1
Factor Graphs for SLAM

Frank Dellaert, Michael Kaess, and Timothy Barfoot

In this chapter we introduce factor graphs and establish the connection with maximum a posteriori (MAP) inference and least-squares for the case of Gaussian priors and Gaussian measurement noise. We focus on the SLAM back-end, after measurements have been extracted by the front-end and data association has been accomplished. We discuss both linear and nonlinear optimization methods for the corresponding least-squares problems, and then make the connection between sparsity, factor graphs, and Bayes nets more explicit. Finally, we apply this to develop the Bayes tree and the incremental smoothing and mapping (iSAM) algorithm.

Historical Note

A *smoothing* approach to SLAM involves not just the most current robot location, but the entire robot trajectory up to the current time. A number of authors consider the problem of smoothing the robot trajectory only [178, 707, 708, 427, 602, 320], now known as *PoseSLAM*. This is particularly suited to sensors such as laser-range finders that yield pairwise constraints between nearby robot poses.

More generally, one can consider the *full SLAM problem* [1092], i.e., the problem of optimally estimating the entire set of sensor poses along with the parameters of all features in the environment. This led to a flurry of work between 2000 and 2005 where these ideas were applied in the context of SLAM [295, 348, 347, 1092]. From a computational view, this optimization-based smoothing was recognized as beneficial since (a) in contrast to the filtering-based covariance or information matrices, which *both* become fully dense over time [855, 1091], the information matrix associated with smoothing is and stays sparse; (b) in typical mapping scenarios (i.e., not repeatedly traversing a small environment) this matrix is a much more compact representation of the map covariance structure.

Square-root Segment Anything Model (SAM), also known as the ‘factor-graph approach’, was introduced in [258, 262] based on the fact that the information matrix or measurement Jacobian can be efficiently factorized using sparse Cholesky or QR factorization, respectively. This yields a square-root information matrix that can be used to immediately obtain the optimal robot trajectory and map. Factoring the information matrix is known in the sequential estimation literature as *square-root information filtering (SRIF)*, and was developed in 1969 for use in JPL's Mariner 10 missions to Venus [90]. The use of square roots results in more accurate and stable algorithms, and, quoting Maybeck [749], "a number of practitioners have argued, with considerable logic, that square root filters should *always* be adopted in preference to the standard Kalman filter recursion".

Below we discuss in detail how factor graphs are a natural representation for the sparsity inherent in SLAM problems, how (sparse) matrix factorization into a matrix-square root is at the heart of solving these problems, and finally how all this relates to the much more general *variable elimination algorithm.* Much of this chapter is an abridged version of a longer article by Dellaert et al. [263].

1.1 Visualizing SLAM With Factor Graphs

In this section we introduce factor graphs as a way of intuitively visualizing the sparse nature of the SLAM problem by first considering a toy example and its factor graph representation. We then show how many different flavors of SLAM can be represented as such, and how even in larger problems the sparse nature of many sparse problems is immediately apparent.

1.1.1 A Toy Example

We begin by examining a simple SLAM scenario to illustrate how factor graphs are constructed. Figure 1.1 shows a simple toy example illustrating the structure of the problem graphically. A robot moving across three successive poses $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_3$ makes bearing observations on two landmarks $\boldsymbol{\ell}_1$ and $\boldsymbol{\ell}_2$. To anchor the solution in space, let us also assume there is an absolute position/orientation measurement on the first pose $\boldsymbol{p}_1$. Without this there would be no information about absolute position, as bearing measurements are all relative.[1]

Because of measurement uncertainty, we cannot hope to recover the true state of the world, but we can obtain a probabilistic description of what can be inferred from the measurements. In the Bayesian probability framework, we use the language of probability theory to assign a subjective degree of belief to uncertain events. We do this using *probability density functions (PDFs)* $p(\boldsymbol{x})$ over the unknown variables $\boldsymbol{x}$. PDFs are non-negative functions satisfying

$$\int p(\boldsymbol{x})\, d\boldsymbol{x} = 1, \tag{1.1}$$

which is the axiom of total probability. In the simple example of Figure 1.1, the

[1] Handling rotations properly is a bit more involved than our treatment in this first chapter lets on. However, the next chapter will put us on a proper footing for such quantities. For now, we will assume they are regular vector quantities and delay discussion of their subtleties.

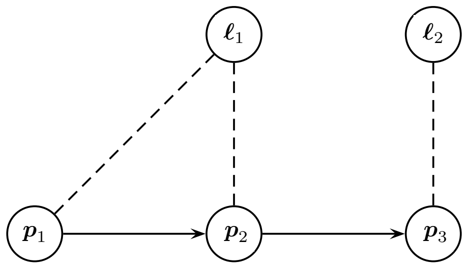


Figure 1.1 A toy simultaneous localization and mapping (SLAM) example with three robot poses and two landmarks. Above we schematically indicate the robot motion with arrows, while the dotted lines indicate bearing measurements.

state, $\boldsymbol{x}$, is

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \boldsymbol{p}_3 \\ \boldsymbol{\ell}_1 \\ \boldsymbol{\ell}_2 \end{bmatrix}, \tag{1.2}$$

which is just a stacking of the individual unknowns.

In SLAM we want to characterize our knowledge about the unknowns $\boldsymbol{x}$, in this case robot poses and the unknown landmark positions, when given a set of *observed* measurements $\boldsymbol{z}$. Using the language of Bayesian probability, this is simply the conditional density

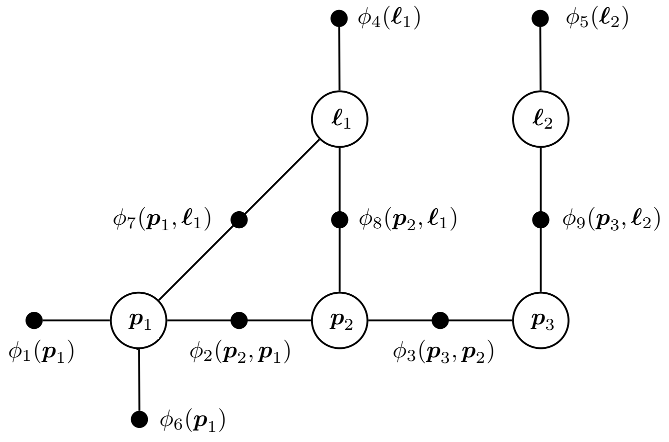
$$p(\boldsymbol{x}|\boldsymbol{z}), \tag{1.3}$$

and obtaining a description like this is called *probabilistic inference*. A prerequisite is to first specify a probabilistic model for the variables of interest and how they give rise to (uncertain) measurements. This is where *probabilistic graphical models* enter the picture.

Probabilistic graphical models provide a mechanism to compactly describe complex probability densities by exploiting the structure in them [600]. In particular, high-dimensional probability densities can often be factorized as a product of many *factors*, each of which is a probability density over a much smaller domain.

1.1.2 A Factor-Graph View

Factor graphs are probabilistic graphical models and they allow us to specify a joint density as a product of factors. However, they are more general in that they can be used to specify *any* factored function $\phi(\boldsymbol{x})$ over a set of variables $\boldsymbol{x}$, not just probability densities.



$\phi_4(\boldsymbol{\ell}_1)$
$\phi_5(\boldsymbol{\ell}_2)$
$\boldsymbol{\ell}_1$
$\boldsymbol{\ell}_2$
$\phi_7(\boldsymbol{p}_1, \boldsymbol{\ell}_1)$
$\phi_8(\boldsymbol{p}_2, \boldsymbol{\ell}_1)$
$\phi_9(\boldsymbol{p}_3, \boldsymbol{\ell}_2)$
$\boldsymbol{p}_1$
$\boldsymbol{p}_2$
$\boldsymbol{p}_3$
$\phi_1(\boldsymbol{p}_1)$
$\phi_2(\boldsymbol{p}_2, \boldsymbol{p}_1)$
$\phi_3(\boldsymbol{p}_3, \boldsymbol{p}_2)$
$\phi_6(\boldsymbol{p}_1)$

Figure 1.2 Factor graph resulting from the example in Figure 1.1.

To motivate this, consider performing inference for the toy SLAM example. The posterior $p(\boldsymbol{x}|\boldsymbol{z})$ can be re-written using Bayes' law, $p(\boldsymbol{x}|\boldsymbol{z}) \propto p(\boldsymbol{z}|\boldsymbol{x})p(\boldsymbol{x})$, as

$$\begin{aligned} p(\boldsymbol{x}|\boldsymbol{z}) \propto\, & p(\boldsymbol{p}_1)\, p(\boldsymbol{p}_2|\boldsymbol{p}_1)\, p(\boldsymbol{p}_3|\boldsymbol{p}_2) && \text{(1.4a)} \\ & \times p(\boldsymbol{\ell}_1)\, p(\boldsymbol{\ell}_2) && \text{(1.4b)} \\ & \times p(\boldsymbol{z}_1|\boldsymbol{p}_1) && \text{(1.4c)} \\ & \times p(\boldsymbol{z}_2|\boldsymbol{p}_1, \boldsymbol{\ell}_1)\, p(\boldsymbol{z}_3|\boldsymbol{p}_2, \boldsymbol{\ell}_1)\, p(\boldsymbol{z}_4|\boldsymbol{p}_3, \boldsymbol{\ell}_2). && \text{(1.4d)} \end{aligned}$$

where we assumed a typical Markov chain generative model for the pose trajectory. Each of the factors represents one piece of information about the unknowns, $\boldsymbol{x}$.

To visualize this factorization, we use a *factor graph.* Figure 1.2 introduces the corresponding factor graph by example: all unknown states $\boldsymbol{x}$, both poses and landmarks, have a node associated with them. Measurements are *not* represented explicitly as they are known, and hence not of interest. In factor graphs we explicitly introduce an additional node type to represent every *factor* in the posterior $p(\boldsymbol{x}|\boldsymbol{z})$. In the figure, each small black node represents a factor, and—importantly—is connected to only those state variables of which it is a function. For example, the factor $\phi_9(\boldsymbol{p}_3, \boldsymbol{\ell}_2)$ is connected only to the variable nodes $\boldsymbol{p}_3$ and $\boldsymbol{\ell}_2$. In more detail, we have

$$\begin{aligned} \phi(\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3, \boldsymbol{\ell}_1, \boldsymbol{\ell}_2) =\, & \phi_1(\boldsymbol{p}_1)\, \phi_2(\boldsymbol{p}_2, \boldsymbol{p}_1)\, \phi_3(\boldsymbol{p}_3, \boldsymbol{p}_2) && \text{(1.5a)} \\ & \times \phi_4(\boldsymbol{\ell}_1)\, \phi_5(\boldsymbol{\ell}_2) && \text{(1.5b)} \\ & \times \phi_6(\boldsymbol{p}_1) && \text{(1.5c)} \\ & \times \phi_7(\boldsymbol{p}_1, \boldsymbol{\ell}_1)\, \phi_8(\boldsymbol{p}_2, \boldsymbol{\ell}_1)\, \phi_9(\boldsymbol{p}_3, \boldsymbol{\ell}_2), && \text{(1.5d)} \end{aligned}$$

where the correspondence between the factors and the original probability densities in (1.4a)-(1.4d) should be obvious.

The factor values need only be *proportional* to the corresponding probability densities: any normalization constants that do not depend on the state variables may be omitted without consequence. Also, in this example, all factors above came either from a prior, e.g., $\phi_1(\boldsymbol{p}_1) \propto p(\boldsymbol{p}_1)$ or from a measurement, e.g., $\phi_9(\boldsymbol{p}_3, \boldsymbol{\ell}_2) \propto p(\boldsymbol{z}_4|\boldsymbol{p}_3, \boldsymbol{\ell}_2)$. Although the measurement variables $\boldsymbol{z}_1..\boldsymbol{z}_4$ are not explicitly shown in the factor graph, those factors are implicitly conditioned on them. Sometimes, when it helps to make this more explicit, factors can be written as (for example) $\phi_9(\boldsymbol{p}_3, \boldsymbol{\ell}_2; \boldsymbol{z}_4)$ or even $\phi_{\boldsymbol{z}_4}(\boldsymbol{p}_3, \boldsymbol{\ell}_2)$.

1.1.3 Factor Graphs as a Language

In addition to providing a formal basis for inference, factor graphs help visualize SLAM problems of many different flavors, give insight into the structure of the problem, and serve as a *lingua franca* that can help practitioners align across team boundaries. Each factor in a factor graph, such as those in Fig 2.2, can be thought of as an equation involving the variables it is connected to. There are typically many more equations than unknowns, which is why we need to quantify the uncertainty in both prior information and measurements. This will lead to a least-squares formulation, appropriately fusing the information from multiple sources.

Many different flavors of the SLAM problem are all easily represented as factor graphs. Figure 1.1 is an example of *landmark-based SLAM* because it involves both pose and landmark variables. Figure 1.3 illustrates several other variants including *bundle adjustment (BA)* (same as landmark-based SLAM but without motion model), *pose-graph optimization (PGO)* (no landmark variables but includes loop closures), and *simultaneous trajectory estimation and mapping (STEAM)* (poses are augmented to include derivatives such as velocity).

The factor graph for a more realistic landmark-based SLAM problem than the toy example could look something like Figure 1.4. This graph was created by simulating a 2D robot, moving in the plane for about 100 time steps, as it observes landmarks. For visualization purposes, each robot pose and landmark is rendered at its ground-truth position in 2D. With this, we see that the odometry factors form a prominent, chain-like backbone, whereas off to the sides binary likelihood factors are connected to the 20 or so landmarks. All factors in such SLAM problems are typically nonlinear, except for priors.

