymbol{\omega}_{k\bullet j}$ `is` k^{th} `row of` $\boldsymbol{\Omega}_j$
        $b_{ij} = (\boldsymbol{\omega}_{2\bullet j}^T \mathbf{w}_i + \boldsymbol{\tau}_{yj}) / (\boldsymbol{\omega}_{3\bullet j}^T \mathbf{w}_i + \boldsymbol{\tau}_{zj})$ `//` τ_{zj} `is` z `component of` $\boldsymbol{\tau}_j$
        $\mathbf{A}_{ij} = [a_{ij}, b_{ij}, 1, 0, 0; 0, 0, 0, b_{ij}, 1]$
      end
    end
    `// Concatenate matrices and data points`
    $\mathbf{x} = [\mathbf{x}_{11}; \mathbf{x}_{12}; \ldots \mathbf{x}_{IJ}]$
    $\mathbf{A} = [\mathbf{A}_{11}; \mathbf{A}_{12}; \ldots \mathbf{A}_{IJ}]$
    `// Compute parameters`
    $\boldsymbol{\theta} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{x}$
    $\boldsymbol{\Lambda} = [\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3; 0, \boldsymbol{\theta}_4, \boldsymbol{\theta}_5; 0, 0, 1]$
  end
  `// Refine parameters with non-linear optimization`
  $\hat{\boldsymbol{\Lambda}} =$
  $\text{argmin}_{\boldsymbol{\Lambda}} \left[\sum_j \min_{\boldsymbol{\Omega}_j, \boldsymbol{\tau}_j} \left[\sum_i (\mathbf{x}_{ij} - \mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}_j, \boldsymbol{\tau}_j])^T (\mathbf{x}_{ij} - \mathbf{pinhole}[\mathbf{w}_i, \boldsymbol{\Lambda}, \boldsymbol{\Omega}_j, \boldsymbol{\tau}_j]) \right] \right]$
end

Algorithm 15.8: Robust learning of projective transformation with RANSAC

The goal of this algorithm is to fit a homography that maps one set of 2D points $\{\mathbf{w}_i\}_{i=1}^{I}$ to another set $\{\mathbf{x}_i\}_{i=1}^{I}\}$, in the case where some of the point matches are known to be wrong (outliers). The algorithm also returns the true matches and the outliers.

The algorithm uses the RANSAC procedure - it repeatedly computes the homography based on a minimal subset of matches. Since there are 8 unknowns in the 3×3 matrix that defines the homography, and each match provides two linear constrains (due to the $x-$ and $y-$coordinates), we need a minimum of four matches to compute the homography. The RANSAC procedure chooses these four matches randomly, computes the homography, and then looks for the amount of agreement in the rest of the dataset. After many iterations of this procedure, we recompute the homography based on the randomly chosen matches with the best agreement and the points that agreed with it (the inliers).

Algorithm 15.8: Robust ML learning of homography

Input : Point pairs $\{\mathbf{x}_i, \mathbf{w}_i\}_{i=1}^{I}$, number of RANSAC steps N, threshold τ
Output: Homography $\mathbf{\Phi}$, inlier indices $\mathcal{I}$
begin
  `// Initialize best inlier set to empty`
  $\mathcal{B} = \{\}$
  for n=*1* **to** N **do**
    `// Draw 4 different random integers between 1 and` I
    $\mathcal{R} = \text{RandomSubset}[\{1 \ldots I\}, 4]$
    `// Compute homography (algorithm 15.4)`
    $\mathbf{\Phi}_n = \text{LearnHomography}[\{\mathbf{x}_i\}_{i\in\mathcal{R}}, \{\mathbf{w}_i\}_{i\in\mathcal{R}}]$
    `// Initialize set of inliers to empty`
    $\mathcal{S}_n = \{\}$
    for i=*1* **to** I **do**
      `// Compute squared distance`
      $d = (\mathbf{x}_i - \text{proj}[\mathbf{w}_i, \mathbf{\Phi}_n])^T(\mathbf{x}_i - \text{proj}[\mathbf{w}_i, \mathbf{\Phi}_n])$
      `// If small enough then add to inliers`
      if $d < \tau^2$ **then**
        $\mathcal{S}_n = \mathcal{S}_n \cap \{i\}$
      end
    end
    `// If best outliers so far then store`
    if $|\mathcal{S}_n| > |\mathcal{B}|$ **then**
      $\mathcal{B} = \mathcal{S}_n$
    end
  end
  `// Compute homography from all outliers`
  $\mathbf{\Phi} = \text{LearnHomography}[\{\mathbf{x}_i\}_{i\in\mathcal{B}}, \{\mathbf{w}_i\}_{i\in\mathcal{B}}]$
end

Algorithm 15.9: Sequential RANSAC for fitting homographies

Sequential RANSAC fits K homographies to disjoint subsets of point pairs $\{\mathbf{w}_i, \mathbf{x}_i\}_{i=1}^{I}$. This procedure is greedy – the algorithm fits the first homography, then removes the inliers from this set from the point pairs and tries to fit a second homography to the remaining points. In principle, this algorithm can find a set of matching planes between two images. However, in practice, it often makes mistakes. It does not exploit information about the spatial coherence of matches and it cannot recover from mistakes in the greedy matching procedure.

Algorithm 15.9: Robust sequential learning of homographies

Input : Points $\{\mathbf{x}_i, \mathbf{w}_i\}_{i=1}^{I}$, RANSAC steps N, inlier threshold τ, number of homographies K
Output: K homographies $\mathbf{\Phi}_k$, and associated inlier indices $\mathcal{I}_k$
begin
  `// Initialize set of indices of remaining point pairs`
  $\mathcal{S} = \{1 \ldots I\}$ **for** k=*1* **to** K **do**
    `// Compute homography using RANSAC (algorithm 51)`
    $[\mathbf{\Phi}_k, \mathcal{I}_k] = \text{LearnHomographyRobust}[\{\mathbf{x}_i\}_{i\in\mathcal{S}}, \{\mathbf{w}_i\}_{i\in\mathcal{S}}, N, \tau]$
    `// Remove inliers from remaining points`
    $\mathcal{S} = \mathcal{S} \backslash \mathcal{I}_k$
    `// Check that there are enough remaining points`
    if $|\mathcal{S}| < 4$ **then**
      break
    end
  end
end

Algorithm 15.10: PEaRL for fitting homographies

The propose, expand and re-learn (PEaRL) attempts to make up for the deficiencies of sequential RANSAC for fitting homographies. It first proposes a large number of possible homographies relating point pairs $\{\mathbf{w}_i, \mathbf{x}_i\}_{i=1}^{I}$. These then compete for the point pairs to be assigned to them and they are re-learnt based on these assignments. The algorithm has a spatial component that encourages nearby points to belong to the same model, and it iterative rather than greedy and so can recover from errors.

Algorithm 15.10: PEaRL learning of homographies

Input : Point pairs $\{\mathbf{x}_i, \mathbf{w}_i\}_{i=1}^{I}$, number of initial models M, inlier threshold τ, mininum number of inliers l, number of iterations J, neighborhood system $\{\mathcal{N}_i\}_{i=1}^{I}$, pairwise cost P

Output: Set of homographies $\mathbf{\Phi}_k$, and associated inlier indices $\mathcal{I}_k$

begin

```
// Propose Step:  generate M hypotheses
m = 1                                                    // hypothesis number
repeat
    // Draw 4 different random integers between 1 and I
    R = RandomSubset[{1...I}, 4]
    // Compute homography (algorithm 47)
    Φ_m = LearnHomography[{x_i}_{i∈R}, {w_i}_{i∈R}]
    I_m = {}                                        // Initialize inlier set to empty
    for i=1 to I do
        d_im = (x_i − proj[w_i, Φ_n])^T (x_i − proj[w_i, Φ_n])
        if d_im < τ^2 then                          // if distance small, add to inliers
            I_n = I_n ∩ {i}
        end
    end
    if |I_m| ≥ l then                               // If enough inliers, get next hypothesis
        m = m + 1
    end
until m < M
for j=1 to J do
    // Expand Step:  returns I × 1 label vector l
    l = AlphaExpand[D, P, {N_i}_{i=1}^I]
    // Re-Learn Step:  re-estimate homographies with support
    for m=1 to M do
        I_m = find[L == m]                          // Extract points with label L
        // If enough support then re-learn, update distances
        if |I_m| ≥ 4 then
            Φ_m = LearnHomography[{x_i}_{i∈I_m}, {w_i}_{i∈I_m}]
            for i=1 to I do
                d_im = (x_i − proj[w_i, Φ_n])^T (x_i − proj[w_i, Φ_n])
            end
        end
    end
end
```

end

Algorithm 16.1: Camera geometry from point matches

This algorithm finds approximate estimates of the rotation and translation (up to scale) between two cameras given a set of I point matches $\{\mathbf{x}_{i1}, \mathbf{x}_{i2}\}_{i=1}^{I}$ between two images. More precisely, the algorithm assumes that the first camera is at the world origin and recovers the extrinsic parameters of the second camera.

