Bayesian inverse kinematics vs. least-squares inverse kinematics in estimates of planar postures and rotations in the absence of soft tissue artifact

Todd C. Pataky[a], Jos Vanrenterghem[b], Mark A. Robinson[c]

[a]*Department of Human Health Sciences, Kyoto University, Japan*
[b]*Department of Rehabilitation Sciences, KU Leuven, Belgium*
[c]*Research Institute for Sport and Exercise Sciences, Liverpool John Moores University, UK*

Abstract

A variety of inverse kinematics (IK) algorithms exist for estimating postures and displacements from a set of noisy marker positions, typically aiming to minimize IK errors by distributing errors amongst all markers in a least-squares (LS) sense. This paper describes how Bayesian inference can contrastingly be used to maximize the probability that a given stochastic kinematic model would produce the observed marker positions. We developed Bayesian IK for two planar IK applications: (1) kinematic chain posture estimates using an explicit forward kinematics model, and (2) rigid body rotation estimates using implicit kinematic modeling through marker displacements. We then tested and compared Bayesian IK results to LS results in Monte Carlo simulations in which random marker error was introduced using Gaussian noise amplitudes ranging uniformly between 0.2 mm and 2.0 mm. Results showed that Bayesian IK was more accurate than LS-IK in over 92% of simulations, with the exception of one center-of-rotation coordinate planar rotation, for which Bayesian IK was more accurate in only 68% of simulations. Moreover, while LS errors increased with marker noise, Bayesian errors were comparatively unaffected by noise amplitude. Nevertheless, whereas the LS solutions required average computational durations of less than 0.5 s, average Bayesian IK durations ranged from 11.6 s for planar rotation to over 2000 s for kinematic chain postures. These results suggest that Bayesian IK can yield order-of-magnitude IK improvements for simple planar IK, but also that its computational demands may make it impractical for some applications.

Word counts:
Abstract: 249 (max 250)
Main text: 3249 (max 3500)

Corresponding Author:
Todd Pataky, pataky.todd.2m@kyoto-u.ac.jp, +81-075-751-4175

Keywords: *biomechanics; motion capture; human movement analysis; Bayesian analysis; Markov-Chain Monte Carlo simulations*

November 28, 2018

1. Introduction

Marker-based inverse kinematics (IK) is a fundamental component of motion capture, and a variety of IK algorithms exist in the literature for estimating segmental pose from noisy marker measurement (Aristidou and Lasenby, 2009). In Biomechanics, a wide variety of IK algorithms exist for minimizing noisy marker errors in a least-squares (LS) sense (Söderkvist and Wedin, 1993; Halvorsen et al., 1999; Gamage and Lasenby, 2002; Halvorsen, 2003; Schwartz and Rozumalski, 2005; Wells et al., 2017), including near-real time LS algorithms (Borbély and Szolgay, 2017; Pizzolato et al., 2016). Augmented LS algorithms have also been proposed, including for example Kalman filtering, which can substantially reduce LS errors (De Groote et al., 2008). As far as we are aware, all commonly employed biomechanics-specific software packages including Visual3D (C-Motion, Inc., Germantown, USA) and OpenSim (Delp et al., 2007) use LS-based IK. Nevertheless, in the broader literature, and in the robotics literature in particular, a variety of non-LS IK algorithms have been proposed, including artificial neural networks (Duka, 2014), heuristic methods (Aristidou and Lasenby, 2011), and Bayesian inference (Courty and Arnaud, 2008).

This paper focusses on Bayesian IK because it is known that Bayesian algorithms usually yield more accurate results than LS algorithms for generic (non-IK) parameter estimation problems, even for relatively simple cases like regression (Elster and Wübbeler, 2017) and multivariate mean estimation (Yuan et al., 2016). Moreover, it has been shown in the animation literature that optimized Bayesian implementations can outperform LS methods in animation IK problems involving a large number of degrees of freedom (>30) (Courty and Arnaud, 2008). Nevertheless, no systematic comparison of Bayesian vs. IK has previously been performed for simple kinematic models, so the minimal, general kinematic characteristics necessary for Bayesian IK and LS-IK diverge are presently unknown.

In generic (non-IK) applications, both LS and Bayesian approaches can be interpreted as follows. Given a set of observed values y, construct a forward model $f(q) = \hat{y}$, where q are arbitrary parameters to be estimated, and $\hat{y}$ are the predicted values. Both LS and Bayesian approaches aim to estimate q given y and the forward model. If the observed data y have zero error, and if the forward model is perfect, then there is a unique solution for q and the LS and Bayesian approaches are equivalent. However, when y embodies measurement error and/or when the forward model is imperfect, then there is no unique solution for q, so an objective function must be added in order to estimate q. LS approaches aim to minimize error objective functions like $g(q) = \sum_i (y_i - \hat{y}_i)^2$, where i indexes the observed data points. Bayesian approaches contrastingly aim to maximize probability-based objective functions, which can be approximately interpreted as $h(q) = P(q|y)$, or maximizing the probability of observing the parameters q given the data y. A more precise interpretation is that Bayesian approaches mutually maximize the maximum likelihoods of the parameters' posterior distributions. This procedure involves an iterative computational procedure in which distributions for all parameters are numerically generated based on assumed prior distributions, then updated based on the observed data, and the parameters' distributions are iteratively updated until all distributions' maxima are mutually maximized (Patil et al., 2010) (Appendix A & B).

When interpreted in the context of marker-based IK problems, y represents the marker positions, q represents the arbitrary kinematic parameters to be estimated, like joint angles or joint center locations, and $f(q)$ represents the forward kinematics model. LS approaches minimize predicted marker position errors $\sum_i (y_i - \hat{y}_i)^2$ and Bayesian approaches maximize the probability that a particular set of parameters q could yield the observed marker positions y (Appendix C).

To our knowledge Bayesian inference has not been previously applied to simple IK problems involving a small number of degrees of freedom. If Bayesian IK outperforms LS methods for simple IK problems, it is conceivable that it could be useful for more practical IK problems, including those involving STA. However the utility of Bayesian IK should first be assessed using simpler models in order to elucidate the model characteristics for which Bayesian IK and LS-IK diverge.

The purpose of this study was to compare Bayesian IK to LS-IK performance in estimates of planar kinematic chain poses and in planar rigid body rotations involving no STA. To this end we review the fundamentals of Bayesian inference (Appendix A & B), extend these concept to basic IK problems (Appendix C), then demonstrate why the LS and Bayesian solutions can diverge, even for simple IK problems (Appendix D). The remainder of the main manuscript extends the simple IK discussed in Appendix C–D to slightly more complex planar mechanisms so that the corresponding Bayesian IK performance could be compared directly to the performances of two common STA-free

LS techniques from the literature (Söderkvist and Wedin, 1993; Halvorsen et al., 1999).

2. Methods

Analyses were conducted in Python 3.6 (van Rossum, 2014) using Anaconda 3.5 (Anaconda, 2017). Appendices containing tutorials for Bayesian inference and Bayesian IK fundamentals, along with supplementary results, are available in a public repository at `https://github.com/0todd0000/BayesIK`.

2.1. Planar kinematic chain posture estimates

2.1.1. Forward kinematics model

Kinematic chains were modeled using rigid segments which each contained one rigid marker plate with four markers (Fig.1). Pilot simulations involving three, five and six markers were conducted but did not qualitatively affect results so are omitted in interest of space.

In order to permit direct comparison between least-squares (LS) and Bayesian techniques, local positions of the markers with respect to the proximal joint centers were assumed to be known. Forward kinematics (computing marker positions from mechanism pose) was implemented by setting the true global position of the proximal joint ($\boldsymbol{r}$) and all segment orientations (ϕ_i), thereby yielding the global positions of all markers. From proximal to distal, three segment lengths were chosen to approximately reflect lower limb segment lengths: 0.45, 0.35 and 0.25 m.

Marker noise was modeled as multivariate Gaussian with uniform standard deviation σ. In Monte Carlo simulations σ was drawn randomly from a uniform distribution with range: [0.1, 2.0 mm], representing the approximate range of marker measurement errors expected with modern motion capture systems (Chen et al., 1994; Windolf et al., 2008). Joint angles were drawn randomly from the uniform distribution spanning [0, 360 deg], thereby encompassing all possible mechanism postures.