Examining the factor graph reveals a great deal of structure by which we can gain insight into a particular instance of the SLAM problem. First, there are landmarks with a great deal of measurements, which we expect to be pinned down very well. Others have only a tenuous connection to the graph, and hence we expect them to be less well determined. For example, the lone landmark near the bottom right has only a single measurement associated with it: if this is a bearing-only measurement,

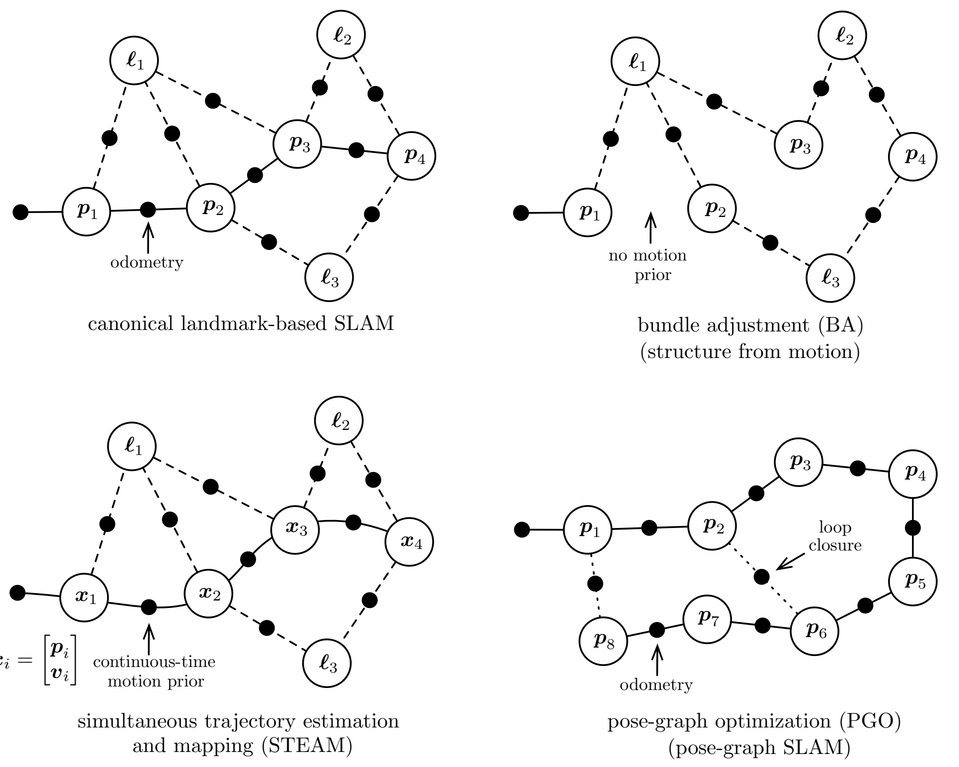


Figure 1.3 A few variants of SLAM problems that can all be viewed through the factor-graph lens. Canonical landmark-based SLAM has both pose and landmark variables; landmarks are measured from poses and there is some motion prior between poses typically based on odometry. BA is the same but without the motion prior. STEAM is similar but now poses can be replaced by higher-order states and a smooth continuous-time motion prior is used. PGO does not have landmark variables but enjoys extra loop-closure measurements between poses.

many assignments of a 2D location to the landmark will be equally 'correct'. This is the same as saying that we have infinite uncertainty in some subset of the domain of the unknowns, which is where prior knowledge should come to the rescue.

1.2 From MAP Inference to Least Squares

In SLAM, *maximum a posteriori (MAP)* inference is the process of determining the values for the unknowns $\boldsymbol{x}$ that maximally agree with the information present in the uncertain measurements. In real life we are *not* given the ground-truth locations for the landmarks, nor the time-varying pose of the robot, although in many practical cases we might have a good initial estimate. Below we review how to model both prior knowledge and measurements using probability densities, how the posterior

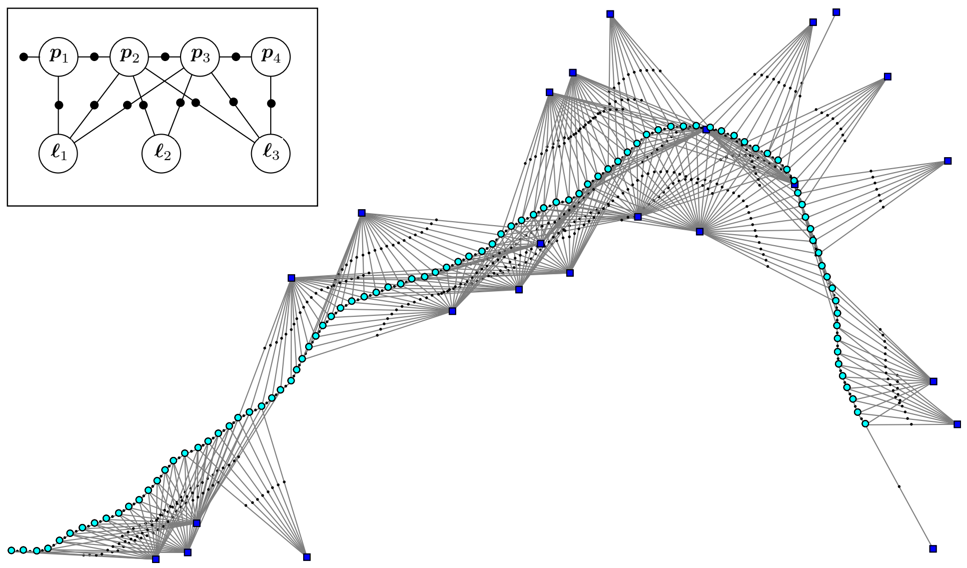


Figure 1.4 Factor graph for a larger, simulated SLAM example.

density given measurements is most conveniently represented as a factor graph, and how given Gaussian priors and Gaussian noise models the corresponding optimization problem is nothing but the familiar nonlinear least-squares problem.

1.2.1 Factor Graphs for MAP Inference

We are interested in the *unknown state variables* $\boldsymbol{x}$, such as poses and/or landmarks, *given* the measurements $\boldsymbol{z}$. The most-often-used *estimator* for these unknown state variables $\boldsymbol{x}$ is the *maximum a posteriori (MAP)* estimate, so named because it maximizes the posterior density $p(\boldsymbol{x}|\boldsymbol{z})$ of the states $\boldsymbol{x}$ given the measurements $\boldsymbol{z}$:

$$\begin{aligned} \boldsymbol{x}^{\text{MAP}} &= \arg\max_{\boldsymbol{x}} p(\boldsymbol{x}|\boldsymbol{z}) && \text{(1.6a)} \\ &= \arg\max_{\boldsymbol{x}} \frac{p(\boldsymbol{z}|\boldsymbol{x})p(\boldsymbol{x})}{p(\boldsymbol{z})} && \text{(1.6b)} \\ &= \arg\max_{\boldsymbol{x}} p(\boldsymbol{z}|\boldsymbol{x})p(\boldsymbol{x}) && \text{(1.6c)} \end{aligned}$$

The second equation above is Bayes' law, and expresses the posterior as the product of the measurement density $p(\boldsymbol{z}|\boldsymbol{x})$ and the prior $p(\boldsymbol{x})$ over the states, appropriately normalized by the factor $p(\boldsymbol{z})$. The third equation drops the $p(\boldsymbol{z})$ since this does not depend on the $\boldsymbol{x}$ and therefore will not impact the $\arg\max$ operation.

We use factor graphs to express the unnormalized posterior $p(\boldsymbol{z}|\boldsymbol{x})p(\boldsymbol{x})$. Formally a factor graph is a bipartite graph $F = (\mathcal{U}, \mathcal{V}, \mathcal{E})$ with two types of nodes: *factors*

$\phi_i \in \mathcal{U}$ and *variables* $\boldsymbol{x}_j \in \mathcal{V}$. Edges $e_{ij} \in \mathcal{E}$ are always between factor nodes and variables nodes. The set of variable nodes adjacent to a factor ϕ_i is written as $\mathcal{X}(\phi_i)$, and we write $\boldsymbol{x}_i$ for an assignment to this set. With these definitions, a factor graph F defines the factorization of a global function $\phi(\boldsymbol{x})$ as

$$\phi(\boldsymbol{x}) = \prod_i \phi_i(\boldsymbol{x}_i). \tag{1.7}$$

In other words, the independence relationships are encoded by the edges e_{ij} of the factor graph, with each factor ϕ_i a function of *only* the variables $\boldsymbol{x}_i$ in its adjacency set $\mathcal{X}(\phi_i)$.

In the rest of this chapter, we show how to find an optimal assignment, the MAP estimate, through optimization over the unknown variables in the factor graph. Indeed, for an arbitrary factor graph, MAP inference comes down to maximizing the product (1.7) of all factor-graph potentials:

$$\boldsymbol{x}^{\text{MAP}} = \arg\max_{\boldsymbol{x}} \phi(\boldsymbol{x}) \tag{1.8a}$$

$$= \arg\max_{\boldsymbol{x}} \prod_i \phi_i(\boldsymbol{x}_i). \tag{1.8b}$$

What is left now is to derive the exact form of the factors $\phi_i(\boldsymbol{x}_i)$, which depends very much on how we model the measurement models $p(\boldsymbol{z}|\boldsymbol{x})$ and the prior densities $p(\boldsymbol{x})$. We discuss this in detail next.

1.2.2 Specifying Probability Densities

The exact form of the densities $p(\boldsymbol{z}|\boldsymbol{x})$ and $p(\boldsymbol{x})$ above depends very much on the application and the sensors used. The most often used densities involve the *multivariate Gaussian density*, with probability density

$$\mathcal{N}(\boldsymbol{\theta}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{|2\pi\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}\|\boldsymbol{\theta} - \boldsymbol{\mu}\|^2_{\boldsymbol{\Sigma}}\right), \tag{1.9}$$

where $\boldsymbol{\mu} \in \mathbb{R}^n$ is the mean, $\boldsymbol{\Sigma}$ is an $n \times n$ covariance matrix, and

$$\|\boldsymbol{\theta} - \boldsymbol{\mu}\|^2_{\boldsymbol{\Sigma}} \triangleq (\boldsymbol{\theta} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\boldsymbol{\theta} - \boldsymbol{\mu}) \tag{1.10}$$

denotes the squared Mahalanobis distance. The normalization constant $\sqrt{|2\pi\boldsymbol{\Sigma}|} = (2\pi)^{n/2} |\boldsymbol{\Sigma}|^{1/2}$, where $|.|$ denotes the matrix determinant, ensures the multivariate Gaussian density integrates to 1.0 over its domain.

Priors on unknown quantities are often specified using a Gaussian density, and in many cases it is both justified and convenient to model measurements as corrupted by zero-mean Gaussian noise. For example, a bearing measurement[2] from a given

[2] As a reminder, there are some subtleties associated with rotational state variables that we will discuss more thoroughly in the next chapter.

pose $\boldsymbol{p}$ to a given landmark $\boldsymbol{\ell}$ would be modeled as

$$\boldsymbol{z} = \boldsymbol{h}(\boldsymbol{p}, \boldsymbol{\ell}) + \boldsymbol{\eta}, \tag{1.11}$$

where $\boldsymbol{h}(\cdot)$ is a *measurement prediction function*, and the noise $\boldsymbol{\eta}$ is drawn from a zero-mean Gaussian density with measurement covariance $\boldsymbol{\Sigma_R}$. This yields the following conditional density $p(\boldsymbol{z}|\boldsymbol{p}, \boldsymbol{\ell})$ on the measurement $\boldsymbol{z}$:

$$p(\boldsymbol{z}|\boldsymbol{p}, \boldsymbol{\ell}) = \mathcal{N}(\boldsymbol{z}; \boldsymbol{h}(\boldsymbol{p}, \boldsymbol{\ell}), \boldsymbol{\Sigma_R}) = \frac{1}{\sqrt{|2\pi\boldsymbol{\Sigma_R}|}} \exp\left(-\frac{1}{2}\left\|\boldsymbol{z} - \boldsymbol{h}(\boldsymbol{p}, \boldsymbol{\ell})\right\|^2_{\boldsymbol{\Sigma_R}}\right). \tag{1.12}$$

The measurement functions $\boldsymbol{h}(\cdot)$ are often nonlinear in practical robotics applications. Still, while they depend on the sensor used and the SLAM front-end, they are typically not difficult to reason about or write down. The measurement function for a 2D bearing measurement is simply

$$\boldsymbol{h}(\boldsymbol{p}, \boldsymbol{\ell}) = \operatorname{atan2}(\ell_y - p_y, \ell_x - p_x), \tag{1.13}$$

where atan2 is the well-known two-argument arctangent variant. Hence, the final *probabilistic measurement model* $p(\boldsymbol{z}|\boldsymbol{p}, \boldsymbol{\ell})$ is obtained as

$$p(\boldsymbol{z}|\boldsymbol{p}, \boldsymbol{\ell}) = \frac{1}{\sqrt{|2\pi\boldsymbol{\Sigma_R}|}} \exp\left(-\frac{1}{2}\left\|\boldsymbol{z} - \operatorname{atan2}(\ell_y - p_y, \ell_x - p_x)\right\|^2_{\boldsymbol{\Sigma_R}}\right). \tag{1.14}$$

Note that we will not *always* assume Gaussian measurement noise: to cope with the occasional data association mistake, for example, many authors have proposed the use of robust measurement densities, with heavier tails than a Gaussian density; these are discussed in Chapter 3.