There is a fourfold ambiguity in the possible solution due to the symmetry of the camera model - it allows for points that are behind the camera to be imaged, although this is clearly not possible in the real world. This algorithm distinguishes between these four solutions by reconstructing all of the points with each and choosing the solution where the largest number are in front of both cameras.

Algorithm 16.1: Extracting relative camera position from point matches

Input : Point pairs $\{\mathbf{x}_{i1}, \mathbf{x}_{i2}\}_{i=1}^{I}$, intrinsic matrices $\boldsymbol{\Lambda}_1, \boldsymbol{\Lambda}_2$
Output: Rotation $\boldsymbol{\Omega}$, translation $\boldsymbol{\tau}$ between cameras
begin
  // Compute fundamental matrix (algorithm 16.2)
  $\mathbf{F} = \text{ComputeFundamental}[\{\mathbf{x}_{1i}, \mathbf{x}_{2i}\}_{i=1}^{I}]$
  // Compute essential matrix
  $\mathbf{E} = \boldsymbol{\Lambda}_2^T \mathbf{F} \boldsymbol{\Lambda}_1$
  // Extract four possible rotation and translations from $\mathbf{E}$
  $\mathbf{W} = [0, -1, 0; 1, 0, 0; 0, 0, -1]$
  $[\mathbf{U}, \mathbf{L}, \mathbf{V}] = \mathbf{svd}[\mathbf{E}]$
  $\boldsymbol{\tau}_1 = \mathbf{ULWU}^T; \boldsymbol{\Omega}_1 = \mathbf{UW}^{-1}\mathbf{V}^T$
  $\boldsymbol{\tau}_2 = \mathbf{ULW}^{-1}\mathbf{U}^T; \boldsymbol{\Omega}_2 = \mathbf{UWV}^T$
  $\boldsymbol{\tau}_3 = -\boldsymbol{\tau}_1; \boldsymbol{\Omega}_3 = \boldsymbol{\Omega}_1$
  $\boldsymbol{\tau}_4 = -\boldsymbol{\tau}_2; \boldsymbol{\Omega}_4 = \boldsymbol{\Omega}_1$
  // For each possibility
  for k=*1* **to** K **do**
    $t_k = 0$ // number of points in front of camera for k^{th} soln
    // For each point
    for i=*1* **to** I **do**
      // Reconstruct point (algorithm 14.3)
      $\mathbf{w} = \text{Reconstruct}[\mathbf{x}_{i1}, \mathbf{x}_{i2}, \boldsymbol{\Lambda}_1, \boldsymbol{\Lambda}_2, \mathbf{0}, \mathbf{I}, \boldsymbol{\Omega}_k, \boldsymbol{\tau}_k]$
      // Compute point in frame of reference of second camera
      $\mathbf{w}' = \boldsymbol{\Omega}_k + \boldsymbol{\tau}_k$
      // Test if point reconstructed in front of both cameras
      if $\mathbf{w}_3 > 0$ *&* $\mathbf{w}'_3 > 0$ **then**
        $t_k = t_k + 1$
      end
    end
  end
  // Choose solution with most support
  $k = \text{argmax}_k[t_k]$
  $\boldsymbol{\Omega} = \boldsymbol{\Omega}_k$
  $\boldsymbol{\tau} = \boldsymbol{\tau}_k$
end

Algorithm 16.2: Eight point algorithm for fundamental matrix

This algorithm takes a set of $I \geq 8$ point correspondences $\{\mathbf{x}_{i1}, \mathbf{x}_{i2}\}_{i=1}^{I}$ between two images and computes the fundamental matrix using the 8 point algorithm. To improve the numerical stability of the algorithm, the point positions are transformed to have unit mean and spherical covariance before the calculation proceeds. The resulting fundamental matrix is modified to compensate for this transformation. This algorithm is usually used to compute an initial estimate for a subsequent non-linear optimization of the symmetric epipolar distance.

Algorithm 16.2: Eight point algorithm for fundamental matrix

Input : Point pairs $\{\mathbf{x}_{1i}, \mathbf{x}_{2i}\}_{i=1}^{I}$
Output: Fundamental matrix $\mathbf{F}$
begin
  `// Compute statistics of data`
  $\boldsymbol{\mu}_1 = \sum_{i=1}^{I} \mathbf{x}_{1i}/I$
  $\boldsymbol{\Sigma}_1 = \sum_{i=1}^{I} (\mathbf{x}_{1i} - \boldsymbol{\mu}_1)(\mathbf{x}_{1i} - \boldsymbol{\mu}_1)/I$
  $\boldsymbol{\mu}_2 = \sum_{i=1}^{I} \mathbf{x}_{2i}/I$
  $\boldsymbol{\Sigma}_2 = \sum_{i=1}^{I} (\mathbf{x}_{2i} - \boldsymbol{\mu}_2)(\mathbf{x}_{2i} - \boldsymbol{\mu}_2)/I$
  for k=*1* **to** K **do**
    `// Compute transformed coordinates`
    $\mathbf{x}_{i1} = \boldsymbol{\Sigma}_1^{-1/2}(\mathbf{x}_{i1} - \boldsymbol{\mu}_1)$
    $\mathbf{x}_{i2} = \boldsymbol{\Sigma}_2^{-1/2}(\mathbf{x}_{i2} - \boldsymbol{\mu}_2)$
    `// Compute constraint`
    $\mathbf{A}_i = [x_{i2}x_{i1}, x_{i2}y_{i1}, x_{i2}, y_{i2}x_{i1}, y_{i2}y_{i1}, y_{i2}, x_{i1}, y_{i1}, 1]$
  end
  `// Append constraints and solve`
  $\mathbf{A} = [\mathbf{A}_1; \mathbf{A}_2; \ldots \mathbf{A}_I]$
  $[\mathbf{U}, \mathbf{L}, \mathbf{V}] = \mathbf{svd}[\mathbf{A}]$
  $\mathbf{F} = [v_{19}, v_{29}, v_{39}; v_{49}, v_{59}, v_{69}; v_{79}, v_{89}, v_{99}]$
  `// Compensate for transformation`
  $\mathbf{T}_1 = [\boldsymbol{\Sigma}_1^{-1/2}, \boldsymbol{\Sigma}_1^{-1/2}\boldsymbol{\mu}_1; 0, 0, 1]$
  $\mathbf{T}_2 = [\boldsymbol{\Sigma}_2^{-1/2}, \boldsymbol{\Sigma}_2^{-1/2}\boldsymbol{\mu}_2; 0, 0, 1]$
  $\mathbf{F} = \mathbf{T}_2^T\mathbf{F}\mathbf{T}_1$
  `// Ensure that matrix has rank 2`
  $[\mathbf{U}, \mathbf{L}, \mathbf{V}] = \mathbf{svd}[\mathbf{F}]$
  $l_{33} = 0$
  $\mathbf{F} = \mathbf{U}\mathbf{L}\mathbf{V}^T$
end

Algorithm 16.3: Robust computation of fundamental matrix with RANSAC

The goal of this algorithm is to estimate the fundamental matrix from 2D point pairs $\{\mathbf{x}_{i1}, \mathbf{x}_{i2}\}_{i=1}^{I}$ to another in the case where some of the point matches are known to be wrong (outliers). The robustness is achieved by applying the RANSAC algorithm. Since the fundamental matrix has a eight unknown quantities, we randomly select eight point pairs at each stage of the algorithm (each pair contributes one constraint). The algorithm also returns the true matches.

Algorithm 16.3: Robust ML fitting of fundamental matrix

Input : Point pairs $\{\mathbf{x}_{i1}, \mathbf{x}_{i2}\}_{i=1}^{I}$, number of RANSAC steps N, threshold τ
Output: Fundamental matrix $\mathbf{F}$, set of inlier indices $\mathcal{I}$
begin
  `// Initialize best inlier set to empty`
  $\mathcal{I} = \{\}$
  for n=*1* **to** N **do**
    `// Draw 8 different random integers between 1 and` I
    $\mathcal{R} = \text{RandomSubset}[\{1 \ldots I\}, 8]$
    `// Compute fundamental matrix (algorithm 16.2)`
    $\mathbf{\Phi}_n = \text{ComputeFundamental}[\{\mathbf{x}_{i1}\}_{i\in\mathcal{R}}, \{\mathbf{x}_{i2}\}_{i\in\mathcal{R}}]$
    `// Initialize set of inliers to empty`
    $\mathcal{S}_n = \{\}$
    for i=*1* **to** I **do**
      `// Compute epipolar line in first image`
      $\tilde{\mathbf{x}}_{i2} = [\mathbf{x}_{i2}; 1]$
      $\mathbf{l} = \tilde{x}_{i2}^T\mathbf{F}$
      `// Compute squared distance to epipolar line`
      $d_1 = (l_1 x_{i1} + l_2 y_{i1} + l_3)^2/(l_1^2 + l_2^2)$
      `// Compute epipolar line in second image`
      $\tilde{\mathbf{x}}_{i1} = [\mathbf{x}_{i1}; 1]$
      $\mathbf{l}_2 = \mathbf{F}\tilde{x}_{i1}$
      `// Compute squared distance to epipolar line`
      $d_2 = (l_1 x_{i2} + l_2 y_{i2} + l_3)^2/(l_1^2 + l_2^2)$
      `// If small enough then add to inliers`
      if $(d_1 < \tau^2)$ && $(d_2 < \tau^2)$ **then**
        $\mathcal{S}_n = \mathcal{S}_n \cap \{i\}$
      end
    end
    `// If best outliers so far then store`
    if $|\mathcal{S}_n| > |\mathcal{I}|$ **then**
      $\mathcal{I} = \mathcal{S}_n$
    end
  end
  `// Compute fundamental matrix from all outliers`
  $\mathbf{\Phi} = \text{ComputeFundamental}[\{\mathbf{x}_{i1}\}_{i\in\mathcal{I}}, \{\mathbf{x}_{i2}\}_{i\in\mathcal{I}}]$
end

Algorithm 16.4: Planar rectification

This algorithm computes homographies that can be used to rectify the two images. The homography for this second image is chosen so that it moves the epipole to infinity along the $x-$axis. The homography for the first image is chosen so that the matches are on the same horizontal lines as in the first image and the distance between the matches is smallest in a least squares sense (i.e., the disparity is smallest).