For each simulation iteration the IK goal was to calculate mechanism posture ($\boldsymbol{r}$ and ϕ_i) given the noisy marker positions, given the aforementioned forward kinematics model. A total of 1000 simulation iterations was conducted for each of the 1-, 2- and 3-link models depicted in Fig.1. More simulation iterations did not qualitatively affect the results so are not reported.

2.1.2. Least-squares approach

The least-squares (LS) approach minimized the function:

$$f(\boldsymbol{\tau}) = \sum_{i}^{M} (\boldsymbol{r}_{\mathrm{m}i} - \boldsymbol{r}'_{\mathrm{m}i}(\boldsymbol{\tau}))^2 \tag{1}$$

where $\boldsymbol{\tau}$ is the generalized posture vector (2), i indexes the M markers, and $\boldsymbol{r}_{\mathrm{m}i}$ and $\boldsymbol{r}'_{\mathrm{m}i}(\boldsymbol{\tau})$ are the 'measured' (simulated noisy) and calculated marker positions, respectively. For the 3-link chain mechanism the generalized posture vector is defined as:

$$\boldsymbol{\tau} \equiv \begin{Bmatrix} r_x & r_y & \phi_1 & \phi_2 & \phi_3 \end{Bmatrix}^{\mathrm{T}} \tag{2}$$

The optimum solution for $\boldsymbol{\tau}$ was found using quasi-Newton minimization (Nocedal and Wright, 2006) (p.136). Optimization was started from the true, known posture.

2.1.3. Bayesian approach

Bayesian IK was implemented using PyMC (Patil et al., 2010). First a Bayesian model was constructed to represent the forward kinematics model (Fig.2). Given the observed marker positions, the Bayesian solution was calculated using Markov-Chain Monte-Carlo simulations, with Metropolis stepping, which mutually maximized the maximum likelihoods of all posture parameters' posterior distributions. To avoid numerical optimization bias associated with the known true posture, Bayesian calculations were started from the LS solution, and with conservative, uniform priors. Specifically, the range for r_x and r_y was ± 100 mm, and the range for ϕ_i was ± 180 deg. The range for measurement precision parameter ε (see Appendix C) was $[0.1\tau,\ 10\tau]$, where τ was the true measurement precision as defined by σ.

The LS and Bayesian IK solutions were compared both overall in terms of: median error, its interquartile range, and computational duration. They were also compared proportionately, by counting the number of (noisy) mechanism postures for which the Bayesian approach yielded lower error for each parameter. Bayesian calculations were performed over 100,000 iterations with a burn-in of 10,000 iterations and a thinning rate of 5. These iteration parameters were selected based on pilot simulations which were found to yield numerically stable results. Note that these 100,000 main Bayesian calculation iterations were performed separately for each of the aforementioned 1000 Monte Carlo iterations.

2.2. Planar rigid body rotation estimates

2.2.1. Forward kinematics model

A single rigid body was rotated through angle $\Delta\phi$ about a proximal joint center located at position $\boldsymbol{r}$ (Fig.3). During this rotation the ith marker was displaced according to displacement vector $\Delta\boldsymbol{r}_i$. The markers were assumed to be fixed to the rigid body but their true local positions were not assumed to be known.

2.2.2. Least-squares approaches

Two LS approaches from the literature were adopted. The first (Söderkvist and Wedin, 1993) used singular value decomposition to solve the Procrustes problem of optimal transformation between the pre- and post- displacement frames. The second (Halvorsen et al., 1999) used a regular LS analysis of the displacement midpoints.

2.2.3. Bayesian approach

The Bayesian IK approach used herein assumed that the observed pre-displacement marker positions $\boldsymbol{r}_i$ were the true positions, and thus that only the post-displacement marker positions $\boldsymbol{r}_i'$ were affected by the center of rotation $\boldsymbol{r}$ and angular displacement $\Delta\phi$. The IK goal was thus to mutually maximize the posterior maximum likelihoods of $\boldsymbol{r}$ and $\Delta\phi$ given the observed post-rotation marker positions $\boldsymbol{r}_i'$. Bayesian calculations were performed over 10,000 iterations with a burn-in of 5,000 iterations and a thinning rate of 3. Like above, these iteration parameters were selected based on pilot simulations which were found to yield numerically stable results.

3. Results

3.1. Planar kinematic chain posture estimates

For the 1-link kinematic chain model (Fig.1), Bayesian IK was found to yield smaller errors than LS-IK for all three postural parameters: r_x, r_y and ϕ (Fig.5a–c). In particular, while 81.0%, 81.1% and 98.4% of Bayesian IK errors for r_x, r_y and ϕ were less than 0.1 mm, 0.1 mm and 0.1 deg, respectively, only 8.1%, 6.2% and 19.9% of the LS errors were less than these thresholds. Similar results were obtained for the 2- and 3-link models, but in interest of space are presented as supplementary material (Appendix E). Additionally, Bayesian IK yielded smaller error than LS-IK in more than 92% of all kinematic chain simulations (Table 1). Nevertheless, the computational resources required for the LS and Bayesian techniques were very different, with average LS computational durations well under 0.5 s, and Bayesian durations well over 2000 s (Table 2).

3.2. Planar rigid body rotation estimates

Bayesian IK tended to outperform the two LS methods (Fig.5d–f, Fig.6). In particular, 62.1%, 96.7% and 91.9% of Bayesian errors were less than 0.5 mm, 0.5 mm and 0.1 deg for r_x, r_y and $\Delta\phi$, respectively, as compared with 30.7%, 18.1% and 16.1% for Söderkvist and Wedin (1993) and 3.6%, 9.3% and 6.0% for Halvorsen et al. (1999). Bayesian IK errors were smaller than LS errors in over 90% of all simulations, with the exception of r_x for which Bayesian IK errors were lower than Söderkvist and Wedin (1993) in only 67.9% of simulations (Table 1).

Similar results were apparent when considering different noise amplitudes separately (Fig.6): Bayesian IK errors tended to be much smaller than LS-IK errors for all except the smallest noise amplitudes, where the LS result tended to converge to Bayesian levels. Additionally, while LS errors tended to increase with marker noise (Fig.6a–c), this trend was not apparent for the Bayesian errors (Fig.6d–f).

Similar to the kinematic chain results, Bayesian IK computations required a much longer time than did the LS techniques. Whereas average computational durations for the LS techniques were on the order of 1 ms, Bayesian calculations required approximately 10 s Table 2).

4. Discussion

This study summarized the fundamentals of Bayesian inference (Appendix A&B) as they apply to planar IK problems (Appendix C&D), and quantified the accuracies of Bayesian IK and LS-IK for a variety of planar mechanisms. Results show that Bayesian IK and LS-IK solutions diverge even for simple 2-DOF planar mechanisms (Appendix D). For planar kinematic chain pose estimates and for planar rigid body rotations, Bayesian IK yielded order-of-magnitude more accurate IK estimates (Fig. 5), and these were more accurate than LS estimates in 95.3% of simulation results (Table 1). Bayesian IK was additionally less affected by marker noise amplitude than LS-IK (Fig. 6).

The current kinematic chain pose estimates used the identical forward kinematics model for both LS-IK and and Bayesian IK. The LS-IK algorithm started from the true (known) IK solution, giving it the maximum chance to converge to the true solution. Contrastingly the Bayesian IK solution started from the LS-IK solution, giving it the maximum chance to converge to the LS-IK solution. Given this simulation framework, the fact that Bayesian IK was able to find a better solution than LS-IK provides strong evidence that Bayesian IK handles marker error in a numerically superior manner. This suggests more generally that probabilistic maximization as opposed to error minimization may yield better strategy for dealing with biomechanical error.

An additional advantage of Bayesian approaches not explored in this study is that it yields posterior distributions (Appendix A). These distributions represent explicit data-driven estimations of uncertainty, not only for marker noise, but also for all modeled kinematic parameters including joint angle and joint center. LS-IK contrastingly yields only implicit estimates of marker noise, as embodied in the objective function (Eqn.1). In other words, the LS-IK approach provides just a single scalar estimate of uncertainty (the objective function value) that represents only marker-level error. In contrast, Bayesian IK provides a comprehensive model of uncertainty, in the form of posterior distributions for all modeled parameters, including (in this study): marker position, joint center, joint angle and marker measurement precision. Bayesian IK would also yield uncertainty models for additional parameters including: segment lengths, STA parameters, etc., if they were included in the forward kinematics model.