Not all probability densities involved are derived from measurements. For example, in the toy SLAM problem the prior $p(\boldsymbol{x})$ on the trajectory is made up of a prior $p(\boldsymbol{p}_1)$ and conditional densities $p(\boldsymbol{p}_{t+1}|\boldsymbol{p}_t)$, specifying a *probabilistic motion model* that the robot is assumed to obey given known control inputs $\boldsymbol{u}_t$. In practice, we often use a conditional Gaussian assumption,

$$p(\boldsymbol{p}_{t+1}|\boldsymbol{p}_t, \boldsymbol{u}_t) = \frac{1}{\sqrt{|2\pi\boldsymbol{\Sigma_Q}|}} \exp\left(-\frac{1}{2}\left\|\boldsymbol{p}_{t+1} - \boldsymbol{g}(\boldsymbol{p}_t, \boldsymbol{u}_t)\right\|^2_{\boldsymbol{\Sigma_Q}}\right), \tag{1.15}$$

where $\boldsymbol{g}(\cdot)$ is a motion model, and $\boldsymbol{\Sigma_Q}$ a covariance matrix of the appropriate dimensionality, e.g., 3×3 in the case of robots operating in the plane.

Often we have no known control inputs $\boldsymbol{u}_t$ but instead we *measure* how the robot moved, e.g., via an odometry measurement $\boldsymbol{o}_t$. For example, if we assume the odometry simply measures the difference between poses, subject to Gaussian noise with covariance $\boldsymbol{\Sigma_S}$, we obtain

$$p(\boldsymbol{o}_t|\boldsymbol{p}_{t+1}, \boldsymbol{p}_t) = \frac{1}{\sqrt{|2\pi\boldsymbol{\Sigma_S}|}} \exp\left(-\frac{1}{2}\left\|\boldsymbol{o}_t - (\boldsymbol{p}_{t+1} - \boldsymbol{p}_t)\right\|^2_{\boldsymbol{\Sigma_S}}\right). \tag{1.16}$$

If we have *both* known control inputs $\boldsymbol{u}_t$ and odometry measurements $\boldsymbol{o}_t$ we can combine (1.15) and (1.16).

Note that for robots operating in three-dimensional space, we will need slightly more sophisticated machinery to specify densities on nonlinear manifolds such as SE(3), as discussed in the next chapter.

1.2.3 Nonlinear Least Squares

We now show that MAP inference for SLAM problems with Gaussian noise models as above is equivalent to solving a nonlinear least-squares problem. If we assume that all factors are of the form

$$\phi_i(\boldsymbol{x}_i) \propto \exp\left(-\frac{1}{2}\left\|\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i)\right\|^2_{\boldsymbol{\Sigma}_i}\right), \tag{1.17}$$

which include both simple Gaussian priors and likelihood factors derived from measurements corrupted by zero-mean, normally distributed noise. Taking the negative log of (1.8b) and dropping the factor $\frac{1}{2}$ allows us to instead minimize a sum of *nonlinear least-squares* terms:

$$\boldsymbol{x}^{\text{MAP}} = \underset{\boldsymbol{x}}{\arg\min} \sum_i \left\|\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i)\right\|^2_{\boldsymbol{\Sigma}_i}. \tag{1.18}$$

Minimizing this objective function performs sensor fusion through the process of combining several measurement-derived factors, and possibly several priors, to uniquely determine the MAP solution for the unknowns.

An important and non-obvious observation is that the factors in (1.18) typically represent rather *under-specified* densities on the involved unknown variables $\boldsymbol{x}_i$. Indeed, except for simple prior factors, the measurements $\boldsymbol{z}_i$ are typically of lower dimension than the unknowns $\boldsymbol{x}_i$. In those cases, the factor by itself accords the same likelihood to an infinite subset of the domain of $\boldsymbol{x}_i$. For example, a 2D measurement in a camera image is consistent with an entire ray of 3D points that project to the same image location.

Even though the functions $\boldsymbol{h}_i$ are nonlinear, *if* we have a decent initial guess available, then the nonlinear optimization methods we discuss in this chapter will be able to converge to the global minimum of (1.18). We should caution, however, that as our objective in (1.18) is *non-convex*, there is no guarantee that we will not get stuck in a local minimum if our initial guess is poor. This has led to so-called *certifiably optimal* solvers, which are the subject of a later chapter. Below, however, we focus on local methods rather than global solvers. We start off below by considering the the easier problem of solving a *linearized* version of the problem.

1.3 Solving Linear Least Squares

Before tackling the more difficult problem of nonlinear least squares, in this section we first show to *linearize* the problem, show how this leads to a *linear* least squares problem, and review matrix factorization as computationally efficient way to solve the corresponding *normal equations*. A seminal reference for these methods is the book by Golub and Loan [397].

1.3.1 Linearization

We can linearize all measurement functions $\boldsymbol{h}_i(\cdot)$ in the nonlinear least-squares objective function (1.18) using a simple Taylor expansion,

$$\boldsymbol{h}_i(\boldsymbol{x}_i) = \boldsymbol{h}_i(\boldsymbol{x}_i^0 + \boldsymbol{\delta}_i) \approx \boldsymbol{h}_i(\boldsymbol{x}_i^0) + \boldsymbol{H}_i\boldsymbol{\delta}_i, \tag{1.19}$$

where the *measurement Jacobian* $\boldsymbol{H}_i$ is defined as the (multivariate) partial derivative of $\boldsymbol{h}_i(\cdot)$ at a given linearization point $\boldsymbol{x}_i^0$,

$$\boldsymbol{H}_i \triangleq \left.\frac{\partial \boldsymbol{h}_i(\boldsymbol{x}_i)}{\partial \boldsymbol{x}_i}\right|_{\boldsymbol{x}_i^0}, \tag{1.20}$$

and $\boldsymbol{\delta}_i \triangleq \boldsymbol{x}_i - \boldsymbol{x}_i^0$ is the *state update vector*. Note that we make an assumption that $\boldsymbol{x}_i$ lives in a vector space or, equivalently, can be represented by a *vector*. This is not always the case, e.g., when some of the unknown states in $\boldsymbol{x}$ represent 3D rotations or other more complex manifold types. We will revisit this issue in Chapter 2.

Substituting the Taylor expansion (1.19) into the nonlinear least-squares expression (1.18) we obtain a *linear* least-squares problem in the state update vector $\boldsymbol{\delta}$,

$$\boldsymbol{\delta}^* = \arg\min_{\boldsymbol{\delta}} \sum_i \left\| \boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i^0) - \boldsymbol{H}_i\boldsymbol{\delta}_i \right\|_{\boldsymbol{\Sigma}_i}^2 \tag{1.21a}$$

$$= \arg\min_{\boldsymbol{\delta}} \sum_i \left\| \left(\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i^0)\right) - \boldsymbol{H}_i\boldsymbol{\delta}_i \right\|_{\boldsymbol{\Sigma}_i}^2, \tag{1.21b}$$

where $\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i^0)$ is the *prediction error* at the linearization point, *i.e.,* the difference between actual and predicted measurement. Above, $\boldsymbol{\delta}^*$ denotes the solution to the locally linearized problem.

By a simple change of variables we can drop the covariance matrices $\boldsymbol{\Sigma}_i$ from this point forward: defining $\boldsymbol{\Sigma}^{1/2}$ as the matrix square root of $\boldsymbol{\Sigma}$, we can rewrite the square Mahalanobis norm as follows:

$$\|\boldsymbol{e}\|_{\boldsymbol{\Sigma}}^2 \triangleq \boldsymbol{e}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{e} = \left(\boldsymbol{\Sigma}^{-1/2}\boldsymbol{e}\right)^\top \left(\boldsymbol{\Sigma}^{-1/2}\boldsymbol{e}\right) = \left\|\boldsymbol{\Sigma}^{-1/2}\boldsymbol{e}\right\|_2^2. \tag{1.22}$$

Hence, we can eliminate the covariances $\boldsymbol{\Sigma}_i$ by pre-multiplying the Jacobian $\boldsymbol{H}_i$ and the prediction error in each term in (1.21b) with $\boldsymbol{\Sigma}_i^{-1/2}$:

$$\boldsymbol{A}_i = \boldsymbol{\Sigma}_i^{-1/2}\boldsymbol{H}_i \tag{1.23a}$$

$$\boldsymbol{b}_i = \boldsymbol{\Sigma}_i^{-1/2}\left(\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i^0)\right). \tag{1.23b}$$

This process is a form of *whitening*. For example, in the case of scalar measurements it simply means dividing each term by the measurement standard deviation σ_i. Note that this eliminates the units of the measurements (e.g., length, angles) so that the different rows can be combined into a single cost function.

1.3.2 SLAM as Least-Squares

After linearization, we finally obtain the following standard least-squares problem:

$$\boldsymbol{\delta}^* = \arg\min_{\boldsymbol{\delta}} \sum_i \|\boldsymbol{A}_i \boldsymbol{\delta}_i - \boldsymbol{b}_i\|_2^2 \tag{1.24a}$$

$$= \arg\min_{\boldsymbol{\delta}} \|\boldsymbol{A}\boldsymbol{\delta} - \boldsymbol{b}\|_2^2 \,, \tag{1.24b}$$

Above $\boldsymbol{A}$ and $\boldsymbol{b}$ are obtained by collecting all whitened Jacobian matrices $\boldsymbol{A}_i$ and whitened prediction errors $\boldsymbol{b}_i$ into one large matrix $\boldsymbol{A}$ and right-hand-side (RHS) vector $\boldsymbol{b}$, respectively.

The Jacobian $\boldsymbol{A}$ is a large-but-sparse matrix, with a block structure that mirrors the structure of the underlying factor graph. We will examine this sparsity structure in detail below. First, however, we review the the classical linear algebra approach to solving this least-squares problem.

1.3.3 Matrix Factorization for Least-Squares

For a full-rank $m \times n$ matrix $\boldsymbol{A}$, with $m \geq n$, the unique least-squares solution to (1.24b) can be found by solving the *normal equations*:

$$\left(\boldsymbol{A}^\top \boldsymbol{A}\right) \boldsymbol{\delta}^* = \boldsymbol{A}^\top \boldsymbol{b}. \tag{1.25}$$

This is normally done by factoring the *information matrix* $\boldsymbol{\Lambda}$ (also called the Hessian matrix), defined and factored as follows:

$$\boldsymbol{\Lambda} \triangleq \boldsymbol{A}^\top \boldsymbol{A} = \boldsymbol{R}^\top \boldsymbol{R}. \tag{1.26}$$

Above, the *Cholesky triangle* $\boldsymbol{R}$ is an upper-triangular $n \times n$ matrix[3] and is computed using *Cholesky factorization*, a variant of lower-upper (LU) factorization for symmetric positive-definite matrices. After this, $\boldsymbol{\delta}^*$ can be found by solving first

$$\boldsymbol{R}^\top \boldsymbol{y} = \boldsymbol{A}^\top \boldsymbol{b} \tag{1.27}$$

for $\boldsymbol{y}$ and then

$$\boldsymbol{R}\,\boldsymbol{\delta}^* = \boldsymbol{y} \tag{1.28}$$

[3] Some treatments, including [397], define the Cholesky triangle as the lower-triangular matrix $\boldsymbol{L} = \boldsymbol{R}^\top$, but the other convention is more convenient here.

for $\boldsymbol{\delta}^*$ by forward and backward substitution, respectively. For dense matrices, Cholesky factorization requires $n^3/3$ flops, and the entire algorithm, including computing half of the symmetric $\boldsymbol{A}^\top \boldsymbol{A}$, requires $(m + n/3)n^2$ flops. One could also use lower-diagonal-upper (LDU) factorization, a variant of Cholesky decomposition that avoids the computation of square roots.

An alternative to Cholesky factorization that is more accurate and more numerically stable is to proceed via *QR-factorization*, which works *without* computing the information matrix $\boldsymbol{\Lambda}$. Instead, we compute the QR-factorization of $\boldsymbol{A}$ itself along with its corresponding RHS:

$$\boldsymbol{A} = \boldsymbol{Q}\begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{bmatrix}, \quad \begin{bmatrix} \boldsymbol{d} \\ \boldsymbol{e} \end{bmatrix} = \boldsymbol{Q}^\top \boldsymbol{b}. \tag{1.29}$$

Here $\boldsymbol{Q}$ is an $m \times m$ orthogonal matrix, $\boldsymbol{d} \in \mathbb{R}^n$, $\boldsymbol{e} \in \mathbb{R}^{m-n}$, and $\boldsymbol{R}$ is the *same* upper-triangular Cholesky triangle. The preferred method for factorizing a dense matrix $\boldsymbol{A}$ is to compute $\boldsymbol{R}$ column by column, proceeding from left to right. For each column j, all nonzero elements below the diagonal are zeroed out by multiplying $\boldsymbol{A}$ on the left with a *Householder reflection matrix* $\boldsymbol{H}_j$. After n iterations $\boldsymbol{A}$ is completely factorized:

$$\boldsymbol{H}_n \cdots \boldsymbol{H}_2 \boldsymbol{H}_1 \boldsymbol{A} = \boldsymbol{Q}^\top \boldsymbol{A} = \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{bmatrix}. \tag{1.30}$$

The orthogonal matrix $\boldsymbol{Q}$ is not usually formed: instead, the transformed RHS $\boldsymbol{Q}^\top \boldsymbol{b}$ is computed by appending $\boldsymbol{b}$ as an extra column to $\boldsymbol{A}$. Because the $\boldsymbol{Q}$ factor is orthogonal, we have

$$\|\boldsymbol{A}\,\boldsymbol{\delta} - \boldsymbol{b}\|_2^2 = \left\|\boldsymbol{Q}^\top \boldsymbol{A}\boldsymbol{\delta} - \boldsymbol{Q}^\top \boldsymbol{b}\right\|_2^2 = \|\boldsymbol{R}\,\boldsymbol{\delta} - \boldsymbol{d}\|_2^2 + \|\boldsymbol{e}\|_2^2\,, \tag{1.31}$$

where we made use of the equalities from (1.29). Clearly, $\|\boldsymbol{e}\|_2^2$ will be the least-squares sum of squared residuals, and the least-squares solution $\boldsymbol{\delta}^*$ can be obtained by solving the triangular system

$$\boldsymbol{R}\,\boldsymbol{\delta}^* = \boldsymbol{d} \tag{1.32}$$

via back-substitution. Note that the upper-triangular factor $\boldsymbol{R}$ obtained using QR factorization is the same (up to possible sign changes on the diagonal) as would be obtained by Cholesky factorization, since

$$\boldsymbol{A}^\top \boldsymbol{A} = \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{bmatrix}^\top \boldsymbol{Q}^\top \boldsymbol{Q} \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{bmatrix} = \boldsymbol{R}^\top \boldsymbol{R}, \tag{1.33}$$

where we again made use of the fact that $\boldsymbol{Q}$ is orthogonal. The cost of QR is dominated by the cost of the Householder reflections, which is $2(m - n/3)n^2$. Comparing this with Cholesky, we see that both algorithms require $O(mn^2)$ operations when $m \gg n$, but that QR-factorization is slower by a factor of 2.