Algorithm 16.4: Planar rectification

Input : Point pairs $\{\mathbf{x}_{i1}, \mathbf{x}_{i2}\}_{i=1}^{I}$
Output: Homographies $\boldsymbol{\Phi}_1$, $\boldsymbol{\Phi}_2$ to transform first and second images
begin
  `// Compute fundamental matrix (algorithm 55)`
  $\mathbf{F} = \text{ComputeFundamental}[\{\mathbf{x}_{1i}, \mathbf{x}_{2i}\}_{i=1}^{I}]$
  `// Compute epipole in image 2`
  $[\mathbf{U}, \mathbf{L}, \mathbf{V}] = \mathbf{svd}[\mathbf{F}]$
  $\mathbf{e} = [u_{13}, u_{23}, u_{33}]^T$
  `// Compute three transformation matrices`
  $\mathbf{T}_1 = [0, 0, -\delta_x; 0, 0, \delta_y, 0, 0, 1]$
  $\theta = \text{atan2}[e_y - \delta_y, e_x - \delta_x]$
  $\mathbf{T}_2 = [\cos[\theta], \sin[\theta], 0; -\sin[\theta], \cos[\theta], 0; 0, 0, 1]$
  $\mathbf{T}_3 = [1, 0, 0; 0, 1, 0, -1/(\cos[\theta], \sin[\theta]), 0, 1]]$
  `// Compute homography for second image`
  $\boldsymbol{\Phi}_2 = \mathbf{T}_3\mathbf{T}_2, \mathbf{T}_1$
  `// Compute factorization of fundamental matrix`
  $\mathbf{L} = \mathbf{diag}[l_{11}, l_{22}, (l_{11} + l_{22})/2]$
  $\mathbf{W} = [0, -1, 0; 1, 0, 0; 0, 0, 1]$
  $\mathbf{M} = \mathbf{ULWV}^T$
  `// Prepare matrix for soln for` $\boldsymbol{\Phi}_1$
  for *k=1* **to** K **do**
    $\mathbf{x}'_{i1} = \mathbf{hom}[\mathbf{x}_{i1}, \boldsymbol{\Phi}_2\mathbf{M}]$
    `// Transform points`
    $\mathbf{x}'_{i2} = \mathbf{hom}[\mathbf{x}_{i2}, \boldsymbol{\Phi}_2]$
    `// Create elements of A and b`
    $\mathbf{A}_i = [x'_{i1}, y'_{i1}, 1]$
    $b_i = x'_{i2}$
  end
  `// Concatenate elements of A and b`
  $\mathbf{A} = [\mathbf{A}_1; \mathbf{A}_2; \ldots \mathbf{A}_I]$
  $\mathbf{b} = [b_1; b_2; \ldots b_I]$
  `// Solve for` $\boldsymbol{\alpha}$
  $\boldsymbol{\alpha} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$
  `// Calculate homography in first image`
  $\boldsymbol{\Phi}_1 = (\mathbf{I} + [1, 0, 0]^T\boldsymbol{\alpha}^T)\boldsymbol{\Phi}_2\mathbf{M}$
end

Algorithm 17.1: Generalized Procrustes analysis

The goal of generalized Procrustes analysis is to align a set of shape vectors $\{\mathbf{w_i}\}_{i=1}^{I}$ with respect to a given transformation family (Euclidean, similarity, affine etc.). Each shape vector consists of a set of N 2D points $\mathbf{w}_i = [\mathbf{w}_{i1}^T, \mathbf{w}_{i2}^T, \ldots \mathbf{w}_{iN}^T]^T$. In the algorithm below, we will use the example of registering with respect to a Similarity transformation, which consists of a rotation $\mathbf{\Omega}$, scaling ρ and translation $\boldsymbol{\tau}$.

Algorithm 17.1: Generalized Procrustes analysis

Input : Shape vectors $\{\mathbf{w}_i\}_{i=1}^{I}$, number of factors, K
Output: Template $\tilde{\mathbf{w}}$, transformations $\{\mathbf{\Omega}_i, \boldsymbol{\rho}_i, \boldsymbol{\tau}_i\}_{i=1}^{I}$, number of iterations K
begin
  Initialize $\tilde{\mathbf{w}} = \mathbf{w}_1$
  `// Main iteration loop`
  for k=*1* **to** K **do**
    `// For each transformation`
    for i=*1* **to** I **do**
      `// Compute transformation to template (algorithm 15.2)`
      $[\mathbf{\Omega}_i, \rho_i, \boldsymbol{\tau}_i] = \text{EstimateSimilarity}[\{\tilde{\mathbf{w}}_n\}_{n=1}^{N}, \{\mathbf{w}_{in}\}_{n=1}^{N}]$
    end
    `// Update template (average of inverse transform)`
    $\tilde{\mathbf{w}}_i = \sum_{i=1}^{I} \mathbf{\Omega}_i^T(\mathbf{w}_{in} - \boldsymbol{\tau}_i)/(I\rho_i)$
    `// Normalize template`
    $\tilde{\mathbf{w}}_i = \tilde{\mathbf{w}}_i/|\tilde{\mathbf{w}}_i|$
  end
end

Algorithm 17.2: Probabilistic principal components analysis

The probabilistic principal components analysis algorithm describes a set of I $D \times 1$ data examples $\{\mathbf{x}_i\}_{i=1}^I$ with the model

$$Pr(\mathbf{x}_i) = \text{Norm}_{\mathbf{x}_i}[\boldsymbol{\mu}, \boldsymbol{\Phi}\boldsymbol{\Phi}^T + \sigma^2\mathbf{I}]$$

where $\boldsymbol{\mu}$ is the $D \times 1$ mean vector, $\boldsymbol{\Phi}$ is a $D \times K$ matrix containing the K principal components in its columns. The principal components define a K dimensional subspace and the parameter σ^2 explains the variation of the data around this subspace.

Notice that this model is very similar to factor analysis (see Algorithm 6.3). The only difference is that here we have spherical additive noise $\sigma^2\mathbf{I}$ rather than a diagonal noise components $\boldsymbol{\Sigma}$. This small change has important ramifications for the learning algorithm; we no longer need to use an iterative learning procedure based on the EM algorithm and can instead learn the parameters in closed form.

Algorithm 17.2: ML learning of PPCA model

Input : Training data $\{\mathbf{x}_i\}_{i=1}^I$, number of principal components, K
Output: Parameters $\boldsymbol{\mu}, \boldsymbol{\Phi}, \sigma^2$
begin
  `// Estimate mean parameter`
  $\boldsymbol{\mu} = \sum_{i=1}^I \mathbf{x}_i / I$
  `// Form matrix of mean-zero data`
  $\mathbf{X} = [\mathbf{x}_1 - \boldsymbol{\mu}, \mathbf{x}_2 - \boldsymbol{\mu}, \ldots, \mathbf{x}_I - \boldsymbol{\mu}]$
  `// Decompose` $\mathbf{X}$ `to matrices` $\mathbf{U}, \mathbf{L}, \mathbf{V}$
  $[\mathbf{V}\mathbf{L}\mathbf{V}^T] = \mathbf{svd}[\mathbf{X}^T\mathbf{X}]$
  $\mathbf{U} = \mathbf{W}\mathbf{V}\mathbf{L}^{-1/2}$
  `// Estimate noise parameter`
  $\sigma^2 = \sum_{j=K+1}^D l_{jj}/(D-K)$
  `// Estimate principal components`
  $\mathbf{U}_k = [\mathbf{u}_1, \mathbf{u}_2, \ldots \mathbf{u}_K]$
  $\mathbf{L}_k = \mathbf{diag}[l_{11}, l_{22}, \ldots l_{KK}]$
  $\boldsymbol{\Phi} = \mathbf{U}_K(\mathbf{L}_K - \sigma^2\mathbf{I})^{1/2}$
end

Algorithm 18.1: ML learning of subspace identity model

This describes the jth of J data examples from the ith of I identities as

$$\mathbf{x}_{ij} = \boldsymbol{\mu} + \boldsymbol{\Phi}\mathbf{h}_i + \boldsymbol{\epsilon}_{ij},$$

where $\mathbf{x}_{ij}$ is the $D\times 1$ observed data, $\boldsymbol{\mu}$ is the $D\times 1$ mean vector, $\boldsymbol{\Phi}$ is the $D\times K$ factor matrix, $\mathbf{h}_i$ is the $K\times 1$ hidden variable representing the identity and $\boldsymbol{\epsilon}_{ij}$ is a $D\times 1$ additive normal noise multivariate noise with diagonal covariance $\boldsymbol{\Sigma}$.