Bayesian IK's comprehensive model of kinematic uncertainty may be advantageous because the posterior distributions from IK can be propagated as prior distributions to subsequent data processing steps including smoothing and inverse dynamics. Ultimately this uncertainty may be important when attempting to make statistical inferences regarding the populations that the data are meant to represent. In other words, in all biomechanical analyses involving LS estimates, uncertainty is by definition considered only at inter-trial or inter-subject levels. Bayesian IK, and potentially also Bayesian inverse dynamics, could change this framework to also include uncertainty in each estimated parameter, including measured external forces and estimated joint and segment dynamics. This may be important because apparent inter-trial and/or inter-subject differences could conceivably fall within the range of Bayesian uncertainty, implying that these differences could be artifacts of computational uncertainty rather than functionally important differences. In contrast, if inter-trial and/or inter-subject differences fall well outside Bayesian IK's posterior distributions, it would reinforce current practice standards in which these differences are regarded as potentially functionally important.

Although LS-IK aims to minimize marker error, we show in Supplementary Material that Bayesian IK converges to LS-IK, unintuitively only when measurement error is presumed to be zero, or equivalently when measurements are presumed to be infinitely precise (Appendix C). For all other presumed marker error levels, Bayesian IK generally yields more accurate results than LS-IK (Appendix D). This paradox highlights the contrasting perspectives that LS-IK and Bayesian IK have of measured marker positions. LS-IK regards measured marker positions as known, and distributes presumed errors equally amongst the markers unless markers are non-uniformly weighted, in which case it would distribute errors equally in a weighted sense. Bayesian IK contrastingly regards measured marker positions more flexibly, as just one manifestation of an infinite continuum of measurement outcome possibilities. This flexible perspective allows Bayesian IK to ignore poor measurements in favor of the measurements which are more consistent with the forward stochastic model. Equivalently, Bayesian IK constitutes a form of objective marker weighting, in which markers are dynamically weighted based on the measured positions' probabilistic consistency with the forward kinematics model.

The main limitation of this study was that only planar poses and rotations were considered. For Bayesian IK to be useful for experimental biomechanics it would have to support estimates of 3D poses and transformations. Bayesian IK has already been implemented for such 3D problems in the robotics literature (Courty and Arnaud, 2008) so is likely also applicable to marker-based IK, but we leave this for future work.

A second limitation is that this study considered only single data frames: single postures or single displacements. It is conceivable that sequential Bayesian IK estimates over subsequent frames yield high frequency noise, and consequentially unrealistic accelerations. Nevertheless, it has been shown that augmentative IK algorithms like those including Kalman filtering, can reduce IK errors by on-the-order of 50% (De Groote et al., 2008), so any unrealistic frame-to-frame accelerations that emerge from Bayesian IK could likely by smoothed in a similar manner. On the other hand, it is conceivable that Bayesian IK itself could smooth multi-frame IK estimates, and even improve IK results for sequential frames if velocity and/or acceleration terms are incorporated into a multi-frame forward kinematics model. This type of model could be fitted to a sequence of measured marker positions, and this would be expected to improve the physical consistency of the estimated parameters relative to separate-frame fitting.

The main limitation of the current Bayesian IK implementation was computational time, which was order-of-magnitude larger than for LS-IK (Table 2). While the current Bayesian IK implementation is likely impractical for general laboratory use, it has been shown that optimized Bayesian IK can execute much more rapidly, and even faster than LS-IK for large degree-of-freedom problems (Courty and Arnaud, 2008). This computational efficiency may be useful for both real-time IK (Borbély and Szolgay, 2017) and real-time inverse dynamics applications (Pizzolato et al., 2016).

A final limitation was that we did not consider soft tissue artifact (STA). STA corrections could be easily incorporated within a Bayesian IK framework provided the STA can be modeled in a forward sense. However, since this was the first study of which we are aware to compare Bayesian IK to LS-IK for single joint kinematics, we chose to focus on simple-as-possible mechanisms to isolate effects of marker measurement error on IK error. In addition to a more complex forward model, STA-corrected Bayesian IK would involve both increased computational resource demands and experimental validation of the forward STA model. We therefore leave STA-corrected Bayesian IK for future work.

In summary, the current results suggest that Bayesian IK offers order-of-magnitude improved accuracy over LS-IK for estimations of both planar kinematic chain poses and planar rigid body rotations. This improved accuracy stems from differences in optimization objectives: whereas LS-IK minimizes error, Bayesian IK instead maximizes probability. Additional studies are needed to elucidate the accuracy of Bayesian IK for 3D motion.

Acknowledgments

This work was supported in part by Kiban B Grant 17H02151 from the Japan Society for the Promotion of Science.

Conflict of Interest Statement

The authors report no conflict of interest, financial or otherwise.

References

Anaconda, 2017. Anaconda Software Distribution version 3-5.1. URL: `https://anaconda.com`.

Aristidou, A., and Lasenby, J., 2009. *Inverse Kinematics: a review of existing techniques and introduction of a new iterative fast solver*. University of Cambridge Technical Report CUED/F-INFENG/TR-632.

Aristidou, A., and Lasenby, J., 2011. FABRIK: A fast, iterative solver for the Inverse Kinematics problem. *Graphical Models 73*, 243–260.

Borbély, B. J., and Szolgay, P., 2017. Real-time inverse kinematics for the upper limb: a model-based algorithm using segment orientations. *BioMedical Engineering OnLine* (pp. 1–29).

Chen, L., Armstrong, C. W., and Raftopoulos, D. D., 1994. An investigation on the accuracy of three-dimensional space reconstruction using the direct linear transformation technique. *Journal of Biomechanics 27*, 493–500.

Courty, N., and Arnaud, E., 2008. Inverse kinematics using sequential Monte Carlo methods. In F. J. Perales, and R. B. Fisher (Eds.), *Articulated Motion and Deformable Objects. AMDO 2008. Lecture Notes in Computer Science, vol 5098* (pp. 1–10). Berlin, Heidelberg: Springer.

De Groote, F., De Laet, T., Jonkers, I., and De Schutter, J., 2008. Kalman smoothing improves the estimation of joint kinematics and kinetics in marker-based human gait analysis. *Journal of Biomechanics 41*, 3390–3398.

Delp, S. L., Anderson, F. C., Arnold, A. S., Loan, P., Habib, A., John, C. T., Guendelman, E., and Thelen, D. G., 2007. OpenSim: Open-Source Software to Create and Analyze Dynamic Simulations of Movement. *IEEE transactions on bio-medical engineering 54*, 1940–1950.

Duka, A.-V., 2014. Neural network based inverse kinematics solution for trajectory tracking of a robotic arm. *Procedia Technology 12*, 20–27.

Elster, C., and Wübbeler, G., 2017. Bayesian regression versus application of least squares—an example. *Metrologia 53*, 10–16.

Gamage, S., and Lasenby, J., 2002. New least squares solutions for estimating the average centre of rotation and the axis of rotation. *Journal of Biomechanics 35*, 87–93.

Halvorsen, K., 2003. Bias compensated least squares estimate of the center of rotation. *Journal of Biomechanics 36*, 999–1008.

Halvorsen, K., Lesser, M., and Lundberg, A., 1999. A new method for estimating the axis of rotation and the center of rotation. *Journal of Biomechanics 32*, 1221–1227.

Nocedal, J. I., and Wright, S. J., 2006. *Numerical Optimization.* Springer, New York.

Patil, A., Huard, D., and Fonnesbeck, C. J., 2010. PyMC: Bayesian stochastic modelling in Python. *Journal of Statistical Software* .

Pizzolato, C., Reggiani, M., Modenese, L., and Lloyd, D. G., 2016. Real-time inverse kinematics and inverse dynamics for lower limb applications using OpenSim. *Computer Methods in Biomechanics and Biomedical Engineering 20*, 436–445.

van Rossum, G., 2014. The python library reference release 2.7.8. URL: `https://docs.python.org/2/library/`.