In summary, the linearized optimization problem associated with SLAM can be concisely stated in terms of basic linear algebra. It comes down to factorizing either the information matrix $\boldsymbol{\Lambda}$ or the measurement Jacobian $\boldsymbol{A}$ into square-root form. Because they are based on matrix square roots derived from the *SAM* problem, we have referred to this family of approaches as *square-root SAM*, or $\sqrt{\text{SAM}}$ for short [258, 262].

1.4 Nonlinear Optimization

In this section, we discuss some classic optimization approaches to the nonlinear least-squares problem defined in (1.18). As a reminder, in SLAM the nonlinear least-squares objective function is given by

$$J(\boldsymbol{x}) \triangleq \sum_i \|\boldsymbol{z}_i - \boldsymbol{h}_i(\boldsymbol{x}_i)\|^2_{\boldsymbol{\Sigma}_i} \tag{1.34}$$

and corresponds to a nonlinear factor graph derived from the measurements along with prior densities on some or all unknowns.

Nonlinear least-squares problems cannot be solved directly in general, but require an iterative solution starting from a suitable initial estimate. Nonlinear optimization methods do so by solving a succession of linear approximations to (1.18) in order to approach the minimum [272]. A variety of algorithms exist that differ in how they locally approximate the cost function, and in how they find an improved estimate based on that local approximation. A general in-depth treatment of nonlinear solvers is provided by [820], while [397] focuses on the linear-algebra perspective.

All of the algorithms share the following basic structure: they start from an initial estimate $\boldsymbol{x}^0$. In each iteration, an update step $\boldsymbol{\delta}$ is calculated and applied to obtain the next estimate $\boldsymbol{x}^{t+1} = \boldsymbol{x}^t + \boldsymbol{\delta}$. This process ends when certain convergence criteria are reached, such as the norm of the change $\boldsymbol{\delta}$ falling below a small threshold.

1.4.1 Steepest Descent

Steepest Descent (SD) uses the direction of steepest descent at the current estimate to calculate the following update step:

$$\boldsymbol{\delta}^{\text{sd}} = -\alpha \left. \nabla J(\boldsymbol{x}) \right|_{\boldsymbol{x}=\boldsymbol{x}^t} . \tag{1.35}$$

Here the negative gradient is used to identify the direction of steepest descent. For the nonlinear least-squares objective function (1.34), we locally approximate the objective function as a quadratic, $J(\boldsymbol{x}) \approx \|\boldsymbol{A}(\boldsymbol{x} - \boldsymbol{x}^t) - \boldsymbol{b}\|_2^2$, and obtain the exact gradient $\left.\nabla J(\boldsymbol{x})\right|_{\boldsymbol{x}=\boldsymbol{x}^t} = -2\boldsymbol{A}^\top \boldsymbol{b}$ at the linearization point $\boldsymbol{x}^t$.

The step size α needs to be carefully chosen to balance between safe updates and reasonable convergence speed. An explicit line search can be performed to find a minimum in the given direction. SD is a simple algorithm, but suffers from slow convergence near the minimum.

1.4.2 Gauss-Newton

Gauss-Newton (GN) provides faster convergence by using a second-order update. GN exploits the special structure of the nonlinear least-squares problem to approximate the Hessian using the Jacobian as $\boldsymbol{A}^\top \boldsymbol{A}$. The GN update step is obtained by solving the normal equations (1.25),

$$\boldsymbol{A}^\top \boldsymbol{A}\, \boldsymbol{\delta}^{\mathrm{gn}} = \boldsymbol{A}^\top \boldsymbol{b}, \tag{1.36}$$

by any of the methods in Section 1.3.3. For a well-behaved (i.e., nearly quadratic) objective function and a good initial estimate, Gauss-Newton exhibits nearly quadratic convergence. If the quadratic fit is poor, a GN step can lead to a new estimate that is further from the minimum and subsequent divergence.

1.4.3 Levenberg-Marquardt

The Levenberg-Marquardt (LM) algorithm allows for iterating multiple times to convergence while controlling in which region one is willing to trust the quadratic approximation made by Gauss-Newton. Hence, such a method is often called a *trust-region method.*

To combine the advantages of both the SD and GN methods, Levenberg [653] proposed to modify the normal equations (1.25) by adding a non-negative constant $\lambda \in \mathbb{R}^+ \cup \{0\}$ to the diagonal

$$\left(\boldsymbol{A}^\top \boldsymbol{A} + \lambda \mathbf{I}\right) \boldsymbol{\delta}^{\mathrm{lm}} = \boldsymbol{A}^\top \boldsymbol{b}. \tag{1.37}$$

Note that for $\lambda = 0$ we obtain GN, and for large λ we approximately obtain $\boldsymbol{\delta}^* \approx \frac{1}{\lambda} \boldsymbol{A}^\top \boldsymbol{b}$, an update in the negative gradient direction of the cost function J (1.34). Hence, LM can be seen to blend naturally between the GN and SD methods.

Marquardt [738] later proposed to take into account the scaling of the diagonal entries to provide faster convergence:

$$\left(\boldsymbol{A}^\top \boldsymbol{A} + \lambda \operatorname{diag}(\boldsymbol{A}^\top \boldsymbol{A})\right) \boldsymbol{\delta}^{\mathrm{lm}} = \boldsymbol{A}^\top \boldsymbol{b}. \tag{1.38}$$

This modification causes larger steps in the steepest-descent direction if the gradient is small (nearly flat directions of the objective function), because there the inverse of the diagonal entries will be large. Conversely, in steep directions of the objective function the algorithm becomes more cautious and takes smaller steps. Both modifications of the normal equations can be interpreted in Bayesian terms as adding a zero-mean prior to the system.

A key difference between GN and LM is that the latter rejects updates that would lead to a higher sum of squared residuals. A rejected update means that the nonlinear function is locally not well-behaved, and smaller steps are needed. This is achieved by heuristically increasing the value of λ, for example by multiplying its current value by a factor of 10, and resolving the modified normal equations.

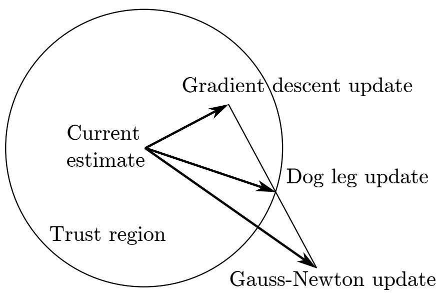


Figure 1.5 Powell's dogleg algorithm combines the separately computed Gauss-Newton and gradient descent update steps.

On the other hand, if a step leads to a reduction of the sum of squared residuals, it is accepted, and the state estimate is updated accordingly. In this case, λ is reduced (*e.g.,* by dividing by a factor of 10), and the algorithm repeats with a new linearization point, until convergence.

1.4.4 Dogleg Minimization

Powell's dogleg (PDL) algorithm [892] can be a more efficient alternative to LM [704]. A major disadvantage of the Levenberg-Marquardt algorithm is that in case a step gets rejected, the modified information matrix has to be refactored, which is the most expensive component of the algorithm. Hence, the key idea behind PDL is to separately compute the GN and SD steps, and then combine appropriately. If the LM step gets rejected, the directions of the GN and SD steps are still valid, and they can be combined in a different way until a reduction in the cost is achieved. Hence, each update of the state estimate only involves one matrix factorization, as opposed to several.

Figure 1.5 shows how the GN and SD steps are combined. The combined step starts with the SD update, followed by a sharp bend (hence the term dogleg) towards the GN update, but stopping at the trust region boundary. Unlike LM, PDL maintains an explicit trust region within which we trust the linear assumption. The appropriateness of the linear approximation is determined by the gain ratio

$$\rho = \frac{J(\boldsymbol{x}^t) - J(\boldsymbol{x}^t + \boldsymbol{\delta})}{L(\mathbf{0}) - L(\boldsymbol{\delta})}, \tag{1.39}$$

where $L(\boldsymbol{\delta}) = \boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{\delta} - \boldsymbol{A}^\top \boldsymbol{b}$ is the linearization of the nonlinear quadratic cost function J from (1.34) at the current estimate $\boldsymbol{x}^t$. If ρ is small (i.e., $\rho < 0.25$) then the cost has not reduced as predicted by the linearization and the trust region is reduced. On the other hand, if the reduction is as predicted (or better, i.e., $\rho > 0.75$), then the trust region is increased depending on the magnitude of the update vector, and the step is accepted.

1.5 Factor Graphs and Sparsity

The solvers presented so far assume that the matrices involved may be dense. Dense methods will not scale to realistic problem sizes in SLAM. For the toy problem in Figure 1.1 a dense method will work fine. The larger simulation example, with its factor graph shown in Figure 1.4, is more representative of real-world problems. However, it is still relatively small as real SLAM problems go, where problems with thousands or even millions of unknowns are common. Yet, we are able to handle these without a problem because of sparsity.

The sparsity can be appreciated directly from looking at the factor graph. It is clear from Figure 1.4 that the graph is *sparse* (i.e., it is by no means a fully connected graph). The odometry chain linking the 100 unknown poses is a linear structure of 100 binary factors, instead of the possible 100^2 (binary) factors. In addition, with 20 landmarks we could have up to 2000 factors linking each landmark to each pose: the true number is closer to 400. And finally, there are no factors between landmarks at all. This reflects that we have not been given any information about their relative position. This structure is typical of most SLAM problems.

1.5.1 The Sparse Jacobian and its Factor Graph

The key to modern SLAM algorithms is exploiting sparsity, and an important property of factor graphs in SLAM is that they represent the sparse block structure in the resulting sparse Jacobian matrix $\boldsymbol{A}$. To see this, let us revisit the least-squares problem that is the key computation in the inner loop of the nonlinear SLAM problem:

$$\boldsymbol{\delta}^* = \arg\min_{\boldsymbol{\delta}} \sum_i \|\boldsymbol{A}_i\,\boldsymbol{\delta}_i - \boldsymbol{b}_i\|_2^2\,. \tag{1.40}$$

Each term above is derived from a factor in the original, nonlinear SLAM problem, linearized around the current linearization point (1.21b). The matrices $\boldsymbol{A}_i$ can be broken up in blocks corresponding to each variable, and collected in a large, block-sparse Jacobian whose sparsity structure is given exactly by the factor graph.

Even though these linear problems typically arise as inner iterations in nonlinear optimization, we drop the $\boldsymbol{\delta}$ notation below, as everything holds for general linear problems regardless of their origin.