Algorithm 18.1: Maximum likelihood learning for identity subspace model

Input : Training data $\{\mathbf{x}_{ij}\}_{i=1,j=1}^{I,J}$, number of factors, K
Output: Maximum likelihood estimates of parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\Phi}, \boldsymbol{\Sigma}\}$
begin
  Initialize $\boldsymbol{\theta} = \boldsymbol{\theta}_0$ [a]
  `// Set mean`
  $\boldsymbol{\mu} = \sum_{i=1}^{I}\sum_{j=1}^{J}\mathbf{x}_{ij}/IJ$
  repeat
    `// Expectation step`
    for *i=1* **to** I **do**
      $\mathsf{E}[\mathbf{h}_i] = (J\boldsymbol{\Phi}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\Phi} + \mathbf{I})^{-1}\boldsymbol{\Phi}^T\boldsymbol{\Sigma}^{-1}\sum_{j=1}^{J}(\mathbf{x}_{ij} - \boldsymbol{\mu})$
      $\mathsf{E}[\mathbf{h}_i\mathbf{h}_i^T] = (J\boldsymbol{\Phi}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\Phi} + \mathbf{I})^{-1} + \mathsf{E}[\mathbf{h}_i]\mathsf{E}[\mathbf{h}_i]^T$
    end
    `// Maximization step`
    $\boldsymbol{\Phi} = \left(\sum_{i=1}^{I}\sum_{j=1}^{J}(\mathbf{x}_{ij} - \boldsymbol{\mu})\mathsf{E}[\mathbf{h}_i]^T\right)\left(\sum_{i=1}^{I}J\mathsf{E}[\mathbf{h}_i\mathbf{h}_i^T]\right)^{-1}$
    $\boldsymbol{\Sigma} = \frac{1}{IJ}\sum_{i=1}^{I}\sum_{j=1}^{J}\mathsf{diag}\left[(\mathbf{x}_{ij} - \boldsymbol{\mu})(\mathbf{x}_{ij} - \boldsymbol{\mu})^T - \boldsymbol{\Phi}\mathsf{E}[\mathbf{h}_i](\mathbf{x}_{ij} - \boldsymbol{\mu})^T\right]$
    `// Compute data log likelihood`
    for *i=1* **to** I **do**
      $\mathbf{x}_i' = [\mathbf{x}_{i1}^T, \mathbf{x}_{i2}^T, \ldots, \mathbf{x}_{iJ}^T]^T$ `// compound data vector,` $JD\times 1$
    end
    $\boldsymbol{\mu}' = [\boldsymbol{\mu}^T, \boldsymbol{\mu}^T \ldots \boldsymbol{\mu}^T]^T$ `// compound mean vector,` $JD\times 1$
    $\boldsymbol{\Phi}' = [\boldsymbol{\Phi}^T, \boldsymbol{\Phi}^T \ldots \boldsymbol{\Phi}^T]^T$ `// compound factor matrix,` $JD\times K$
    $\boldsymbol{\Sigma}' = \mathsf{diag}[\boldsymbol{\Sigma}, \boldsymbol{\Sigma}, \ldots \boldsymbol{\Sigma}]$ `// compound covariance,` $JD\times JD$
    $L = \sum_{i=1}^{I}\log\left[\mathsf{Norm}_{\mathbf{x}_i'}[\boldsymbol{\mu}', \boldsymbol{\Phi}'\boldsymbol{\Phi}'^T + \boldsymbol{\Sigma}']\right]$ [b]
  until *No further improvement in L*
end

[a] It is usual to initialize $\boldsymbol{\Phi}$ to random values. The D diagonal elements of $\boldsymbol{\Sigma}$ can be initialized to the variances of the D data dimensions.
[b] In high dimensions it is worth reformulating the covariance of this matrix using the matrix inversion lemma.

Algorithm 18.2: ML learning of PLDA model

PLDA describes the jth of J data examples from the ith of I identities as

$$\mathbf{x}_{ij} = \boldsymbol{\mu} + \boldsymbol{\Phi}\mathbf{h}_i + \boldsymbol{\Psi}\mathbf{s}_{ij} + \boldsymbol{\epsilon}_{ij},$$

where all terms are the same as in subspace identity model but now we add $\boldsymbol{\Psi}$, the $D \times L$ within-individual factor matrix and $\mathbf{s}_{ij}$ the $L \times 1$ style variable.

Algorithm 18.2: Maximum likelihood learning for PLDA model

Input : Training data $\{\mathbf{x}_{ij}\}_{i=1,j=1}^{I,J}$, numbers of factors, K, L
Output: Maximum likelihood estimates of parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\Phi}, \boldsymbol{\Psi}, \boldsymbol{\Sigma}\}$
begin
  Initialize $\boldsymbol{\theta} = \boldsymbol{\theta}_0$ [a]
  // Set mean
  $\boldsymbol{\mu} = \sum_{i=1}^{I}\sum_{j=1}^{J}\mathbf{x}_{ij}/IJ$
  repeat
    $\boldsymbol{\mu}' = [\boldsymbol{\mu}^T, \boldsymbol{\mu}^T \ldots \boldsymbol{\mu}^T]^T$ // compound mean vector, $JD \times 1$
    $\boldsymbol{\Phi}' = [\boldsymbol{\Phi}^T, \boldsymbol{\Phi}^T \ldots \boldsymbol{\Phi}^T]^T$ // compound factor matrix 1, $JD \times K$
    $\boldsymbol{\Psi}' = \text{diag}[\boldsymbol{\Psi}, \boldsymbol{\Psi}, \ldots \boldsymbol{\Psi}]$ // compound factor matrix 2, $JD \times JL$
    $\boldsymbol{\Phi}' = [\boldsymbol{\Phi}', \boldsymbol{\Psi}']$ // concatenate matrices $JD \times (K+JL)$
    $\boldsymbol{\Sigma}' = \text{diag}[\boldsymbol{\Sigma}, \boldsymbol{\Sigma}, \ldots \boldsymbol{\Sigma}]$ // compound covariance, $JD \times JD$
    // Expectation step
    for i=1 **to** I **do**
      $\mathbf{x}_i' = [\mathbf{x}_{i1}^T, \mathbf{x}_{i2}^T, \ldots, \mathbf{x}_{iJ}^T]^T$ // compound data vector, $JD \times 1$
      $\boldsymbol{\mu}_{\mathbf{h}_i'} = (\boldsymbol{\Phi}'^T\boldsymbol{\Sigma}'^{-1}\boldsymbol{\Phi}' + \mathbf{I})^{-1}\boldsymbol{\Phi}'^T\boldsymbol{\Sigma}'^{-1}(\mathbf{x}_i' - \boldsymbol{\mu}')$
      $\boldsymbol{\Sigma}_{\mathbf{h}_i'} = (\boldsymbol{\Phi}'^T\boldsymbol{\Sigma}'^{-1}\boldsymbol{\Phi}' + \mathbf{I})^{-1} + \mathsf{E}[\mathbf{h}_i']\mathsf{E}[\mathbf{h}_i']^T$
      for j=1 **to** J **do**
        $\mathcal{S}_{ij} = [1 \ldots K, K+(J-1)L+1 \ldots K+JL]$
        $E[\mathbf{h}_{ij}''] = \boldsymbol{\mu}_{\mathbf{h}_i'}(\mathcal{S}_{ij})$ // Extract subvector of mean
        $E[\mathbf{h}_{ij}''\mathbf{h}_{ij}''^T] = \boldsymbol{\Sigma}_{\mathbf{h}_i'}(\mathcal{S}_{ij}, \mathcal{S}_{ij})$ // Extract submatrix from covariance
      end
    end
    // Maximization step
    $\boldsymbol{\Phi}'' = \left(\sum_{i=1}^{I}\sum_{j=1}^{J}(\mathbf{x}_{ij} - \boldsymbol{\mu})\mathsf{E}[\mathbf{h}_{ij}'']^T\right)\left(\sum_{i=1}^{I}\sum_{j=1}^{J}\mathsf{E}[\mathbf{h}_{ij}''\mathbf{h}_{ij}''^T]\right)^{-1}$
    $\boldsymbol{\Sigma} = \frac{1}{IJ}\sum_{i=1}^{I}\sum_{j=1}^{J}\text{diag}\left[(\mathbf{x}_{ij} - \boldsymbol{\mu})(\mathbf{x}_{ij} - \boldsymbol{\mu})^T - [\boldsymbol{\Phi}, \Psi]\mathsf{E}[\mathbf{h}_{ij}](\mathbf{x}_{ij} - \boldsymbol{\mu})^T\right]$
    $\boldsymbol{\Phi} = \boldsymbol{\Phi}''(:, 1:K)$ // Extract original factor matrix
    $\boldsymbol{\Psi} = \boldsymbol{\Phi}''(:, K+1:K+L)$ // Extract other factor matrix
    // Compute data log likelihood
    $L = \sum_{i=1}^{I}\log\left[\mathsf{Norm}_{\mathbf{x}_i'}[\boldsymbol{\mu}', \boldsymbol{\Phi}'\boldsymbol{\Phi}'^T + \Sigma']\right]$
  until *No further improvement in* L
end

[a] Initialize $\boldsymbol{\Psi}$ to random values, other variables as in identity subspace model.