Schwartz, M. H., and Rozumalski, A., 2005. A new method for estimating joint parameters from motion data. *Journal of Biomechanics 38*, 107–116.

Söderkvist, I., and Wedin, P. Å., 1993. Determining the movements of the skeleton using well-configured markers. *Journal of Biomechanics 26*, 1473–1477.

Wells, D. J. M., Alderson, J. A., Dunne, J., Elliott, B. C., and Donnelly, C. J., 2017. Prescribing joint co-ordinates during model preparation to improve inverse kinematic estimates of elbow joint angles. *Journal of Biomechanics 51*, 111–117.

Windolf, M., Götzen, N., and Morlock, M., 2008. Systematic accuracy and precision analysis of video motion capturing systems—exemplified on the Vicon-460 system. *Journal of Biomechanics 41*, 2776–2780.

Yuan, M., Wan, C., and Wei, L., 2016. Superiority of empirical Bayes estimator of the mean vector in multivariate normal distribution. *Science China Mathematics 59*, 1175–1186.

Table 1: Percentage of simulations in which Bayesian errors were smaller than least-squares errors (overall median percentage = 95.3%). Results are shown for the proximal joint center (r_x and r_y) and rotation angles (ϕ). For the rotation results (bottom two rows) ϕ_1 represents angular displacement. Each percent value was derived from 1000 simulations.

Model	Reference	r_x	r_y	ϕ_1	ϕ_2	ϕ_3
1-link posture	-	96.2	97.1	96.4	-	-
2-link posture	-	94.7	94.2	95.3	94.8	-
3-link posture	-	95.4	92.7	94.1	93.9	95.7
1-link rotation	Halvorsen et al. (1999)	97.6	97.1	98.2	-	-
1-link rotation	Söderkvist et al. (1993)	67.9	95.6	93.0	-	-

Table 2: Mean computational durations for the four IK methods. Durations for Bayesian IK are presented with units: seconds; all other durations have units: milliseconds.

Model	Quasi-Newton (ms)	Halvorsen (ms)	Soderkvist (ms)	Bayesian (s)
1-link posture	78.8	-	-	2044.0
2-link posture	306.2	-	-	4746.2
3-link posture	285.9	-	-	2987.0
1-link rotation	-	1.7	0.2	11.6

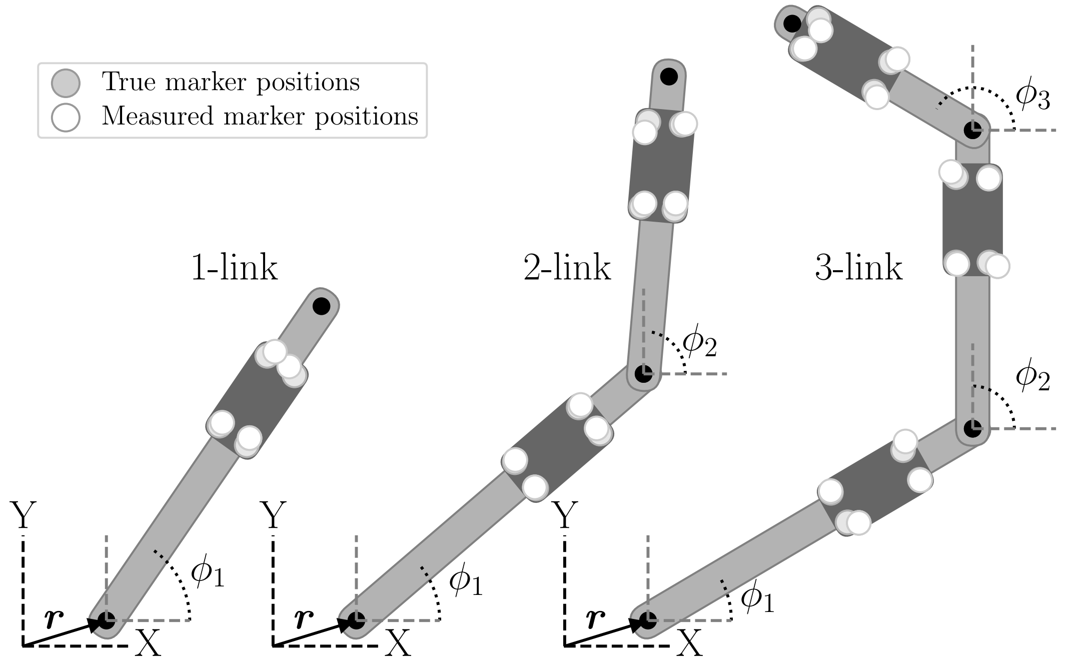


Figure 1: Planar n-link kinematic chain models. The true local positions of all marker coordinates (i.e. positions with respect to each segment) were assumed to be known. The inverse kinematics goal was to estimate mechanism posture including: global position $\boldsymbol{r}$ and global orientations ϕ_i , where i represents the ith joint.

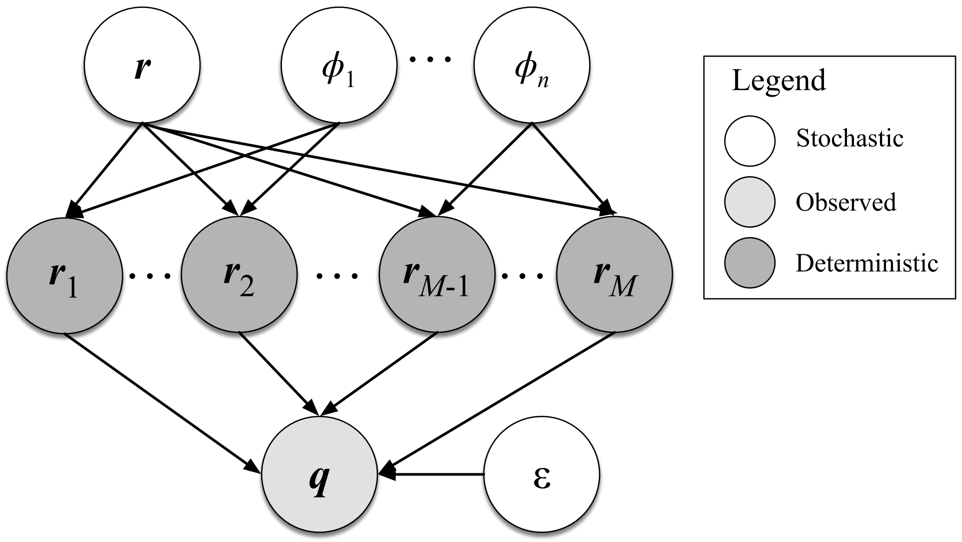


Figure 2: Bayesian causal, forward kinematics model of the planar n-link kinematic chain postures depicted in Fig.1. The top row represents the posture to be estimated: the global position of the the first link's proximal joint center ($\boldsymbol{r}$) and segment orientations (ϕ_i). Once values for these posture parameters are assigned, the true values of all global marker positions $\boldsymbol{r}_{\mathrm{M}j}$ are known. The observed marker positions $\boldsymbol{q}$ are a function of these true positions and random error ε. The Bayesian IK goal was to mutually maximize the posterior maximum likelihoods of all stochastic variables given the observations $\boldsymbol{q}$.