Consider the factor graph for the small toy example in Figure 1.1. After linearization, we obtain a sparse system $[\boldsymbol{A}|\boldsymbol{b}]$ with the block structure in Figure 1.6. Comparing this with the factor graph, it is obvious that every factor corresponds to a block-row, and every variable corresponds to a block-column of $\boldsymbol{A}$. In total there are nine block-rows, one for every factor in the factorization of $\phi(\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3, \boldsymbol{\ell}_1, \boldsymbol{\ell}_2)$.

$$[\boldsymbol{A}|\boldsymbol{b}] = \begin{array}{c} \\ \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \\ \phi_5 \\ \phi_6 \\ \phi_7 \\ \phi_8 \\ \phi_9 \end{array} \begin{array}{c} \begin{array}{cccccc} \delta\boldsymbol{\ell}_1 & \delta\boldsymbol{\ell}_2 & \delta\boldsymbol{p}_1 & \delta\boldsymbol{p}_2 & \delta\boldsymbol{p}_3 & \boldsymbol{b} \end{array} \\ \left[\begin{array}{ccccc|c} & & \boldsymbol{A}_{13} & & & \boldsymbol{b}_1 \\ & & \boldsymbol{A}_{23} & \boldsymbol{A}_{24} & & \boldsymbol{b}_2 \\ & & & \boldsymbol{A}_{34} & \boldsymbol{A}_{35} & \boldsymbol{b}_3 \\ \boldsymbol{A}_{41} & & & & & \boldsymbol{b}_4 \\ & \boldsymbol{A}_{52} & & & & \boldsymbol{b}_5 \\ & & \boldsymbol{A}_{63} & & & \boldsymbol{b}_6 \\ \boldsymbol{A}_{71} & & \boldsymbol{A}_{73} & & & \boldsymbol{b}_7 \\ \boldsymbol{A}_{81} & & & \boldsymbol{A}_{84} & & \boldsymbol{b}_8 \\ & \boldsymbol{A}_{92} & & & \boldsymbol{A}_{95} & \boldsymbol{b}_9 \end{array}\right] \end{array}$$

Figure 1.6 Block structure of the sparse Jacobian $\boldsymbol{A}$ for the toy SLAM example in Figure 1.1 with $\boldsymbol{\delta} = \left[\delta\boldsymbol{\ell}_1^\top \; \delta\boldsymbol{\ell}_2^\top \; \delta\boldsymbol{p}_1^\top \; \delta\boldsymbol{p}_2^\top \; \delta\boldsymbol{p}_3^\top\right]^\top$. Blank entries are zeros.

$$\begin{bmatrix} \boldsymbol{\Lambda}_{11} & & \boldsymbol{\Lambda}_{13} & \boldsymbol{\Lambda}_{14} & \\ & \boldsymbol{\Lambda}_{22} & & & \boldsymbol{\Lambda}_{25} \\ \boldsymbol{\Lambda}_{31} & & \boldsymbol{\Lambda}_{33} & \boldsymbol{\Lambda}_{34} & \\ \boldsymbol{\Lambda}_{41} & & \boldsymbol{\Lambda}_{43} & \boldsymbol{\Lambda}_{44} & \boldsymbol{\Lambda}_{45} \\ & \boldsymbol{\Lambda}_{52} & & \boldsymbol{\Lambda}_{54} & \boldsymbol{\Lambda}_{55} \end{bmatrix}$$

Figure 1.7 The information matrix $\boldsymbol{\Lambda} \triangleq \boldsymbol{A}^\top \boldsymbol{A}$ for the toy SLAM problem.

1.5.2 The Sparse Information Matrix and its Graph

When using Cholesky factorization for solving the normal equations, as explained in Section 1.3.3, we first form the Hessian or information matrix $\boldsymbol{\Lambda} = \boldsymbol{A}^\top \boldsymbol{A}$.[4] In general, since the Jacobian $\boldsymbol{A}$ is block-sparse, the Hessian $\boldsymbol{\Lambda}$ is expected to be sparse as well. By construction, the Hessian is a symmetric matrix, and if a unique MAP solution to the problem exists, it is also positive definite.

The information matrix $\boldsymbol{\Lambda}$ can be associated with another, *undirected* graphical model for the SLAM problem, namely a *Markov random field (MRF).* In contrast to a factor graph, an MRF is a graphical model that involves only the variables. The graph G of an MRF is an undirected graph: the edges only indicate that there is *some* interaction between the variables involved. At the block level, the sparsity pattern of $\boldsymbol{\Lambda} = \boldsymbol{A}^\top \boldsymbol{A}$ is exactly the adjacency matrix of G.

Figure 1.7 shows the information matrix $\boldsymbol{\Lambda}$ associated with our running toy example. In this case, there are five variables that partition the Hessian as shown. The zero blocks indicate which variables do not interact (e.g., $\boldsymbol{\ell}_1$ and $\boldsymbol{\ell}_2$ have no direct interaction). Figure 1.8 shows the corresponding MRF.

In what follows, we will frequently refer to the undirected graph G of the MRF associated with an inference problem. However, we will not use the MRF graphical

[4] Note that $\boldsymbol{A}^\top \boldsymbol{A}$ is not true Hessian, but is often used to approximate Hessian by truncating a Taylor series of the residual.

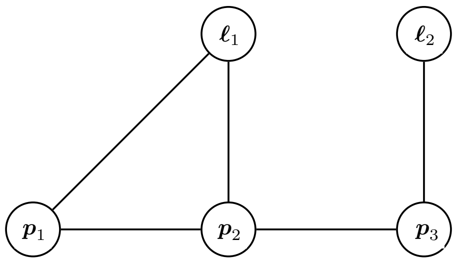


Figure 1.8 The Hessian matrix $\mathbf{\Lambda}$ can be interpreted as the matrix associated with the Markov random field representation for the problem.

model much beyond that as factor graphs are better suited to our needs. They are able to express a finer-grained factorization, and are more closely related to the original problem formulation. For example, if there exist ternary (or higher) factors in the factor graph, the graph G of the equivalent MRF connects those nodes in an undirected clique (a group of fully connected variables), but the origin of the corresponding clique potential is lost. In linear algebra, this reflects the fact that many matrices $\boldsymbol{A}$ can yield the same $\mathbf{\Lambda} = \boldsymbol{A}^\top \boldsymbol{A}$ matrix: important information on the sparsity is lost.

1.5.3 Sparse Factorization

We have seen MAP estimation amounts to solving a linear system of equations as described in Section 1.3.3. In the case of nonlinear least-squares problems, we solve such a system repeatedly in an iterative setup. We have seen in the previous two sections that both $\boldsymbol{A}$ and $\boldsymbol{A}^\top \boldsymbol{A}$ enjoy sparsity determined by the factor graph and MRF connectivity, respectively. Without going into detail, this known sparsity pattern can be used to greatly speed up either Cholesky factorization (in the case of working with $\boldsymbol{A}^\top \boldsymbol{A}$) or QR-factorization (in the case of working with $\boldsymbol{A}$). Efficient software implementations are available, e.g., CHOLMOD [196] and SuiteSparseQR [249], which are also used under the hood by several software packages. In practice, sparse Cholesky or LDU factorization outperform QR factorization on sparse problems as well, and not just by a constant factor.

The flop count for sparse factorization will be much lower than for a dense matrix. Crucially, the column ordering chosen for the sparse matrices can dramatically influence the total flop-count. While any order will ultimately produce an identical MAP estimate, the variable order determines the *fill-in* of matrix factors (i.e., the extra nonzero entries beyond the sparsity pattern of the matrix being factored). It is known that finding the variable ordering that minimizes fill-in during matrix factorization is an NP-hard problem [1234], so we must resort to using good heuristics. This will in turn affect the computational complexity of the factorization algorithm.

We demonstrate this by way of an example. Recall the larger simulation example, with its factor graph shown in Figure 1.4. The sparsity patterns for the corresponding sparse Jacobian matrix $\boldsymbol{A}$ is shown in Figure 1.9. Also shown is the pattern for the information matrix $\boldsymbol{\Lambda} \triangleq \boldsymbol{A}^\top \boldsymbol{A}$, in the top-right corner. On the right of Figure 1.9, we show the resulting upper triangular Choleksy factor $\boldsymbol{R}$ for two different orderings. Both of them are sparse, and both of them satisfy $\boldsymbol{R}^\top \boldsymbol{R} = \boldsymbol{A}^\top \boldsymbol{A}$ (up to a permutation of the variables), but they differ in the amount of sparsity they exhibit. It is exactly this that will determine how expensive it is to factorize $\boldsymbol{A}$. The first version of the ordering comes naturally: the poses come first and then the landmarks, leading to a sparse $\boldsymbol{R}$ factor with 9399 nonzero entries. In contrast, the sparse factor $\boldsymbol{R}$ in the bottom right was obtained by reordering the variables according to the Column approximate minimum degree permutation (COLAMD) heuristic [36, 250] and only has 4168 nonzero entries. Yet back-substitution gives exactly the same solution for both versions.

It is worth mentioning that other tools, like pre-conditioned conjugate gradient, can solve the normal equations *iteratively*. In visual SLAM, which has a very specific sparsity pattern, power iterations have also been used successfully [1169]. However, sparse factorization is still the method of choice for most SLAM problems and has a nice graphical model interpretation, which we discuss next.

1.6 Elimination

We have so far restricted ourselves to a linear-algebra explanation of performing inference for SLAM. In this section, we expand our worldview by thinking about inference more abstractly using graphical models directly. This will ultimately lead us to current state-of-the-art SLAM solvers based on a concept called the *Bayes tree* for incremental smoothing and mapping in the next section.

1.6.1 Variable Elimination Algorithm

There exists a general algorithm that can, given any (preferably sparse) factor graph, compute the corresponding posterior density $p(\boldsymbol{x}|\boldsymbol{z})$ on the unknown variables $\boldsymbol{x}$ in a form that allows easy recovery of the MAP solution to the problem. As we saw, a factor graph represents the unnormalized posterior $\phi(\boldsymbol{x}) \propto p(\boldsymbol{x}|\boldsymbol{z})$ as a product of factors, and in SLAM problems this graph is typically generated directly from the measurements. The *variable elimination* algorithm is a recipe for converting a factor graph into another graphical model called a *Bayes net*, which depends only on the unknown variables $\boldsymbol{x}$. This then allows for easy MAP inference (as well as other operations such as sampling and/or marginalization).

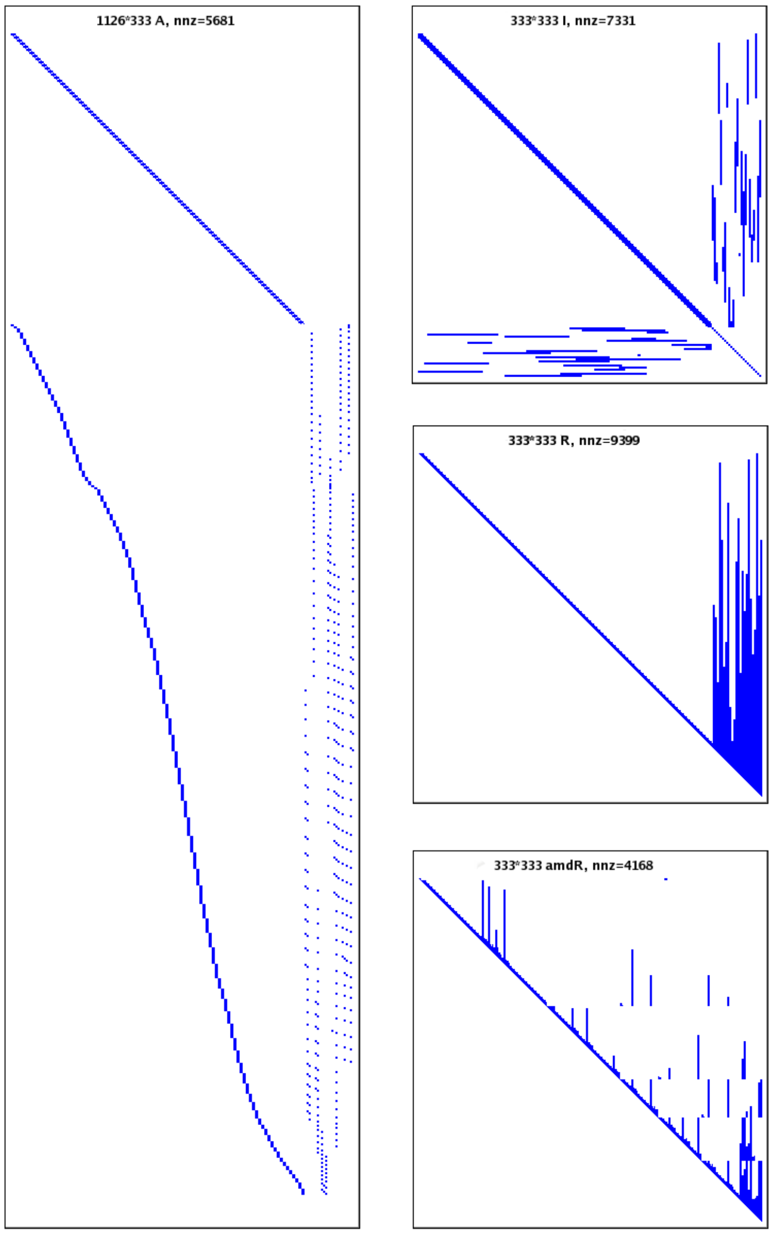


Figure 1.9 On the left, the measurement Jacobian $\boldsymbol{A}$ associated with the problem in Figure 1.4, which has $3 \times 95 + 2 \times 24 = 333$ unknowns. The number of rows, 1126, is equal to the number of (scalar) measurements. Also given is the number of nonzero entries “nnz”. On the right: (top) the information matrix $\boldsymbol{\Lambda} \triangleq \boldsymbol{A}^{\top}\boldsymbol{A}$; (middle) its upper triangular Cholesky triangle $\boldsymbol{R}$; (bottom) an alternative factor *amdR* obtained with a better variable ordering (COLAMD).

In particular, the variable elimination algorithm is a way to factorize any factor graph of the form

$$\phi(\boldsymbol{x}) = \phi(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \tag{1.41}$$

into a factored Bayes net probability density of the form

$$p(\boldsymbol{x}) = p(\boldsymbol{x}_1|\boldsymbol{s}_1)p(\boldsymbol{x}_2|\boldsymbol{s}_2)\ldots p(\boldsymbol{x}_n) = \prod_j p(\boldsymbol{x}_j|\boldsymbol{s}_j), \tag{1.42}$$

where $\boldsymbol{s}_j$ denotes an assignment to the *separator* $\boldsymbol{s}(\boldsymbol{x}_j)$ associated with variable $\boldsymbol{x}_j$ under the chosen variable ordering $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. The separator is defined as the set of variables on which $\boldsymbol{x}_j$ is conditioned, after elimination. While this factorization is akin to the chain rule, eliminating a sparse factor graph will typically lead to small separators.