Algorithm 18.3: ML learning of asymmetric bilinear model

This describes the jth data example from the ith identities and the kth styles as

$$\mathbf{x}_{ijs} = \boldsymbol{\mu}_s + \boldsymbol{\Phi}_s \mathbf{h}_i + \boldsymbol{\epsilon}_{ijs},$$

where the terms have the same interpretation as for the subspace identity model except now there is one set of parameters $\boldsymbol{\theta}_s = \{\boldsymbol{\mu}_s, \boldsymbol{\Phi}_s, \boldsymbol{\Sigma}_s\}$ per style, s.

Algorithm 18.3: Maximum likelihood learning for asymmetric bilinear model

Input : Training data $\{\mathbf{x}_{ij}\}_{i=1,j=1,s=1}^{I,J,S}$, number of factors, K
Output: ML estimates of parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}_{1\ldots S}, \boldsymbol{\Phi}_{1\ldots S}, \boldsymbol{\Sigma}_{1\ldots S}\}$
begin
  Initialize $\boldsymbol{\theta} = \boldsymbol{\theta}_0$
  for s=*1* **to** S **do**
    $\boldsymbol{\mu}_s = \sum_{i=1}^{I}\sum_{j=1}^{J} \mathbf{x}_{ijs}/IJ$ `// Set mean`
  end
  repeat
    `// Expectation step`
    for i=*1* **to** I **do**
      $\mathsf{E}[\mathbf{h}_i] = (\mathbf{I} + J\sum_{s=1}^{S} \boldsymbol{\Phi}_s^T \boldsymbol{\Sigma}_s^{-1} \boldsymbol{\Phi}_s)^{-1} \sum_{s=1}^{S} \boldsymbol{\Phi}_s^T \boldsymbol{\Sigma}_s^{-1} \sum_{j=1}^{J} (\mathbf{x}_{ijs} - \boldsymbol{\mu}_s)$
      $\mathsf{E}[\mathbf{h}_i\mathbf{h}_i^T] = (\mathbf{I} + J\boldsymbol{\Phi}_s^T \boldsymbol{\Sigma}_s^{-1} \boldsymbol{\Phi}_s)^{-1} + \mathsf{E}[\mathbf{h}_i]\mathsf{E}[\mathbf{h}_i]^T$
    end
    `// Maximization step`
    for s=*1* **to** S **do**
      $\boldsymbol{\Phi}_s = \left(\sum_{i=1}^{I}\sum_{j=1}^{J} (\mathbf{x}_{ijs} - \boldsymbol{\mu}_s)\mathsf{E}[\mathbf{h}_i]^T\right)\left(\sum_{i=1}^{I} J\mathsf{E}[\mathbf{h}_i\mathbf{h}_i^T]\right)^{-1}$
      $\boldsymbol{\Sigma}_s = \frac{1}{IJ}\sum_{i=1}^{I}\sum_{j=1}^{J} \mathsf{diag}\left[(\mathbf{x}_{ijs} - \boldsymbol{\mu}_s)(\mathbf{x}_{ijs} - \boldsymbol{\mu}_s)^T - \boldsymbol{\Phi}_s\mathsf{E}[\mathbf{h}_i](\mathbf{x}_{ijs} - \boldsymbol{\mu}_s)^T\right]$
    end
    `// Compute data log likelihood`
    for s=*1* **to** S **do**
      $\boldsymbol{\Phi}_s' = [\boldsymbol{\Phi}_s^T, \boldsymbol{\Phi}_s^T \ldots \boldsymbol{\Phi}_s^T]^T$
      $\boldsymbol{\Sigma}_s' = \mathsf{diag}[\boldsymbol{\Sigma}_s, \boldsymbol{\Sigma}_s, \ldots \boldsymbol{\Sigma}_s]$
      for i=*1* **to** I **do**
        $\mathbf{x}_{is}' = [\mathbf{x}_{i1s}^T, \mathbf{x}_{i2s}^T, \ldots, \mathbf{x}_{iJs}^T]^T$
        $\mathbf{x}_i' = [\mathbf{x}_{i1}^T, \mathbf{x}_{i2}^T, \ldots, \mathbf{x}_{iS}^T]^T$ `// compound data vector,` $JSD\times 1$
      end
    end
    $\boldsymbol{\mu}' = [\boldsymbol{\mu}^T, \boldsymbol{\mu}^T \ldots \boldsymbol{\mu}^T]^T$ `// compound mean vector,` $JSD\times 1$
    $\boldsymbol{\Phi}' = [\boldsymbol{\Phi}_1'^T, \boldsymbol{\Phi}_2'^T \ldots \boldsymbol{\Phi}_S'^T]^T$ `// compound factor matrix,` $JSD\times K$
    $\boldsymbol{\Sigma}' = \mathsf{diag}[\boldsymbol{\Sigma}_1', \boldsymbol{\Sigma}_2', \ldots \boldsymbol{\Sigma}_S']$ `// compound covariance,` $JSD\times JSD$
    $L = \sum_{i=1}^{I} \log\left[\mathsf{Norm}_{\mathbf{x}_i'}[\boldsymbol{\mu}', \boldsymbol{\Phi}'\boldsymbol{\Phi}'^T + \boldsymbol{\Sigma}']\right]$
  until *No further improvement in* L
end

Algorithm 18.4: Style translation with asymmetric bilinear model

To translate a data example from one style to another we first estimate the hidden variable associated with the example, and then use the generative equation to simulate the new style. We cannot know the hidden variable for certain, but we can compute it's posterior distribution, which has a Gaussian form, and then choose the MAP solution which is the mean of this Gaussian.

Algorithm 18.4: Style translation with asymmetric bilinear model

Input : Example $\mathbf{x}$ in style s_1, model parameters $\boldsymbol{\theta}$
Output: Prediction for data $\mathbf{x}^*$ in style s_2
begin
  `// Estimate hidden variable`
  $\mathsf{E}[\mathbf{h}] = (\mathbf{I} + \boldsymbol{\Phi}_{s_1}^T \boldsymbol{\Sigma}_{s_1}^{-1} \boldsymbol{\Phi}_{s_1})^{-1} \boldsymbol{\Phi}_{s_1}^T \boldsymbol{\Sigma}_{s_1}^{-1} (\mathbf{x} - \boldsymbol{\mu}_{s_1})$
  `// Predict in different style`
  $\mathbf{x}^* = \boldsymbol{\mu}_{s_2} + \boldsymbol{\Phi}_{s_2} E[\mathbf{h}]$
end

Algorithm 19.1: Kalman filter

To define the Kalman filter, we must specify the temporal and measurement models. First, the temporal model relates the states $\mathbf{w}$ at times $t-1$ and t and is given by

$$Pr(\mathbf{w}_t|\mathbf{w}_{t-1}) = \text{Norm}_{\mathbf{w}_t}[\boldsymbol{\mu}_p + \boldsymbol{\Psi}\mathbf{w}_{t-1}, \boldsymbol{\Sigma}_p].$$

where $\boldsymbol{\mu}_p$ is a $D_{\mathbf{w}}\times 1$ vector, which represents the mean change in the state and $\boldsymbol{\Psi}$ is a $D_{\mathbf{w}}\times D_{\mathbf{w}}$ matrix, which relates the mean of the state at time t to the state at time $t-1$. This is known as the *transition* matrix. The transition noise $\boldsymbol{\Sigma}_p$ determines how closely related the states are at times t and $t-1$.

Second, the measurement model relates the data $\mathbf{x}_t$ at time t to the state $\mathbf{w}_t$,

$$Pr(\mathbf{x}_t|\mathbf{w}_t) = \text{Norm}_{\mathbf{x}_t}[\boldsymbol{\mu}_m + \boldsymbol{\Phi}\mathbf{w}_t, \boldsymbol{\Sigma}_m].$$

where $\boldsymbol{\mu}_m$ is a $D_{\mathbf{x}}\times 1$ mean vector and $\boldsymbol{\Phi}$ is a $D_{\mathbf{x}}\times D_{\mathbf{w}}$ matrix relating the $D_{\mathbf{x}}\times 1$ measurement vector to the $D_{\mathbf{w}}\times 1$ state. The measurement noise $\boldsymbol{\Sigma}_m$ defines additional uncertainty on the measurements that cannot be explained by the state.