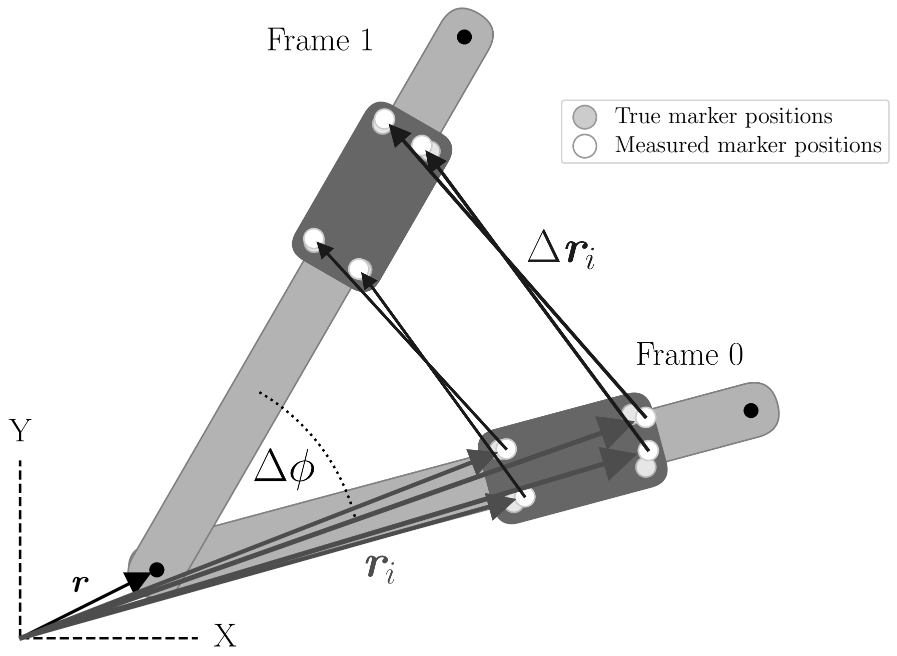


Figure 3: Planar angular displacement-based inverse kinematics (IK) model. Based on initial marker positions $\boldsymbol{r}_i$ and marker displacements $\Delta\boldsymbol{r}_i$, the IK goal was to estimate the center of rotation $\boldsymbol{r}$ and the angular displacement $\Delta\phi$. Unlike the posture model (Fig.1), and following least-squares techniques (Söderkvist & Wedin 1993; Halvorsen et al. 1999), this model did not assume that true local marker positions were known.

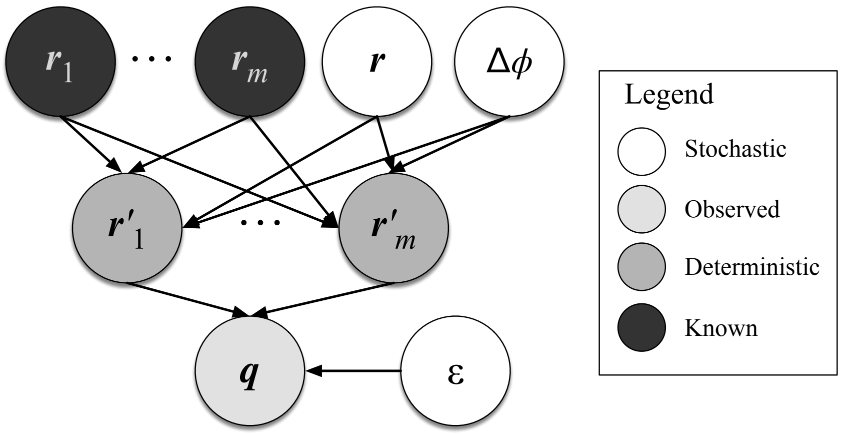


Figure 4: Bayesian causal, forward kinematics model of planar rotation. In this model, initial marker positions $\boldsymbol{r}_i$ are assumed to have been measured accurately, and are thus treated as known constants. When values for the center of rotation $\boldsymbol{r}$ and angular displacement $\Delta\phi$ are decided, the true post-rotation marker positions $\boldsymbol{r}'_i$ are also known. These true positions plus measurement error ϵ yield the observed post-rotation marker positions $\boldsymbol{q}$. Like the posture model (Fig.2), the Bayesian IK goal was to mutually maximize the posterior maximum likelihoods of all stochastic variables given the observations $\boldsymbol{q}$.

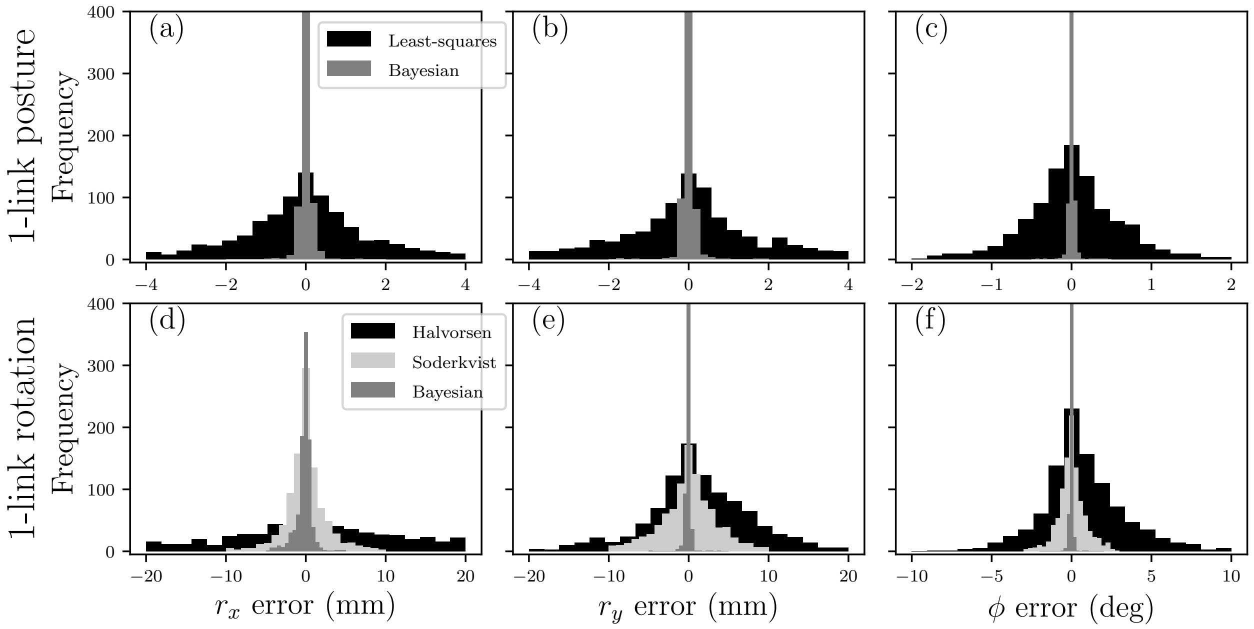


Figure 5: Error distributions for estimates of 1-link posture (a–c) and rotation (d–f). The three columns represent the three estimated parameters (x position, y position, angular position/rotation). A total of 1000 simulations was conducted for each panel in which both marker noise amplitude and angular posture / displacement were randomly varied. Similar results were obtained for 2-link and 3-link posture estimates (see Supplementary Material)

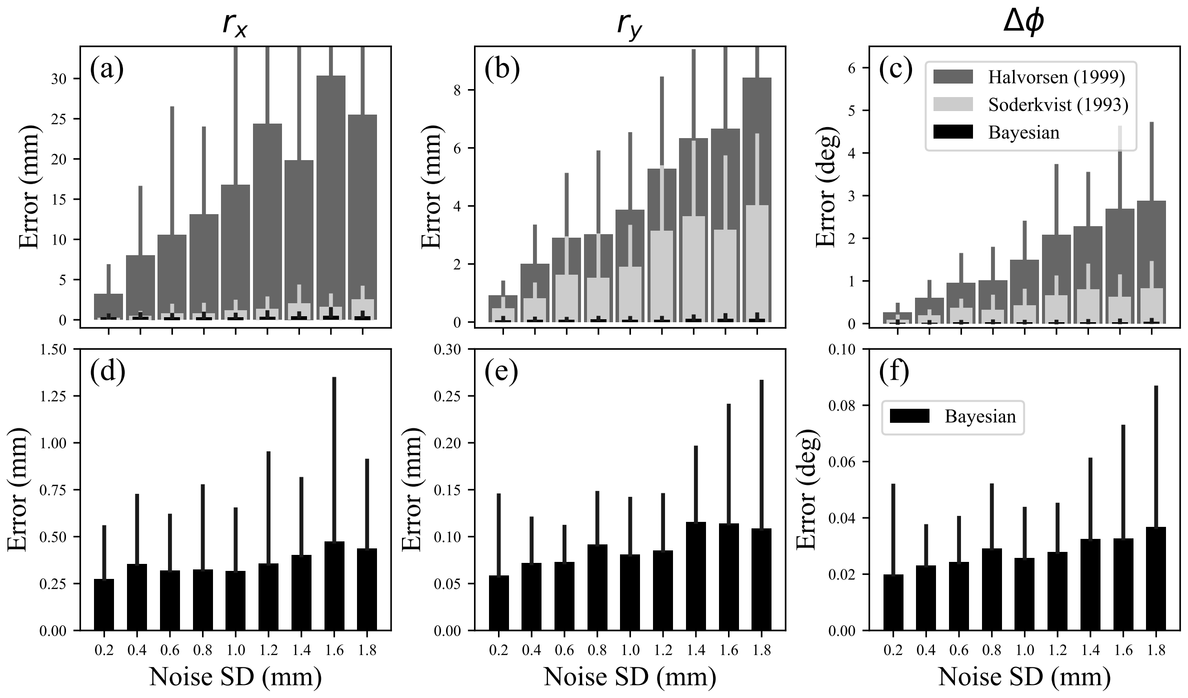


Figure 6: IK errors vs. marker noise standard deviation (SD). Vertical bars indicate interquartile ranges. (a–c) Errors from all three methods. (d–f) Bayesian errors only (magnified for clarity).