The elimination algorithm proceeds by eliminating one variable $\boldsymbol{x}_j$ at a time, starting with the complete factor graph $\phi_{1:n}$. As we eliminate each variable $\boldsymbol{x}_j$, we generate a single conditional $p(\boldsymbol{x}_j|\boldsymbol{s}_j)$, as well as a reduced factor graph $\phi_{j+1:n}$ on the remaining variables. After all variables have been eliminated, the algorithm returns the resulting Bayes net with the desired factorization.

To eliminate a single variable $\boldsymbol{x}_j$ given a partially eliminated factor graph $\phi_{j:n}$, we first remove all factors $\phi_i(\boldsymbol{x}_i)$ that are adjacent to $\boldsymbol{x}_j$ and multiply them into the product factor $\psi(\boldsymbol{x}_j, \boldsymbol{s}_j)$. We then factorize $\psi(\boldsymbol{x}_j, \boldsymbol{s}_j)$ into a conditional distribution $p(\boldsymbol{x}_j|\boldsymbol{s}_j)$ on the eliminated variable $\boldsymbol{x}_j$, and a new factor $\tau(\boldsymbol{s}_j)$ on the separator $\boldsymbol{s}_j$:

$$\psi(\boldsymbol{x}_j, \boldsymbol{s}_j) = p(\boldsymbol{x}_j|\boldsymbol{s}_j)\tau(\boldsymbol{s}_j). \tag{1.43}$$

Hence, *the entire factorization from $\phi(\boldsymbol{x})$ to $p(\boldsymbol{x})$ is seen to be a succession of n local factorization steps.* When eliminating the last variable $\boldsymbol{x}_n$ the separator $\boldsymbol{s}_n$ will be empty, and the conditional produced will simply be a prior $p(\boldsymbol{x}_n)$ on $\boldsymbol{x}_n$.

One possible elimination sequence for the toy example is shown in Figure 1.10, for the ordering $\boldsymbol{\ell}_1$, $\boldsymbol{\ell}_2$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, $\boldsymbol{p}_3$. In each step, the variable being eliminated is shaded gray, and the new factor $\tau(\boldsymbol{s}_j)$ on the separator $\boldsymbol{s}_j$ is shown in red. Taken as a whole, the variable elimination algorithm factorizes the factor graph $\phi(\boldsymbol{\ell}_1, \boldsymbol{\ell}_2, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3)$ into the Bayes net in Figure 1.10 (bottom right), corresponding to the factorization

$$p(\boldsymbol{\ell}_1, \boldsymbol{\ell}_2, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3) = p(\boldsymbol{\ell}_1|\boldsymbol{p}_1, \boldsymbol{p}_2)\, p(\boldsymbol{\ell}_2|\boldsymbol{p}_3)\, p(\boldsymbol{p}_1|\boldsymbol{p}_2)\, p(\boldsymbol{p}_2|\boldsymbol{p}_3)\, p(\boldsymbol{p}_3). \tag{1.44}$$

1.6.2 Linear-Gaussian Elimination

In the case of linear measurement functions and additive normally distributed noise, the *elimination algorithm is equivalent to sparse matrix factorization.* Both sparse Cholesky and QR factorization are a special case of the general algorithm.

As explained before, the elimination algorithm proceeds one variable at a time. For every variable $\boldsymbol{x}_j$ we remove all factors $\phi_i(\boldsymbol{x}_i)$ adjacent to $\boldsymbol{x}_j$ and form the

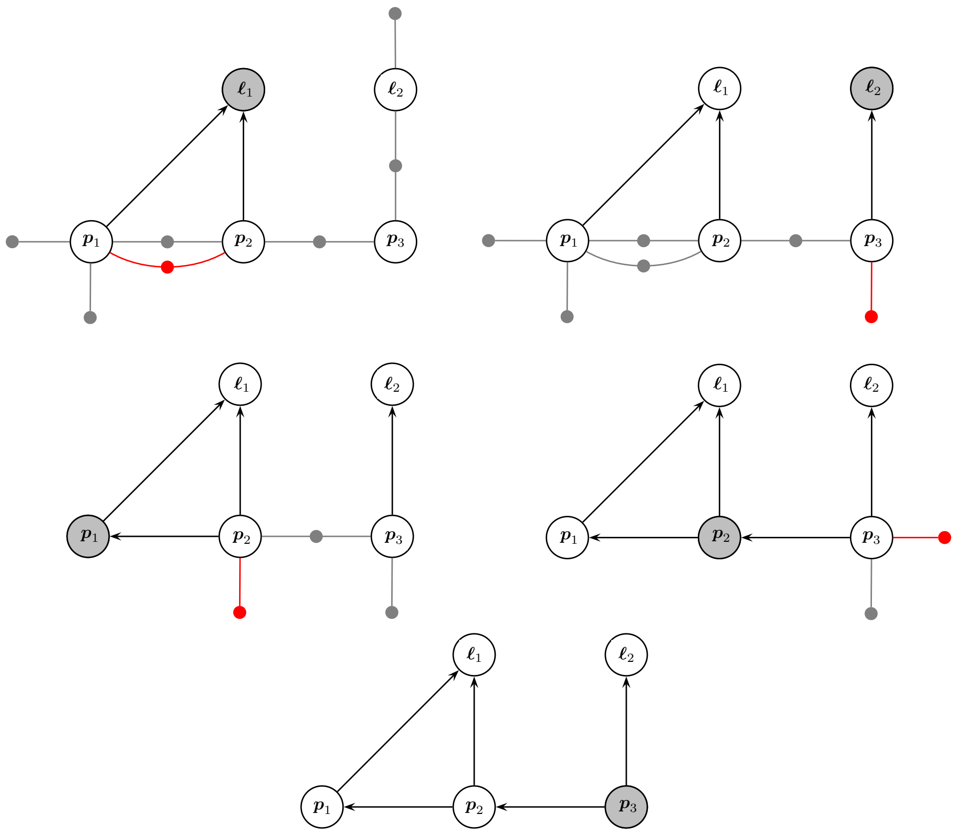


Figure 1.10 Variable elimination for the toy SLAM example, transforming the factor graph from Figure 1.2 into a Bayes net (bottom right), using the ordering $\boldsymbol{\ell}_1$, $\boldsymbol{\ell}_2$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, $\boldsymbol{p}_3$.

intermediate product factor $\psi(\boldsymbol{x}_j, \boldsymbol{s}_j)$. This can be done by accumulating all the matrices $\boldsymbol{A}_i$ into a new, larger block-matrix $\bar{\boldsymbol{A}}_j$, as we can write

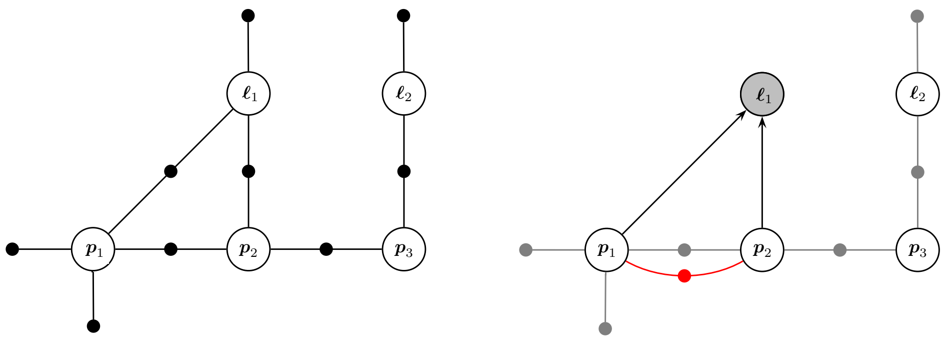
$$\psi(\boldsymbol{x}_j, \boldsymbol{s}_j) \leftarrow \prod_{i \in \mathcal{N}_j} \phi_i(\boldsymbol{x}_i) \tag{1.45a}$$

$$= \exp\left(-\frac{1}{2}\sum_i \|\boldsymbol{A}_i \boldsymbol{x}_i - \boldsymbol{b}_i\|_2^2\right) \tag{1.45b}$$

$$= \exp\left(-\frac{1}{2}\left\|\bar{\boldsymbol{A}}_j[\boldsymbol{x}_j; \boldsymbol{s}_j] - \bar{\boldsymbol{b}}_j\right\|_2^2\right), \tag{1.45c}$$

where the new RHS vector $\bar{\boldsymbol{b}}_j$ stacks all $\boldsymbol{b}_i$ and ‘;’ also denotes vertical stacking.

Consider eliminating the variable $\boldsymbol{\ell}_1$ in the toy example. The adjacent factors are ϕ_4, ϕ_7, and ϕ_8, in turn inducing the separator $\boldsymbol{s}_1 = [\boldsymbol{p}_1; \boldsymbol{p}_2]$. The product factor is



$$\left[\begin{array}{ccccc}
 & & \boldsymbol{A}_{13} & & \\
 & & \boldsymbol{A}_{23} & \boldsymbol{A}_{24} & \\
 & & & \boldsymbol{A}_{34} & \boldsymbol{A}_{35} \\
\boldsymbol{A}_{41} & & & & \\
 & \boldsymbol{A}_{52} & & & \\
 & & \boldsymbol{A}_{63} & & \\
\boldsymbol{A}_{71} & & \boldsymbol{A}_{73} & & \\
\boldsymbol{A}_{81} & & & \boldsymbol{A}_{84} & \\
 & \boldsymbol{A}_{92} & & & \boldsymbol{A}_{95}
\end{array}\right]
\qquad
\left[\begin{array}{c|cccc}
\boldsymbol{R}_1 & & \boldsymbol{T}_{13} & \boldsymbol{T}_{14} & \\
\hline
 & & \boldsymbol{A}_{13} & & \\
 & & \boldsymbol{A}_{23} & \boldsymbol{A}_{24} & \\
 & & & \boldsymbol{A}_{34} & \boldsymbol{A}_{35} \\
 & \boldsymbol{A}_{52} & & & \\
 & & \boldsymbol{A}_{63} & & \\
 & \boldsymbol{A}_{92} & & & \boldsymbol{A}_{95} \\
 & & \tilde{\boldsymbol{A}}_{13} & \tilde{\boldsymbol{A}}_{14} &
\end{array}\right]$$

Figure 1.11 Eliminating the variable $\boldsymbol{\ell}_1$ as a partial sparse factorization step.

then equal to

$$\psi\left(\boldsymbol{\ell}_1, \boldsymbol{p}_1, \boldsymbol{p}_2\right) = \exp\left(-\frac{1}{2}\left\|\bar{\boldsymbol{A}}_1[\boldsymbol{\ell}_1; \boldsymbol{p}_1; \boldsymbol{p}_2] - \bar{\boldsymbol{b}}_1\right\|_2^2\right), \tag{1.46}$$

with

$$\bar{\boldsymbol{A}}_1 \triangleq \begin{bmatrix} \boldsymbol{A}_{41} & & \\ \boldsymbol{A}_{71} & \boldsymbol{A}_{73} & \\ \boldsymbol{A}_{81} & & \boldsymbol{A}_{84} \end{bmatrix}, \qquad \bar{\boldsymbol{b}}_1 \triangleq \begin{bmatrix} \boldsymbol{b}_4 \\ \boldsymbol{b}_7 \\ \boldsymbol{b}_8 \end{bmatrix}. \tag{1.47}$$

Looking at the sparse Jacobian in Figure 1.6, this simply boils down to taking out the block-rows with nonzero blocks in the first column, corresponding to the three factors adjacent to $\boldsymbol{\ell}_1$.

Next, factorizing the product $\psi(\boldsymbol{x}_j, \boldsymbol{s}_j)$ can be done in several different ways. We discuss the QR variant, as it more directly connects to the linearized factors. In particular, the augmented matrix $[\bar{\boldsymbol{A}}_j|\bar{\boldsymbol{b}}_j]$ corresponding to the product factor $\psi(\boldsymbol{x}_j, \boldsymbol{s}_j)$ can be rewritten using partial QR-factorization [397] as follows:

$$[\bar{\boldsymbol{A}}_j|\bar{\boldsymbol{b}}_j] = \boldsymbol{Q}\begin{bmatrix} \boldsymbol{R}_j & \boldsymbol{T}_j & \boldsymbol{d}_j \\ & \tilde{\boldsymbol{A}}_\tau & \tilde{\boldsymbol{b}}_\tau \end{bmatrix}, \tag{1.48}$$

where $\boldsymbol{R}_j$ is an upper-triangular matrix. This allows us to factor $\psi(\boldsymbol{x}_j, \boldsymbol{s}_j)$ as follows:

$$\begin{aligned}\psi(\boldsymbol{x}_j, \boldsymbol{s}_j) &= \exp\left\{-\frac{1}{2}\left\|\bar{\boldsymbol{A}}_j[\boldsymbol{x}_j; \boldsymbol{s}_j] - \bar{\boldsymbol{b}}_j\right\|_2^2\right\} &\text{(1.49a)}\\ &= \exp\left\{-\frac{1}{2}\left\|\boldsymbol{R}_j\boldsymbol{x}_j + \boldsymbol{T}_j\boldsymbol{s}_j - \boldsymbol{d}_j\right\|_2^2\right\}\exp\left\{-\frac{1}{2}\left\|\tilde{\boldsymbol{A}}_\tau\boldsymbol{s}_j - \tilde{\boldsymbol{b}}_\tau\right\|_2^2\right\}\\ &= p(\boldsymbol{x}_j|\boldsymbol{s}_j)\tau(\boldsymbol{s}_j), &\text{(1.49b)}\end{aligned}$$

where we used the fact that the rotation matrix $\boldsymbol{Q}$ does not alter the value of the norms involved.