The Kalman filter is a set of rules for computing the marginal posterior probability $Pr(\mathbf{w}_t|\mathbf{x}_{1\ldots t})$ based on a normally distributed estimate of the marginal posterior probability $Pr(\mathbf{w}_{t-1}|\mathbf{x}_{1\ldots t-1})$ at the previous time and a new measurement $\mathbf{x}_t$. In this algorithm we denote the mean of the posterior marginal probability as $\boldsymbol{\mu}_{t-1}$ and the variance as $\boldsymbol{\Sigma}_{t-1}$.

Algorithm 19.1: The Kalman filter

Input : Measurements $\{\mathbf{x}\}_{t=1}^{T}$, temporal params $\boldsymbol{\mu}_p, \boldsymbol{\Psi}, \boldsymbol{\Sigma}_p$, measurement params $\boldsymbol{\mu}_m, \boldsymbol{\Phi}, \boldsymbol{\Sigma}_m$
Output: Means $\{\boldsymbol{\mu}_t\}_{t=1}^{T}$ and covariances $\{\boldsymbol{\Sigma}_t\}_{t=1}^{T}$ of marginal posterior distributions
begin
  // Initialize mean and covariance
  $\boldsymbol{\mu}_0 = \mathbf{0}$
  $\boldsymbol{\Sigma}_0 = \boldsymbol{\Sigma}_0$ // Typically set to large multiple of identity
  // For each time step
  for t=1 **to** T **do**
    // State prediction
    $\boldsymbol{\mu}_+ = \boldsymbol{\mu}_p + \boldsymbol{\Psi}\boldsymbol{\mu}_{t-1}$
    // Covariance prediction
    $\boldsymbol{\Sigma}_+ = \boldsymbol{\Sigma}_p + \boldsymbol{\Psi}\boldsymbol{\Sigma}_{t-1}\boldsymbol{\Psi}^T$
    // Compute Kalman gain
    $\mathbf{K} = \boldsymbol{\Sigma}_+\boldsymbol{\Phi}^T(\boldsymbol{\Sigma}_m + \boldsymbol{\Phi}\boldsymbol{\Sigma}_+\boldsymbol{\Phi}^T)^{-1}$
    // State update
    $\boldsymbol{\mu}_t = \boldsymbol{\mu}_+ + \mathbf{K}(\mathbf{x}_t - \boldsymbol{\mu}_m - \boldsymbol{\Phi}\boldsymbol{\mu}_+)$
    // Covariance update
    $\boldsymbol{\Sigma}_t = (\mathbf{I} - \mathbf{K}\boldsymbol{\Phi})\boldsymbol{\Sigma}_+$
  end
end

Algorithm 19.2: Fixed interval Kalman smoother

The fixed interval smoother consists of a backward set of recursions that estimate the marginal posterior distributions $Pr(\mathbf{w}_t|\mathbf{x}_{1...T})$ of the state at each time step, taking into account all of the measurements $\mathbf{x}_{1...T}$. In these recursions, the marginal posterior distribution $Pr(\mathbf{w}_t|\mathbf{x}_{1...T})$ of the state at time t is updated, and, based on this result, the marginal posterior $Pr(\mathbf{w}_{t-1}|\mathbf{x}_{1...T})$ at time $t-1$ is updated and so on.

In the algorithm, we denote the mean and variance of the marginal posterior $Pr(\mathbf{w}_t|\mathbf{x}_{1...T})$ at time t by $\boldsymbol{\mu}_{t|T}$ and $\boldsymbol{\Sigma}_{t|T}$, respectively The notation $\boldsymbol{\mu}_{+|t}$ and $\boldsymbol{\Sigma}_{+|t}$ denotes the mean and variance of the posterior distribution $Pr(\mathbf{w}_t|\mathbf{x}_{1...t-1})$ of the state at time t based on the measurements up to time $t-1$ (i.e., what we denoted as $\boldsymbol{\mu}_+$ and $\boldsymbol{\Sigma}_+$ during the forward Kalman filter recursions).

Algorithm 19.2: Fixed interval Kalman smoother

Input : Means, variances $\{\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t}, \boldsymbol{\mu}_{+|t}, \boldsymbol{\Sigma}_{+|t}\}_{t=1}^{T}$, temporal param $\boldsymbol{\Psi}$
Output: Means $\{\boldsymbol{\mu}_{t|T}\}_{t=1}^{T}$ and covariances $\{\boldsymbol{\Sigma}_{t|T}\}_{t=1}^{T}$ of marginal posterior distributions
begin
  `// For each time step`
  for *t=T-1* **to** 1 **do**
    `// Compute gain matrix`
    $\mathbf{C}_t = \boldsymbol{\Sigma}_{t|t}\boldsymbol{\Psi}^T\boldsymbol{\Sigma}_{+|t}^{-1}$
    `// Compute mean`
    $\boldsymbol{\mu}_{t|T} = \boldsymbol{\mu}_t + \mathbf{C}_t(\boldsymbol{\mu}_{t+1|T} - \boldsymbol{\mu}_{+|t})$
    `// Compute variance`
    $\boldsymbol{\Sigma}_{t|T} = \boldsymbol{\Sigma}_t + \mathbf{C}_t(\boldsymbol{\Sigma}_{t+1|T} - \boldsymbol{\Sigma}_{+|t})\mathbf{C}_t^T$
  end
end

Algorithm 19.3: Extended Kalman filter

The extended Kalman filter (EKF) is designed to cope with more general temporal models, where the relationship between the states at time t is an arbitrary nonlinear function $\mathbf{f}[\bullet,\bullet]$ of the state at the previous time step and a stochastic contribution $\boldsymbol{\epsilon}_p$

$$\mathbf{w}_t = \mathbf{f}[\mathbf{w}_{t-1}, \boldsymbol{\epsilon}_p],$$

where the covariance of the noise term $\boldsymbol{\epsilon}_p$ is $\boldsymbol{\Sigma}_p$ as before. Similarly, it can cope with a nonlinear relationship $\mathbf{g}[\bullet,\bullet]$ between the state and the measurements

$$\mathbf{x}_t = \mathbf{g}[\mathbf{w}_t, \boldsymbol{\epsilon}_m],$$

where the covariance of $\boldsymbol{\epsilon}_m$ is $\boldsymbol{\Sigma}_m$.

The extended Kalman filter works by taking linear approximations to the nonlinear functions at the peak $\boldsymbol{\mu}_t$ of the current estimate using the Taylor expansion. We define the Jacobian matrices,

$$\boldsymbol{\Psi} = \left.\frac{\partial \mathbf{f}[\mathbf{w}_{t-1}, \boldsymbol{\epsilon}_p]}{\partial \mathbf{w}_{t-1}}\right|_{\boldsymbol{\mu}_{t-1},\mathbf{0}} \qquad \boldsymbol{\Upsilon}_p = \left.\frac{\partial \mathbf{f}[\mathbf{w}_{t-1}, \boldsymbol{\epsilon}_p]}{\partial \boldsymbol{\epsilon}_p}\right|_{\boldsymbol{\mu}_{t-1},\mathbf{0}}$$
$$\boldsymbol{\Phi} = \left.\frac{\partial \mathbf{g}[\mathbf{w}_t, \boldsymbol{\epsilon}_m]}{\partial \mathbf{w}_t}\right|_{\boldsymbol{\mu}_+,\mathbf{0}} \qquad \boldsymbol{\Upsilon}_m = \left.\frac{\partial \mathbf{g}[\mathbf{w}_t, \boldsymbol{\epsilon}_m]}{\partial \boldsymbol{\epsilon}_m}\right|_{\boldsymbol{\mu}_+,\mathbf{0}},$$

where $|_{\boldsymbol{\mu}_+,0}$ denotes that the derivative is computed at position $\mathbf{w} = \boldsymbol{\mu}_+$ and $\boldsymbol{\epsilon} = \mathbf{0}$.

Algorithm 19.3: The extended Kalman filter

Input : Measurements $\{\mathbf{x}\}_{t=1}^T$, temporal function $\mathbf{f}[\bullet,\bullet]$, measurement function $\mathbf{g}[\bullet,\bullet]$
Output: Means $\{\boldsymbol{\mu}_t\}_{t=1}^T$ and covariances $\{\boldsymbol{\Sigma}_t\}_{t=1}^T$ of marginal posterior distributions
begin
  // Initialize mean and covariance
  $\boldsymbol{\mu}_0 = \mathbf{0}$
  $\boldsymbol{\Sigma}_0 = \boldsymbol{\Sigma}_0$ // Typically set to large multiple of identity
  // For each time step
  for t=1 **to** T **do**
    // State prediction
    $\boldsymbol{\mu}_+ = \mathbf{f}[\boldsymbol{\mu}_{t-1}, \mathbf{0}]$
    // Covariance prediction
    $\boldsymbol{\Sigma}_+ = \boldsymbol{\Psi}\boldsymbol{\Sigma}_{t-1}\boldsymbol{\Psi}^T + \boldsymbol{\Upsilon}_p\boldsymbol{\Sigma}_p\boldsymbol{\Upsilon}_p^T$
    // Compute Kalman gain
    $\mathbf{K} = \boldsymbol{\Sigma}_+\boldsymbol{\Phi}^T(\boldsymbol{\Upsilon}_m\boldsymbol{\Sigma}_m\boldsymbol{\Upsilon}_m^T + \boldsymbol{\Phi}\boldsymbol{\Sigma}_+\boldsymbol{\Phi}^T)^{-1}$
    // State update
    $\boldsymbol{\mu}_t = \boldsymbol{\mu}_+ + \mathbf{K}(\mathbf{x}_t - \mathbf{g}[\boldsymbol{\mu}_+, \mathbf{0}])$
    // Covariance update
    $\boldsymbol{\Sigma}_t = (\mathbf{I} - \mathbf{K}\boldsymbol{\Phi})\boldsymbol{\Sigma}_+$
  end
end