Supplementary Material

Interactive workbooks and HTML renderings of this supplementary material is available in this project's repository on GitHub:

https://github.com/0todd0000/BayesIK

PDF versions are attached below.

Supplementary material

NOTES:

- This supplementary material assumes that the reader is new to Bayesian analysis and/or to Bayesian analysis in Python using PyMC (http://pymc-devs.github.io/pymc/) (Patil et al. 2010), but is familiar with basic statistical procedures like t tests and linear regression. This material closely follows Davidson-Pilon (2015) (https://camdavidsonpilon.github.io/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/) but from the perspective of experimentatal data analysis in science. It focusses on what Bayesian analysis can do, how to conduct Bayesian analysis using the high-level PyMC (http://pymc-devs.github.io/pymc/) interface, and how to interpret the results.
- Dependencies to run the code contained in this supplementary material include: Python 3, Numpy 1.11, Scipy 1.0, Matplotlib 2.2 and PyMC 2.3. Other versions of these dependencies may also work.
- This supplementary material is available in both static HTML and dynamic Jupyter notebook (http://jupyter.org) formats. In Jupyter the notebooks can be modified and code executed directly in the notebook. Readers uninterested in the Python details can simply ignore the Python code and instead focus on the text, figures and numerical results.

For readers new to Bayesian analysis:

The Bayesian perspective on experimental data is quite different from the perspective adopted by classical null hypothesis testing (NHT) procedures. Unlike introductions to NHT, which typically describe one- and two-sample t tests, it is unnatural to start Bayesian study from one and two sample tests. Bayesian versions of these tests exist, but it is much more natural to introduce Bayesian analysis with prior distributions, posterior distributions, numerical simulations and single-parameter estimates. Since NHT and Bayesian analysis introductions are fundamentally different it may seem difficult to connect Bayesian and NHT concepts. However, after spending some time to understand Bayesian fundamentals, it should gradually become apparent not only how Bayesian fundamentals relate to NHT, but also how the Bayesian perspective offers a considerably more flexibile data analysis framework.

APPENDIX A: Bayesian analysis basics

This Appendix provides an overview of Bayesian analysis. It is directed at beginners and focuses on concepts rather than on mathematics.

Imagine that you are going to conduct an experiment to determine whether in a population of students the left knee extension strength is higher than the right, or vice versa. (This is not a particularly interesting experiment but will serve to illustrate many Bayesian analysis concepts.) Before the experiment you have no idea whether the left or right knee will be stronger, so you presume that the left is just as likely as the right to be stronger.

The relative left-right strength across all subjects can be represented using a single parameter *x* which ranges from zero to one, where *x*=0 implies that 100% of all students have stronger left knees, and where *x*=1 implies that 100% of all students have stronger right knees. A value of *x*=0.5 implies that half of all students have stronger left legs, and half have stronger right legs. The experimental goal will be to find the most likely value of *x* based on the observed data.

Before considering experimental data, let's first consider what we know and what we don't know about *x*:

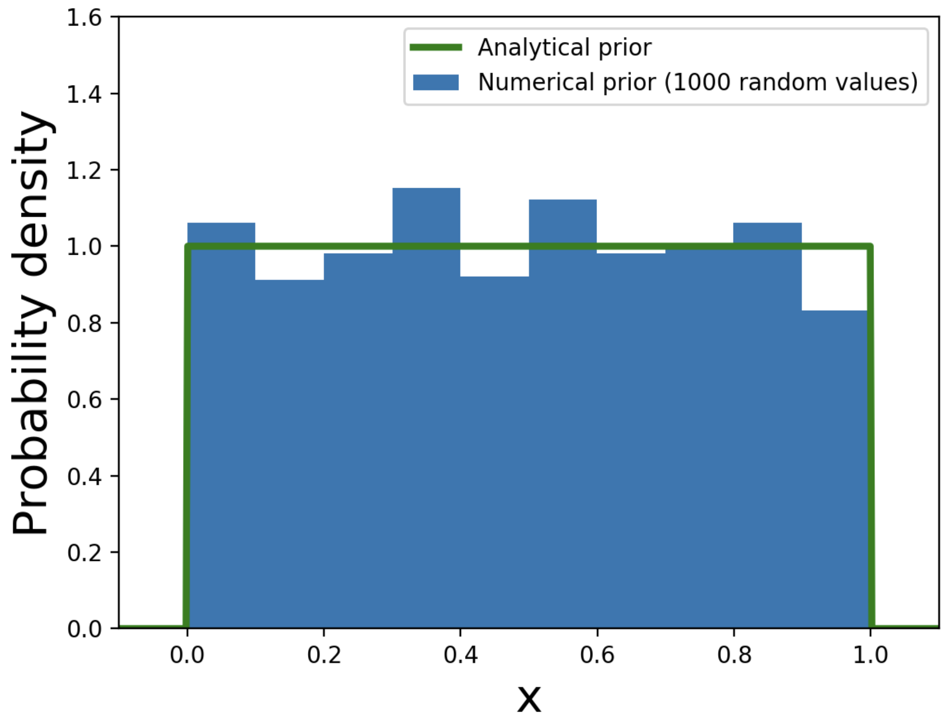
- We know that *x* can range from zero to one.
- We have no idea what the actual value of *x* will be.

Both pieces of information are vital to Bayesian analysis. The first describes the plausible values of x, and the second describes what we believe about x before the experiment. Two more points, also central to Bayesian analysis are:

- We would like to update our pre-experiment beliefs regarding the value of *x* based on the experimental data.
- We would also like to update our uncertainty regarding *x* based on the experimental data, but we recognize that we can never know *x* precisely, so we want to retain uncertainty regarding *x* in our final results.

In Bayesian terms, x is a stochastic variable, and our pre-experiment beliefs regarding x are embodied in the so-called **prior**, or equivalently the "prior distribution of x". Our prior can be visualized as follows (feel free to ignore the Python code and skip to the figure below):

In [1]:
```
%matplotlib notebook

import numpy as np
from scipy import stats
from matplotlib import pyplot
import pymc

#construct an analytical prior:
x = np.linspace(-0.1, 1.1, 501)  #range of plausible x values, here expanded pa
st 0 and 1 for visualization purposes
prior_analytical = stats.uniform.pdf(x)

#construct a numerical prior using PyMC:
nSamples = 1000  #number of random samples used to approximate x's distribution
x_stochastic = pymc.Uniform("MyPriorDistribution", 0, 1)  #stochastic variable
representing our prior
prior_numerical = np.array( [x_stochastic.random()  for i in range(nSamples)] )
#generate random data in the presumed plausible range

#plot the results:
pyplot.figure()
pyplot.hist(prior_numerical, density=True, label="Numerical prior (%d random va
lues)" %nSamples)
pyplot.plot(x, prior_analytical, linewidth=3, color='g', label="Analytical prio
r")
pyplot.xlim(-0.1, 1.1)
pyplot.ylim(0, 1.6)
pyplot.legend()
pyplot.xlabel('x', size=20)
pyplot.ylabel('Probability density', size=20);
```



The figure above depicts our prior both analytically and numerically. The analytical representation is a mathematical function:

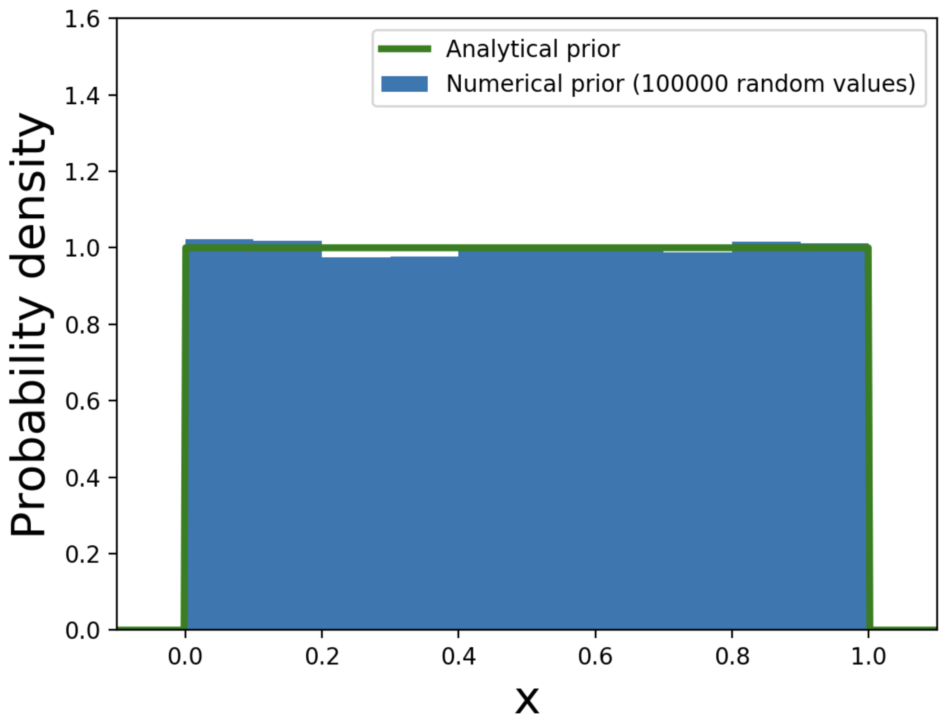
$$f(x) = \begin{cases} 1, & \text{if } 0 \leq x \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

and based on this equation we can compute precise probabilities for *x*. For example, the probability that *x* is between 0.2 and 0.5 is:

$$P(0.2 \leq x \leq 0.5) = \int_{0.2}^{0.5} f(x) = 0.3$$

We could instead regard the prior numerically, as a distribution of randomly sampled numbers, which approximates the analytical distribution. While the analytical distribution is perfectly precise, the numerical distribution's accuracy depends on the number of samples (actual numerical values) we use to construct it. The figure above uses 10,000 values, but if we were to use 1,000,000 values our approximation would be better as depicted below.

In [2]:
```
#construct a numerical prior using PyMC:
nSamples = 100000
prior_numerical = np.array( [x_stochastic.random()  for i in range(nSamples)] )

#plot the results:
pyplot.figure()
pyplot.hist(prior_numerical, density=True, label="Numerical prior (%d random va
lues)" %nSamples)
pyplot.plot(x, prior_analytical, linewidth=3, color='g', label="Analytical prio
r")
pyplot.xlim(-0.1, 1.1)
pyplot.ylim(0, 1.6)
pyplot.legend()
pyplot.xlabel('x', size=20)
pyplot.ylabel('Probability density', size=20);
```



This idea will reappear throughout this supplementary material: the accuracy of a numerically estimated probability density improves as the number of samples gets larger. Since larger numerical samples require more computing resources, it is often important in Bayesian analysis to find a balance between sample size and accuracy. For the remainder of this analysis we presume that 10,000 samples is accurate enough.

A final point regarding this prior, and probability distributions in general, is its most likely value. Even though our prior implies that all values of *x* are equally plausible, the most likely value of *x* is 0.5.

```
In [3]: x_most_likely = prior_numerical.mean()
        print( x_most_likely )
```

```
0.500135480575
```

This is termed the maximum likelihood estimate (MLE) of *x*. To understand why the MLE is 0.5, simply think of an alternative value like 0.7. If the MLE were 0.7 it would imply that right (x=1) is more likely than left (x=0) and therefore that not all values of *x* (from 0 to 1) are equally likely. The prior distribution of x would also be different to the probability density plots above.

Now that we have decided upon our prior distribution for *x*, we are ready to conduct our experiment and then update our beliefs regarding *x* based on the data we measure. Let's imagine first that we measure just a single subject and it turns out that this subject's right knee is stronger than their left knee. This observation can be represented as:

$$x_{\text{obs}} = [\ 1\ ]$$

How should we update our beliefs regarding x based on x_{obs} ? A skeptic might say we shouldn't change our beliefs based on a single observation. A Bayesian would say that we should update our beliefs according to Bayes' rule:

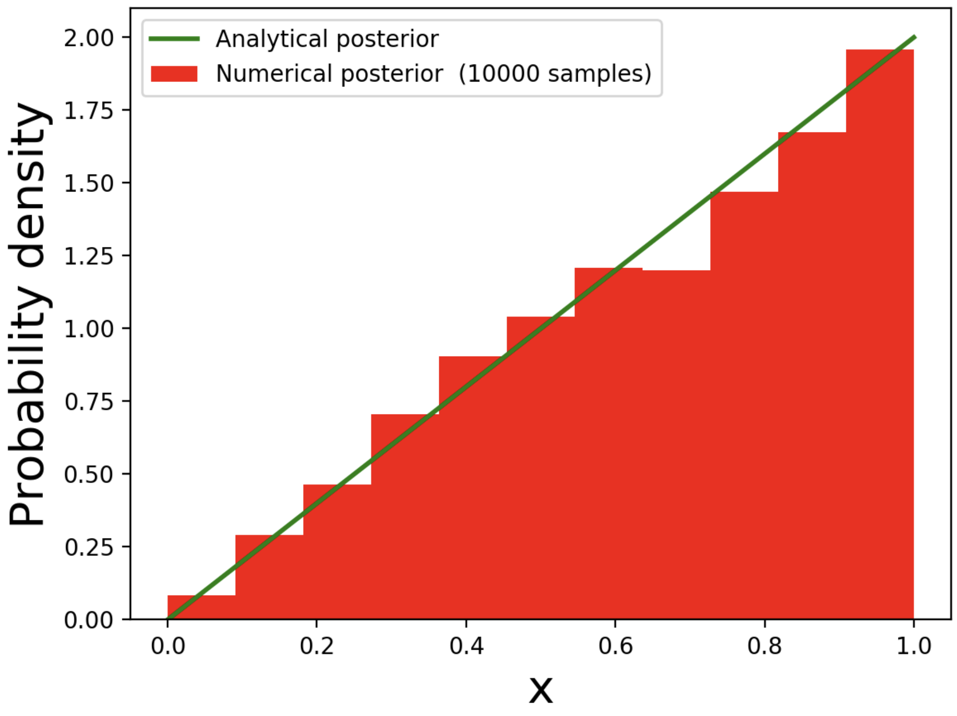
$$P(\ A \mid x_{\text{obs}}\ ) = \frac{P(\ x_{\text{obs}} \mid A)P(\ A\ )}{P(\ x_{\text{obs}}\ )}$$

{NOTE TO SELF: Bayes' rule requires an explanation}

Applying Bayes' rule to our observed data produces the following posterior:

```
In [4]: x_observed = [1]
        n_observations = len(x_observed)
        n_right_stronger = sum(x_observed)

        # create a model of the experimental variables:
        x = pymc.Uniform("x", lower=0, upper=1)  #prior distribution for x
        y = pymc.Binomial("y", n=n_observations, p=x, value=n_right_stronger, observed=
        True) #model of experimental observations

        # numerically fit the model to the data:
        model = pymc.Model([x, y])
        mcmc = pymc.MCMC(model)
        nSamples = 10000
        nBurn = 1000
        mcmc.sample(nSamples+nBurn, nBurn)

        #compute analytical posterior:
        xx = np.linspace(0, 1, 21)
        posterior_analytical = stats.beta.pdf(xx, n_observations+1, n_observations - n_
        right_stronger + 1)

        # check results:
        posterior = mcmc.trace("x")[:]  #numerical posterior
        pyplot.figure()
        pyplot.hist(posterior, density=True, range=(0,1), bins=11, facecolor="r", label
        ="Numerical posterior  (%d samples)" %nSamples)
        pyplot.plot(xx, posterior_analytical, 'g', lw=2, label="Analytical posterior")
        pyplot.legend(loc="upper left")
        pyplot.xlabel('x', size=20)
        pyplot.ylabel('Probability density', size=20);
```

```
 [-----------------100%-----------------] 11000 of 11000 complete in 0.4 sec
```



Bayes' rule has changed the shape of our prior distribution into the posterior distribution depicted above. According to Bayes' rule, we should no longer believe that all values of *x* are equally plausible. Instead we should believe that *x*=1 is more likely than *x*=0, and that *x*'s MLE is 0.67. Note that, like the prior distribution, the total area under the posterior curve is one. This means that the probability that x is between 0 and 1 is 100%.