In the toy example, Figure 1.11 shows the result of eliminating the first variable in the example, the landmark $\boldsymbol{\ell}_1$ with separator $[\boldsymbol{p}_1; \boldsymbol{p}_2]$. We show the operation on the factor graph *and* the corresponding effect on the sparse Jacobian from Figure 1.6, omitting the RHS. The partition above the line corresponds to a sparse, upper-triangular matrix $\boldsymbol{R}$ that is being formed. New contributions to the matrix are shown: blue for the contributions to $\boldsymbol{R}$, and red for newly created factors. For completeness, we show the four remaining variable elimination steps in Figure 1.12, showing an end-to-end example of how QR factorization proceeds on a small example. The final step shows the equivalence between the resulting Bayes net and the sparse upper-triangular factor $\boldsymbol{R}$.

The entire elimination algorithm, using partial QR to eliminate a single variable, is equivalent to *sparse QR factorization*. As the treatment above considers multi-dimensional variables $\boldsymbol{x}_j \in \mathbb{R}^{n_j}$, this is in fact an instance of *multi-frontal QR factorization* [296], as we eliminate several scalar variables at a time, which is beneficial for processor utilization. While in our case the scalar variables are grouped because of their semantic meaning in the inference problem, sparse linear algebra codes typically analyze the problem to group for maximum computational efficiency. In many cases these two strategies are closely aligned.

1.6.3 Sparse Cholesky Factor as a Bayes Net

The equivalence between variable elimination and sparse matrix factorization reveals that the graphical model associated with an upper triangular matrix is a Bayes net! Just like a factor graph is the graphical embodiment of a sparse Jacobian, and an MRF can be associated with the Hessian, a Bayes net reveals the sparsity structure of a Cholesky factor. In hindsight, this perhaps is not too surprising: a Bayes net is a directed acyclic graph (DAG), and that is exactly the 'upper-triangular' property for matrices.

What's more, the Cholesky factor corresponds to a *Gaussian Bayes net*, which we define as one made up of linear-Gaussian conditionals. The variable elimination algorithm holds for general densities, but in case the factor graph only contains linear measurement functions and Gaussian additive noise, the resulting Bayes net

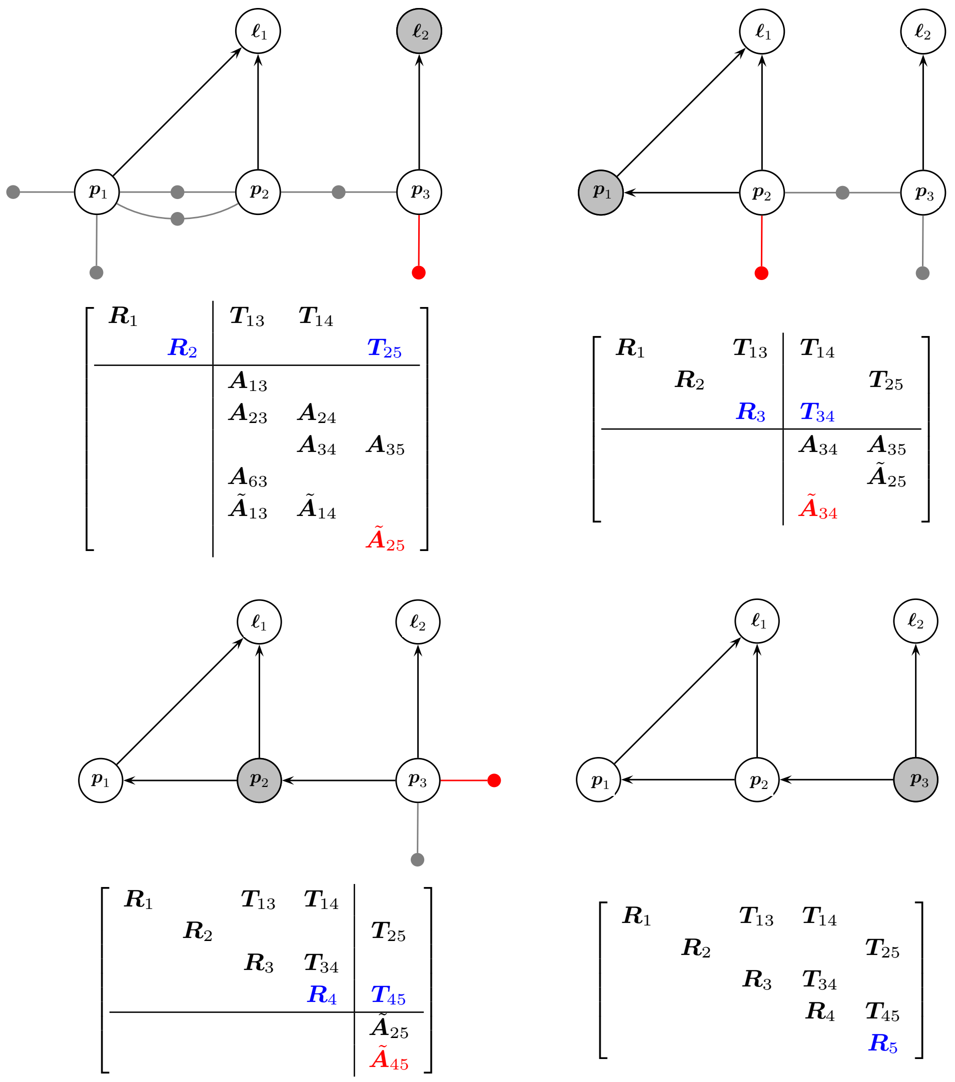


Figure 1.12 The remaining elimination steps for the toy example, completing a full QR factorization. The last step in the bottom right shows the equivalence between the resulting Bayes net and the sparse Cholesky factor $\boldsymbol{R}$.

has a very specific form. We discuss the details below, as well as how to solve for the MAP estimate in the linear case.

As we discussed, the Gaussian factor graph corresponding to the linearized non-linear problem is transformed by elimination into the density $p(\boldsymbol{x})$ given by the now-familiar Bayes-net factorization:

$$p(\boldsymbol{x}) = \prod_j p(\boldsymbol{x}_j | \boldsymbol{s}_j). \tag{1.50}$$

In both QR and Cholesky variants, the conditional densities $p(\boldsymbol{x}_j|\boldsymbol{s}_j)$ are given by

$$p(\boldsymbol{x}_j|\boldsymbol{s}_j) = k \exp\left(-\frac{1}{2}\left\|\boldsymbol{R}_j \boldsymbol{x}_j + \boldsymbol{T}_j \boldsymbol{s}_j - \boldsymbol{d}_j\right\|_2^2\right), \tag{1.51}$$

which is a linear-Gaussian density on the eliminated variable $\boldsymbol{x}_j$. Indeed, we have

$$\|\boldsymbol{R}_j \boldsymbol{x}_j + \boldsymbol{T}_j \boldsymbol{s}_j - \boldsymbol{d}_j\|_2^2 = (\boldsymbol{x}_j - \boldsymbol{\mu}_j)^\top \boldsymbol{R}_j^\top \boldsymbol{R}_j (\boldsymbol{x}_j - \boldsymbol{\mu}_j) \triangleq \|\boldsymbol{x}_j - \boldsymbol{\mu}_j\|_{\boldsymbol{\Sigma}_j}^2, \tag{1.52}$$

where the mean $\boldsymbol{\mu}_j = \boldsymbol{R}_j^{-1}(\boldsymbol{d}_j - \boldsymbol{T}_j \boldsymbol{s}_j)$ depends linearly on the separator $\boldsymbol{s}_j$, and the covariance matrix is given by $\boldsymbol{\Sigma}_j = (\boldsymbol{R}_j^\top \boldsymbol{R}_j)^{-1}$. Hence, the normalization constant $k = |2\pi\boldsymbol{\Sigma}_j|^{-\frac{1}{2}}$.

After the elimination step is complete, back-substitution is used to obtain the MAP estimate of each variable. As seen in Figure 1.12, the last variable eliminated does not depend on any other variables. Thus, the MAP estimate of the last variable can be directly extracted from the Bayes net. By proceeding in reverse elimination order, the values of all the separator variables for each conditional will always be available from the previous steps, allowing the estimate for the current frontal variable to be computed.

At every step, the MAP estimate for the variable $\boldsymbol{x}_j$ is the conditional mean,

$$\boldsymbol{x}_j^* = \boldsymbol{R}_j^{-1}(\boldsymbol{d}_j - \boldsymbol{T}_j \boldsymbol{s}_j^*), \tag{1.53}$$

since by construction the MAP estimate for the separator $\boldsymbol{s}_j^*$ is fully known by this point.

1.7 Incremental SLAM

In an incremental SLAM setting, we want to compute the optimal trajectory and map whenever we receive new measurements while traversing the environment, or at least at regular intervals. One way to do so is to *update* the most recent matrix factorization with the new measurements, to reuse the computation that already incorporated all previous measurements. In the linear case, this is possible through incremental factorization methods, the dense versions of which are also discussed at length in Golub and Loan [397]. However, matrix factorization operates on linear systems, but most SLAM problems of practical interest are *nonlinear*. Using incremental matrix factorization, it is far from obvious how re-linearization can be performed incrementally without refactoring the complete matrix. To overcome this problem we once again resort to graphical models, and introduce a new graphical model, the *Bayes tree*. We then show how to incrementally update the Bayes tree as new measurements and states are added to the system, leading to the incremental smoothing and mapping (iSAM) algorithm.

1.7.1 The Bayes Tree

It is well known that inference in a tree-structured graph is efficient. In contrast, the factor graphs associated with typical robotics problems contain many loops. Still, we can construct a tree-structured graphical model in a two-step process: first, perform variable elimination on the factor graph to obtain a Bayes net with a special property. Second, exploit that special property to find a tree structure over *cliques* in this Bayes net.

In particular, a Bayes net obtained by running the elimination algorithm on a factor graph satisfies a special property: it is *chordal*, meaning that any undirected cycle of length greater than three has a *chord*, i.e., an edge connecting two non-consecutive vertices on the cycle. In AI and machine learning, a chordal graph is more commonly said to be *triangulated*. Because it is still a Bayes net, the corresponding joint density $p(\boldsymbol{x})$ is given by factorizing over the individual variables $\boldsymbol{x}_j$,

$$p(\boldsymbol{x}) = \prod_j p(\boldsymbol{x}_j|\boldsymbol{\pi}_j), \tag{1.54}$$

where $\boldsymbol{\pi}_j$ are the parent nodes of $\boldsymbol{x}_j$. However, although the Bayes net is chordal, at this variable level it is still a non-trivial graph: neither chain-like nor tree-structured. The chordal Bayes net for our running toy SLAM example is shown in the last step of Figure 1.10, and it is clear that there is an undirected cycle $\boldsymbol{p}_1 - \boldsymbol{p}_2 - \boldsymbol{\ell}_1$, implying it does not have a tree-structured form.

By identifying cliques in this chordal graph, the Bayes net may be rewritten as a *Bayes tree*. We introduce this new, tree-structured graphical model to capture the *clique structure* of the Bayes net. It is not obvious that cliques in the Bayes net should form a tree. They do so because of the chordal property, although we will not attempt to prove that here. Listing all these cliques in an undirected tree yields a *clique tree*, also known as a *junction tree* in AI and machine learning. The Bayes tree is just a directed version of this that preserves information about the elimination order.

More formally, a Bayes tree is a directed tree where the nodes represent *cliques* $\boldsymbol{c}_k$ of the underlying chordal Bayes net. In particular, we define one conditional density $p(\boldsymbol{f}_k|\boldsymbol{s}_k)$ per node, with the *separator* $\boldsymbol{s}_k$ as the intersection $\boldsymbol{c}_k \cap \boldsymbol{\varpi}_k$ of the clique $\boldsymbol{c}_k$ and its parent clique $\boldsymbol{\varpi}_k$. The *frontal variables* $\boldsymbol{f}_k$ are the remaining variables, i.e., $\boldsymbol{f}_k \triangleq \boldsymbol{c}_k \setminus \boldsymbol{s}_k$. Notationally, we write $\boldsymbol{c}_k = \boldsymbol{f}_k : \boldsymbol{s}_k$ for a clique. The following expression gives the joint density $p(\boldsymbol{x})$ on the variables $\boldsymbol{x}$ defined by a Bayes tree:

$$p(\boldsymbol{x}) = \prod_k p(\boldsymbol{f}_k|\boldsymbol{s}_k). \tag{1.55}$$

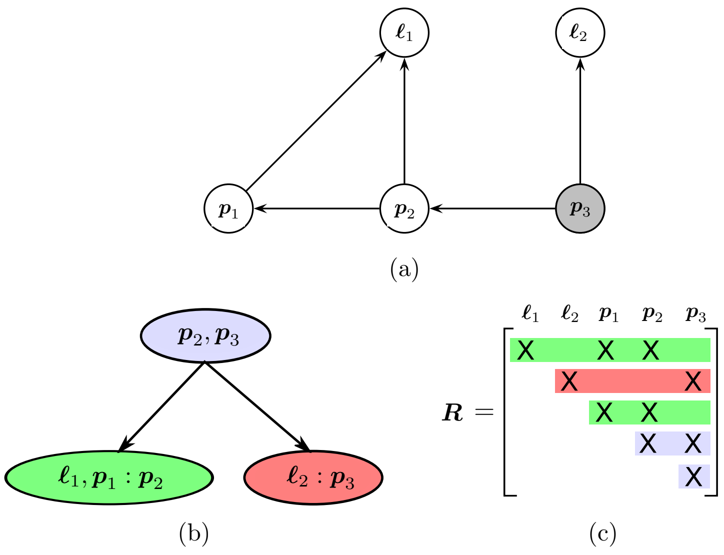

$\boldsymbol{\ell}_1$
$\boldsymbol{\ell}_2$
$\boldsymbol{p}_1$
$\boldsymbol{p}_2$
$\boldsymbol{p}_3$
(a)
$\boldsymbol{p}_2, \boldsymbol{p}_3$
$\boldsymbol{\ell}_1, \boldsymbol{p}_1 : \boldsymbol{p}_2$
$\boldsymbol{\ell}_2 : \boldsymbol{p}_3$
(b)
$\boldsymbol{R} =$
X
(c)