Algorithm 19.4: Iterated extended Kalman filter

The iterated extended Kalman filter passes Q times through the dataset, repeating the computations of the extended Kalman filter. At each iteration it linearizes around the previous estimate of the state, with the idea that the linear approximation will get better and better. We define the initial Jacobian matrices as before:

$$\boldsymbol{\Psi} = \left.\frac{\partial \mathbf{f}[\mathbf{w}_{t-1}, \boldsymbol{\epsilon}_p]}{\partial \mathbf{w}_{t-1}}\right|_{\boldsymbol{\mu}_{t-1}, \mathbf{0}} \qquad \boldsymbol{\Upsilon}_p = \left.\frac{\partial \mathbf{f}[\mathbf{w}_{t-1}, \boldsymbol{\epsilon}_p]}{\partial \boldsymbol{\epsilon}_p}\right|_{\boldsymbol{\mu}_{t-1}, \mathbf{0}}$$
$$\boldsymbol{\Phi}^0 = \left.\frac{\partial \mathbf{g}[\mathbf{w}_t, \boldsymbol{\epsilon}_m]}{\partial \mathbf{w}_t}\right|_{\boldsymbol{\mu}_+, \mathbf{0}} \qquad \boldsymbol{\Upsilon}^0_m = \left.\frac{\partial \mathbf{g}[\mathbf{w}_t, \boldsymbol{\epsilon}_m]}{\partial \boldsymbol{\epsilon}_m}\right|_{\boldsymbol{\mu}_+, \mathbf{0}}.$$

However, on the q^{th} iteration, we use the Jacobians

$$\boldsymbol{\Phi}^q = \left.\frac{\partial \mathbf{g}[\mathbf{w}_t, \boldsymbol{\epsilon}_m]}{\partial \mathbf{w}_t}\right|_{\boldsymbol{\mu}_t^{q-1}, \mathbf{0}} \qquad \boldsymbol{\Upsilon}^q_m = \left.\frac{\partial \mathbf{g}[\mathbf{w}_t, \boldsymbol{\epsilon}_m]}{\partial \boldsymbol{\epsilon}_m}\right|_{\boldsymbol{\mu}_t^{q-1}, \mathbf{0}},$$

where $\boldsymbol{\mu}_t^{q-1}$ is the estimate of the state at the t^{th} time step on the $q-1^{th}$ iteration.

Algorithm 19.4: The iterated extended Kalman filter

Input : Measurements $\{\mathbf{x}\}_{t=1}^T$, temporal function $\mathbf{f}[\bullet, \bullet]$, measurement function $\mathbf{g}[\bullet, \bullet]$
Output: Means $\{\boldsymbol{\mu}_t\}_{t=1}^T$ and covariances $\{\boldsymbol{\Sigma}_t\}_{t=1}^T$ of marginal posterior distributions
begin
  // For each iteration
  for q=*0* **to** Q **do**
    // Initialize mean and covariance
    $\boldsymbol{\mu}_0 = \mathbf{0}$
    $\boldsymbol{\Sigma}_0 = \boldsymbol{\Sigma}_0$ // Typically set to large multiple of identity
    // For each time step
    for t=*1* **to** T **do**
      // State prediction
      $\boldsymbol{\mu}_+ = \mathbf{f}[\boldsymbol{\mu}_{t-1}, \mathbf{0}]$
      // Covariance prediction
      $\boldsymbol{\Sigma}_+ = \boldsymbol{\Psi}\boldsymbol{\Sigma}_{t-1}\boldsymbol{\Psi}^T + \boldsymbol{\Upsilon}_p\boldsymbol{\Sigma}_p\boldsymbol{\Upsilon}_p^T$
      // Compute Kalman gain
      $\mathbf{K} = \boldsymbol{\Sigma}_+\boldsymbol{\Phi}^{qT}(\boldsymbol{\Upsilon}^q_m\boldsymbol{\Sigma}_m\boldsymbol{\Upsilon}^{qT}_m + \boldsymbol{\Phi}^q\boldsymbol{\Sigma}_+\boldsymbol{\Phi}^{qT})^{-1}$
      // State update
      $\boldsymbol{\mu}_t^q = \boldsymbol{\mu}_+ + \mathbf{K}(\mathbf{x}_t - \mathbf{g}[\boldsymbol{\mu}_+, \mathbf{0}])$
      // Covariance update
      $\boldsymbol{\Sigma}_t^q = (\mathbf{I} - \mathbf{K}\boldsymbol{\Phi}^q)\boldsymbol{\Sigma}_+$
    end
  end
end

This algorithm can be improved by running the fixed interval smoother inbetween each iteration and re-linearizing around the smoothed estimates.

Algorithm 19.5: Unscented Kalman filter

The unscented filter is an alternative to the extended Kalman filter that works by approximating the Gaussian state distribution as a set of particles with the same mean and covariance, passing these particles through the non-linear temporal / measurement equations and then recomputing the mean and covariance based on the new positions of these particles. In the example below, we assume that the state has dimensions D_w and use $2D_w + 1$ particles to approximate the world state.

Algorithm 19.5: The unscented Kalman filter

Input : Measurements $\{\mathbf{x}\}_{t=1}^{T}$, temporal, measurement functions $\mathbf{f}[\bullet,\bullet]$, $\mathbf{g}[\bullet,\bullet]$, weight a_0
Output: Means $\{\boldsymbol{\mu}_t\}_{t=1}^{T}$ and covariances $\{\boldsymbol{\Sigma}_t\}_{t=1}^{T}$ of marginal posterior distributions
begin
  `// For each time step`
  for t=*1* **to** T **do**
    `// Approximate state with particles`
    $\hat{\mathbf{w}}^{[0]} = \boldsymbol{\mu}_{t-1}$
    for j=*1* **to** D_w **do**
      $\hat{\mathbf{w}}^{[j]} = \boldsymbol{\mu}_{t-1} + \sqrt{\frac{D_\mathbf{w}}{1-a_0}}\boldsymbol{\Sigma}_{t-1}^{1/2}\mathbf{e}_j$
      $\hat{\mathbf{w}}^{[D_\mathbf{w}+j]} = \boldsymbol{\mu}_{t-1} - \sqrt{\frac{D_\mathbf{w}}{1-a_0}}\boldsymbol{\Sigma}_{t-1}^{1/2}\mathbf{e}_j$
      $a_j = (1-a_0)/(2D_w)$
    end
    `// Pass through measurement eqn and compute predicted mean and covariance`
    $\boldsymbol{\mu}_+ = \sum_{j=0}^{2D_\mathbf{w}} a_j \mathbf{f}[\hat{\mathbf{w}}^{[j]}]$
    $\boldsymbol{\Sigma}_+ = \sum_{j=0}^{2D_\mathbf{w}} a_j (\mathbf{f}[\hat{\mathbf{w}}^{[j]}] - \boldsymbol{\mu}_+)(\mathbf{f}[\hat{\mathbf{w}}^{[j]}] - \boldsymbol{\mu}_+)^T + \boldsymbol{\Sigma}_p$
    `// Approximate predicted state with particles`
    $\hat{\mathbf{w}}^{[0]} = \boldsymbol{\mu}_+$
    for j=*1* **to** D_w **do**
      $\hat{\mathbf{w}}^{[j]} = \boldsymbol{\mu}_+ + \sqrt{\frac{D_\mathbf{w}}{1-a_0}}\boldsymbol{\Sigma}_+^{1/2}\mathbf{e}_j$
      $\hat{\mathbf{w}}^{[D_\mathbf{w}+j]} = \boldsymbol{\mu}_+ - \sqrt{\frac{D_\mathbf{w}}{1-a_0}}\boldsymbol{\Sigma}_+^{1/2}\mathbf{e}_j$
    end
    `// Pass through measurement equation`
    for j=*0* **to** $2D_w$ **do**
      $\hat{\mathbf{x}}^{[j]} = \mathbf{g}[\hat{\mathbf{w}}^{[j]}]$
    end
    `// Compute predicted measurement state and covariance`
    $\boldsymbol{\mu}_x = \sum_{j=0}^{2D_\mathbf{w}} a_j \hat{\mathbf{x}}^{[j]}$
    $\boldsymbol{\Sigma}_x = \sum_{j=0}^{2D_\mathbf{w}} a_j (\hat{\mathbf{x}}^{[j]} - \boldsymbol{\mu}_x)(\hat{\mathbf{x}}^{[j]} - \boldsymbol{\mu}_x)^T + \boldsymbol{\Sigma}_m$
    `// Compute new world state and covariance`
    $\mathbf{K} = \left(\sum_{j=0}^{2D_\mathbf{w}} a_j (\hat{\mathbf{w}}^{[j]} - \boldsymbol{\mu}_+)^T (\hat{\mathbf{x}}^{[j]} - \boldsymbol{\mu}_x)^T\right) \boldsymbol{\Sigma}_x^{-1}$
    $\boldsymbol{\mu}_t = \boldsymbol{\mu}_+ + \mathbf{K}(\mathbf{x}_t - \boldsymbol{\mu}_x)$
    $\boldsymbol{\Sigma}_t = \boldsymbol{\Sigma}_+ - \mathbf{K}\boldsymbol{\Sigma}_x\mathbf{K}^T$
  end
end

Algorithm 19.6: Condensation algorithm

The condensation algorithm completely does away with the Gaussian representation and represents the distributions entirely as sets of weighted particles, where each particle can be interpreted as a hypothesis about the world state and the weight as the probability of this hypothesis being true.