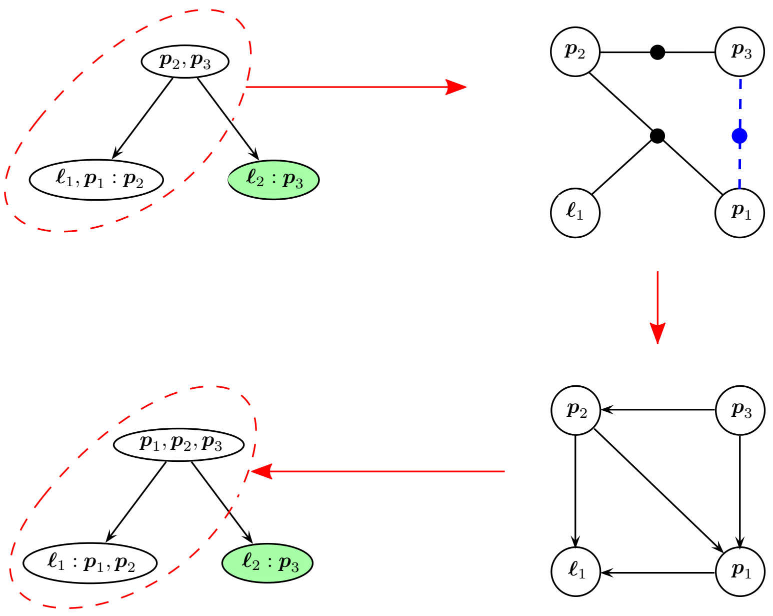
Figure 1.13 The Bayes tree (b) and the associated square root information matrix $\boldsymbol{R}$ (c) describing the clique structure in the chordal Bayes net (a) based on our canonical example from Figure 1.2. A Bayes tree is similar to a clique tree, but is better at capturing the formal equivalence between sparse linear algebra and inference in graphical models. The association of cliques with rows in the $\boldsymbol{R}$ factor is indicated by color.

For the root $\boldsymbol{f}_r$, the separator is empty, i.e., it is a simple prior $p(\boldsymbol{f}_r)$ on the root variables. The way Bayes trees are defined, the separator $\boldsymbol{s}_k$ for a clique $\boldsymbol{c}_k$ is always a subset of the parent clique $\boldsymbol{\varpi}_k$, and hence the directed edges in the graph have the same semantic meaning as in a Bayes net: conditioning.

The Bayes tree associated with our canonical toy SLAM problem (Figure 1.2) is shown in Figure 1.13. The root clique $\boldsymbol{c}_1 = \boldsymbol{p}_2, \boldsymbol{p}_3$ (shown in blue) comprises $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$, which intersects with two other cliques, $\boldsymbol{c}_2 = \boldsymbol{\ell}_1, \boldsymbol{p}_1 : \boldsymbol{p}_2$ shown in green, and $\boldsymbol{c}_3 = \boldsymbol{\ell}_2 : \boldsymbol{p}_3$ shown in red. The colors also indicate how the rows of square-root information matrix $\boldsymbol{R}$ map to the different cliques, and how the Bayes tree captures independence relationships between them. For example, the green and red rows only intersect in variables that belong to the root clique, as predicted.

1.7.2 Updating the Bayes Tree

Incremental inference corresponds to a simple editing of the Bayes tree. This view provides a better explanation and understanding of the otherwise abstract incremental matrix factorization process. It also allows us to store and compute the square-root information matrix in the form of a Bayes tree, a deeply meaningful sparse storage scheme.



$\boldsymbol{p}_2, \boldsymbol{p}_3$
$\boldsymbol{\ell}_1, \boldsymbol{p}_1 : \boldsymbol{p}_2$
$\boldsymbol{\ell}_2 : \boldsymbol{p}_3$
$\boldsymbol{p}_2$
$\boldsymbol{p}_3$
$\boldsymbol{\ell}_1$
$\boldsymbol{p}_1$
$\boldsymbol{p}_2$
$\boldsymbol{p}_3$
$\boldsymbol{\ell}_1$
$\boldsymbol{p}_1$
$\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3$
$\boldsymbol{\ell}_1 : \boldsymbol{p}_1, \boldsymbol{p}_2$
$\boldsymbol{\ell}_2 : \boldsymbol{p}_3$

Figure 1.14 Updating a Bayes tree with a new factor, based on the example in Figure 1.13. The affected part of the Bayes tree is highlighted for the case of adding a new factor between $\boldsymbol{p}_1$ and $\boldsymbol{p}_3$. Note that the right branch (green) is not affected by the change. (top right) The factor graph generated from the affected part of the tree with the new factor (dashed blue) inserted. (bottom right) The chordal Bayes net resulting from eliminating the factor graph. (bottom left) The Bayes tree created from the chordal Bayes net, with the unmodified right 'orphan' sub-tree from the original Bayes tree added back in.

To incrementally update the Bayes tree, we selectively convert part of the Bayes tree back into factor-graph form. When a new measurement is added this corresponds to adding a factor, e.g., a measurement involving two variables will induce a new binary factor $\phi(\boldsymbol{x}_j, \boldsymbol{x}_{j'})$. In this case, *only* the paths in the Bayes tree between the cliques containing $\boldsymbol{x}_j$ and $\boldsymbol{x}_{j'}$ and the root will be affected. The sub-trees below these cliques are unaffected, as are any other sub-trees not containing $\boldsymbol{x}_j$ or $\boldsymbol{x}_{j'}$. Hence, to update the Bayes tree, the affected parts of the tree are converted back into a factor graph, and the new factor associated with the new measurement is added to it. By re-eliminating this temporary factor graph, using whatever elimination ordering is convenient, a new Bayes tree is formed and the unaffected sub-trees can be reattached.

In order to understand why only the top part of the tree is affected, we look at two important properties of the Bayes tree. These directly arise from the fact that it encodes the information flow during elimination. The Bayes tree is formed from the chordal Bayes net following the inverse elimination order. In this way, variables in each clique collect information from their child cliques via the elimination of these children. Thus, information in any clique propagates only upwards to the root. Second, the information from a factor enters elimination only when the first variable connected to that factor is eliminated. Combining these two properties, we see that a new factor cannot influence any other variables that are not successors of the factor's variables. However, a factor involving variables having different (i.e., independent) paths to the root means that these paths must now be re-eliminated to express the new dependency between them.

Figure 1.14 shows how these incremental factorization/inference steps are applied to our canonical SLAM example. In this example, we add a new factor between $\boldsymbol{p}_1$ and $\boldsymbol{p}_3$, affecting only the left branch of the tree, marked by the red dashed line in to top-left figure. We then create the factor graph shown in the top-right figure by creating a factor for each of the clique densities, $p(\boldsymbol{p}_2, \boldsymbol{p}_3)$ and $p(\boldsymbol{\ell}_1, \boldsymbol{p}_1|\boldsymbol{p}_2)$, and add the new factor $f(\boldsymbol{p}_1, \boldsymbol{p}_3)$. The bottom-right figure shows the eliminated graph using the ordering $\boldsymbol{\ell}_1, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3$. And finally, in the bottom-left figure, the reassembled Bayes tree is shown consisting of two parts: the Bayes tree derived from the eliminated graph, and the unaffected clique from the original Bayes tree (shown in green).

Figure 1.15 shows an example of the Bayes tree for a small SLAM sequence. Shown is the tree for step 400 of the well-known Manhattan world simulated sequence by Olson et al. [830]. As a robot explores the environment, new measurements only affect parts of the tree, and only those parts are re-calculated.

1.7.3 Incremental Smoothing and Mapping

Putting all of the above together and addressing some practical considerations about re-linearization yields a state-of-the-art incremental, nonlinear approach to MAP estimation in robotics, iSAM. The first version, iSAM1[538], used the incremental matrix factorization methods from Golub and Loan [397]. However, linearization in iSAM1 was handled in a sub-optimal way: it was done for the full factor graph at periodic instances and/or when matrix fill-in became unwieldy. The second version of the approach, iSAM2, uses a Bayes tree representation for the posterior density [540]. It then employs Bayes tree incremental updating as each new measurement comes in, as described above.

What variable ordering should we use in re-eliminating the affected cliques? Only the variables in the affected part of the Bayes tree are updated. One strategy then is to apply COLAMD locally to the affected variables. However, we can do better: we force recently accessed variables to the end of the ordering, i.e., into the root clique. For this incremental variable ordering strategy one can use the constrained COLAMD algorithm [250]. This both forces the most recently accessed variables to the end and still provides a good overall ordering. Generally, subsequent updates will then only affect a small part of the tree, and can therefore be expected to be efficient in most cases, except for large loop closures.

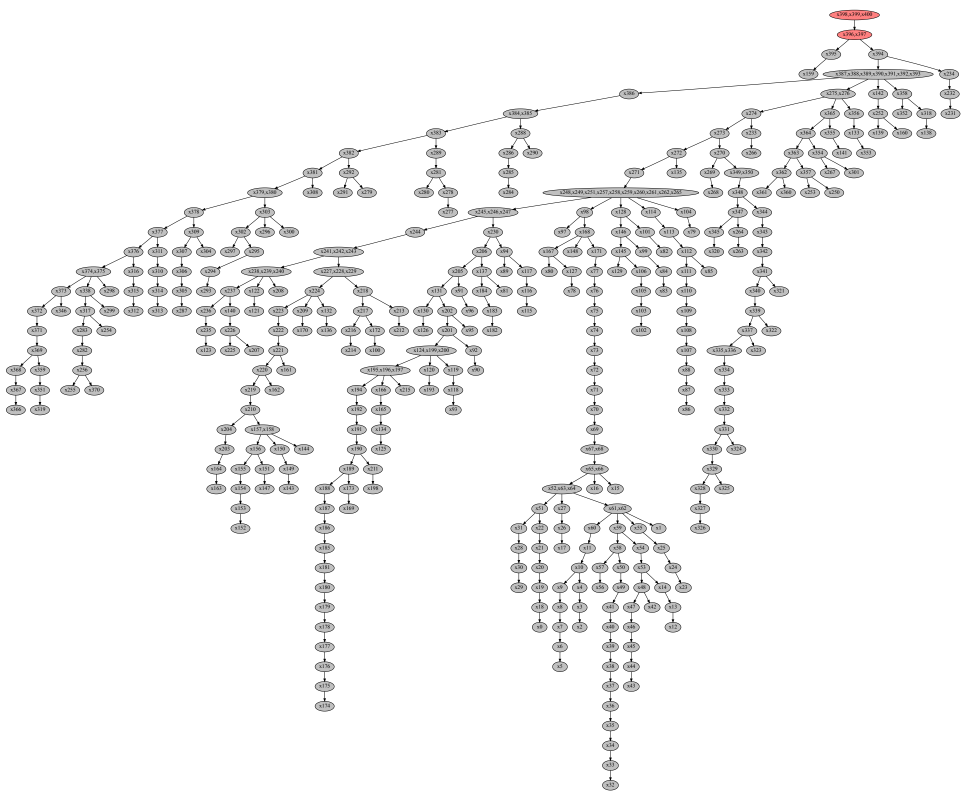


Figure 1.15 An example of the Bayes tree data structure for a small SLAM sequence. The incremental nonlinear least-squares estimation algorithm iSAM2 [540] is based on viewing incremental factorization as editing the graphical model corresponding to the posterior probability of the solution, the Bayes tree. As a robot explores the environment, new measurements often only affect small parts of the tree, and only those parts are recalculated (shown in red).

After updating the tree we also need to *update the solution.* Back-substitution in the Bayes tree proceeds from the root (which does not depend on any other variables) and proceeds to the leaves. However, it is typically not necessary to recompute a solution for all variables: local updates to the tree often do not affect variables in remote parts of the tree. Instead, at each clique we can check the difference in variable estimates that is propagated downwards and stop when this difference falls below a small threshold.

Our motivation for introducing the Bayes tree was to incrementally solve nonlinear optimization problems. For this we *selectively re-linearize* factors that contain variables whose deviation from the linearization point exceeds a small threshold. In contrast to the tree modification above, we now have to redo all cliques that contain the affected variables, not just as frontal variables, but also as separator variables. This affects larger parts of the tree, but in most cases is still significantly cheaper than recomputing the complete tree. We also have to go back to the original factors, instead of directly turning the cliques into a factor graph. And that requires caching certain quantities during elimination. The overall incremental nonlinear algorithm, iSAM2, is described in much more detail in [540].

iSAM1 and iSAM2 have been applied successfully to many different robotics estimation problems with non-trivial constraints between variables that number into the millions, as will be discussed subsequent chapters. Both are implemented in the GTSAM library, which can be found at `https://github.com/borglab/gtsam`.