Algorithm 19.6: The condensation algorithm

Input : Measurements $\{\mathbf{x}\}_{t=1}^{T}$, temporal model $Pr(\mathbf{w}_t|\mathbf{w}_{t-1})$, measurement model $Pr(\mathbf{x}_t|\mathbf{w}_t)$
Output: Weights $\{a_t^{[j]}\}_{t=1}^{T}$, hypotheses $\{w_t^{[j]}\}_{t=1}^{T}$
begin
  `// Initialise weights to equal`
  $\mathbf{a}_0 = [1/J, 1/J, \ldots, 1/J]$
  `// Initialize hypotheses to plausible values for state`
  for j=*1* **to** J **do**
    $\mathbf{w}_0^{[j]} = \text{Initialize}[\,]$
  end
  `// For each time step`
  for t=*1* **to** T **do**
    `// For each particle`
    for j=*1* **to** J **do**
      `// Sample from` $1 \ldots J$ `according to probabilities` $a_{t-1}^{[1]} \ldots a_{t-1}^{[J]}$
      $n = \text{sampleFromCategorical}[\mathbf{a}_{t-1}]$
      `// Draw sample from temporal update model`
      $\hat{\mathbf{w}}_t^{[j]} = \text{sample}[Pr(\mathbf{w}_t|\mathbf{w}_{t-1} = \hat{\mathbf{w}}_{t-1}^{[n]})]$
      `// Set weight for particle according to measurement model`
      $a_t^{[j]} = Pr(\mathbf{x}_t|\hat{\mathbf{w}}_t^{[j]})$
    end
    `// Normalise weights`
    $\mathbf{a}_t = \mathbf{a}_t / (\sum_{j=1}^{J} a_t^{[j]})$
  end
end

Algorithm 20.1: Bag of features model

The bag of features model treats each object class as a distribution over discrete features f regardless of their position in the image. Assume that there are I images with J_i features in the ith image. Denote the jth feature in the ith image as f_{ij}. Then we have

$$Pr(\mathcal{X}_i|w=n) = \prod_{j=1}^{I_j} \text{Cat}_{f_{ij}}[\boldsymbol{\lambda}_n]$$

Algorithm 20.1: Learn bag of words model

Input : Features $\{\mathbf{f}_{ij}\}_{i=1,j=1}^{I,J_i}$, $\{w_i\}_{i=1}^{I}$, Dirichlet parameter α
Output: Model parameters $\{\boldsymbol{\lambda}_m\}_{m=1}^{M}$
begin
  `// For each object class`
  for n=*1* **to** N **do**
    `// For each feature`
    for k=*1* **to** L **do**
      `// Compute number of times feature` k `observed for object` m
      $N_{nk}^f = \sum_{i=1}^{I}\sum_{j=1}^{J_i}\delta[w_i-n]\delta[f_{ij}-k]$
    end
    `// Compute parameter`
    $\lambda_{nk} = (N_{nk}^f + \alpha - 1)/(\sum_{k=1}^{K} N_{nk}^f + K\alpha - 1)$
  end
end

Algorithm 20.2: Latent Dirichlet Allocation

The latent Dirichlet allocation model models a discrete set of features $f_{ij} \in 1 \ldots K$ as a mixture of M categorical distributions (parts), where the categorical distributions themselves are shared, but the mixture weights $\boldsymbol{\pi}_i$ differ from image to image

Algorithm 20.2: Learn latent Dirichlet allocation model

Input : Features $\{\mathbf{f}_{ij}\}_{i=1,j=1}^{I,J_i}$, $\{w_i\}_{i=1}^{I}$, Dirichlet parameters α, β
Output: Model parameters $\{\boldsymbol{\lambda}_m\}_{m=1}^{M}, \{\boldsymbol{\pi}_i\}_{i=1}^{I}$
begin
  `// Initialize categorical parameters`
  $\boldsymbol{\theta} = \boldsymbol{\theta}_0$ [a]
  `// Initialize count parameters`
  $\mathbf{N}^{(f)} = 0$
  $\mathbf{N}^{(p)} = 0$
  for i=*1* **to** I **do**
    for j=*1* **to** J **do**
      `// Initialize hidden variables`
      $p_{ij} = \text{randInt}[M]$
      `// Update count parameters`
      $N^{(f)}_{p_{ij},f_{ij}} = N^{(f)}_{p_{ij},f_{ij}} + 1$
      $N^{(p)}_{i,p_{ij}} = N^{(f)}_{i,p_{ij}} + 1$
    end
  end
  `// Main MCMC Loop`
  for t=*1* **to** T **do**
    $\mathbf{p}^{(t)} = \text{MCMCSample}[\mathbf{p}, \mathbf{f}, \mathbf{N}^{(f)}, \mathbf{N}^{(w)}, \{\boldsymbol{\lambda}_m\}_{m=1}^{M}, \{\boldsymbol{\pi}_i\}_{i=1}^{I}, M, K]$
  end
  `// Choose samples to use for parameter estimate`
  $\mathcal{S}_t = [\text{BurnInTime} : \text{SkipTime} : \text{Last Sample}]$
  for i=*1* **to** I **do**
    for m=*1* **to** M **do**
      $\pi_{i,m} = \sum_{j=1}^{J_i} \sum_{t \in \mathcal{S}_t} \delta[p_{ij}^{[t]} - m] + \alpha$
    end
    $\boldsymbol{\pi}_i = \boldsymbol{\pi}_i / \sum_{m=1}^{M} \pi_{im}$
  end
  for m=*1* **to** M **do**
    for k=*1* **to** K **do**
      $\lambda_{m,k} = \sum_{i=1}^{I} \sum_{j=1}^{J_i} \sum_{t \in \mathcal{S}_t} \delta[p_{ij}^{[t]} - m]\delta[f_{ij} - k] + \beta$
    end
    $\boldsymbol{\lambda}_m = \boldsymbol{\lambda}_m / \sum_{k=1}^{K} \lambda_{m,k}$
  end
end

[a] One way to do this would be to set the categorical parameters $\{\boldsymbol{\lambda}_m\}_{m=1}^{M}, \{\boldsymbol{\pi}_i\}_{i=1}^{I}$ to random values by generating positive random vectors and normalizing them to sum to one.

Algorithm 20.2b: Gibbs' sampling for LDA

The preceding algorithm relies on Gibbs sampling from the posterior distribution over the part labels. This can be achieved efficiently using the following method.

Algorithm 20.2b: MCMC Sampling for LDA

Input : $\mathbf{p}, \mathbf{f}, \mathbf{N}^{(f)}, \mathbf{N}^{(w)}, \{\boldsymbol{\lambda}_m\}_{m=1}^{M}, \{\boldsymbol{\pi}_i\}_{i=1}^{I}, M, K$
Output: Part sample $\mathbf{p}$
begin
  repeat
    `// Choose next feature`
    $(a, b) = \text{ChooseFeature}[J_1, J_2, \ldots J_I]$
    `// Remove feature from statistics`
    $N^{(f)}_{p_{ab}, f_{ab}} = N^{(f)}_{p_{ab}, f_{ab}} - 1$
    $N^{(p)}_{a, p_{ab}} = N^{(p)}_{p_{ab}} - 1$
    for m=*1* **to** M **do**
      $q_m = (N^{(f)}_{m, f_{ab}} + \beta)(N^{(p)}_{a,m} + \alpha)$
      $q_m = q_m / (\sum_{k=1}^{K}(N^{(f)}_{m,k} + \beta) \sum_{m=1}^{N}(N^{(p)}_{a,m} + \alpha))$
    end
    `// Normalize`
    $\mathbf{q} = \mathbf{q} / (\sum_{m=1}^{M} q_m)$
    `// Draw new feature`
    $p_{ij} = \text{DrawCategorical}[\mathbf{q}]$
    `// Replace feature in statistics`
    $N^{(f)}_{p_{ab}, f_{ab}} = N^{(f)}_{p_{ab}, f_{ab}} + 1$
    $N^{(p)}_{a, p_{ab}} = N^{(p)}_{p_{ab}} + 1$
  until *All parts p_{ij} updated*
end