In [5]:
```
x_most_likely = posterior.mean()
print( x_most_likely )
```

```
0.659363282929
```

NOTE:

Understanding the posterior's analytical derivation is not essential for understanding how Bayesian analysis actually works. For more complex priors and/or for more complex data models it is cumbersome or even impossible to compute posteriors analytically. To avoid those complexities this Appendix discusses only the numerical approach, but presents the analytical results in the figures as a reference. Readers interested in analytical solutions should consult textbooks like (REFERENCE).

This analysis represents a conceptual departure from classical null hypothesis testing (NHT). NHT requires variance estimates (usually as standard deviations), and thus multiple observations, but the Bayesian approach can operate on a single observation. This is made possible by our specification of a prior, which implicitly embodies variance.

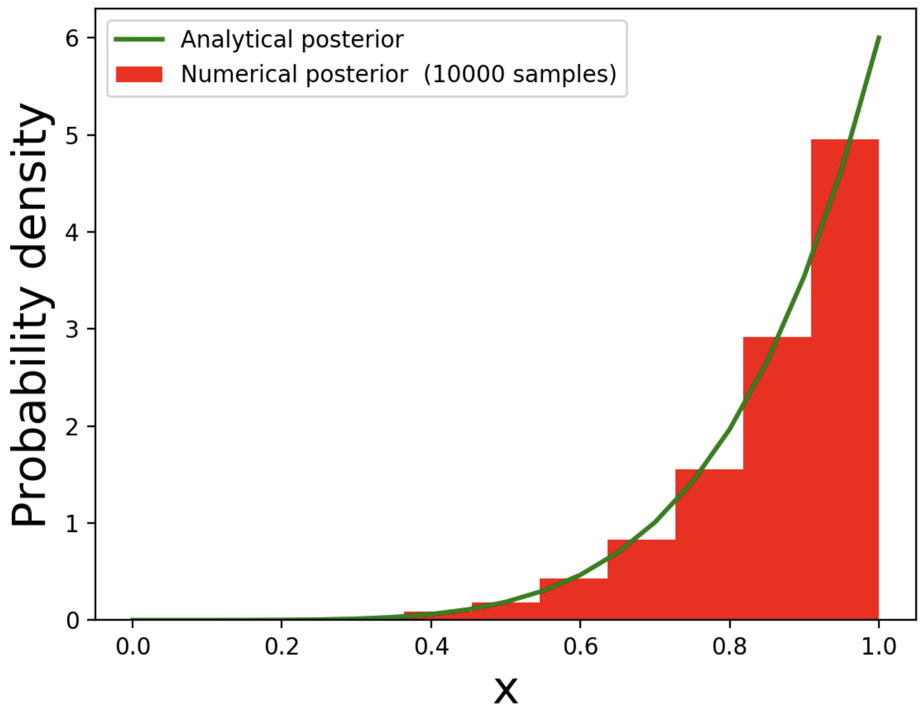
The prior can be thought of as a compliant 1D surface, in this case a perfectly flat surface, and our observation (x=1) can be thought of as a peg that pokes the surface upward at that point, dragging the rest of the surface with it to retain a total area of one.

This raises an unsettling question, and an important limitation of Bayesian analysis: *what if we had specified a different prior distribution?* That is an important question to which we will momentarily return. First let's consider what happens with multiple observations.

Imagine that we measured four more subjects, all who exhibit a stronger right than left knee. What does our posterior look like now?

```
In [6]: x_observed = [1, 1, 1, 1, 1]
        n_observations = len(x_observed)
        n_right = sum(x_observed)
        n_left = n_observations - n_right

        # create a model of the experimental variables:
        x = pymc.Uniform("x", lower=0, upper=1)  #prior distribition for x
        y = pymc.Binomial("y", n=n_observations, p=x, value=n_right, observed=True) #mo
        del of experimental observations

        # numerically fit the model to the data:
        model = pymc.Model([x, y])
        mcmc = pymc.MCMC(model)
        mcmc.sample(11000, 1000)

        #compute analytical posterior:
        xx = np.linspace(0, 1, 21)
        posterior_analytical = stats.beta.pdf(xx, n_right+1, n_left+1)
        # check results:
        posterior = mcmc.trace("x")[:]  #numerical posterior
        pyplot.figure()
        pyplot.hist(posterior, density=True, range=(0,1), bins=11, facecolor="r", label
        ="Numerical posterior  (%d samples)" %nSamples)
        pyplot.plot(xx, posterior_analytical, 'g', lw=2, label="Analytical posterior")
        pyplot.legend(loc="upper left")
        pyplot.xlabel('x', size=20)
        pyplot.ylabel('Probability density', size=20);
```

```
 [-----------------100%-----------------] 11000 of 11000 complete in 0.3 sec
```



Our posterior is now biased even more toward x=1, reflecting our updated belief that the true value of x is likely 1 or close to it. This posterior implies that the probability that x's true value is zero is very small, and indeed that all values less than 0.5 are quite unlikely. We can compute the probability that x's true value is less than 0.5 as:

```
In [7]: b = posterior < 0.5     #binary vector containing ones where posterior is less
        than 0.5, and zeros elsewhere
        s = sum(b)              #total number of values less than 0.5
        p = float(s) / nSamples #proportion of values less than 0.5
        print( p )
```

```
0.0193
```

Or more briefly as:

```
In [8]: p = (posterior < 0.5).mean()
        print( p )
```

```
0.0193
```

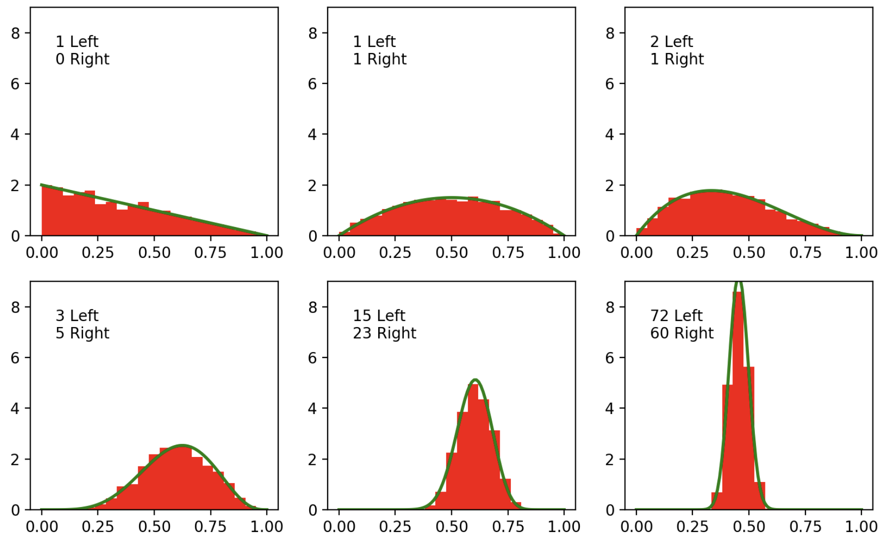
In the calculations we can see another important departure from classical NHT. Whereas NHT usually involves a single result: a test statistic and associated p value, Bayesian posteriors allow for much more flexible data-relevant questions. One example is: *what is the probability that x is not between 0.4 and 0.6?* That question is very data-relevant, and is also not readily addressible using NHT. With Bayesian analysis the answer can easily be calculated from the posterior:

```
In [9]: p = ( (posterior < 0.4) | (posterior > 0.6) ).mean()
        print( p )
```

```
0.956
```

Before returning to the issue of how the prior affects the posterior, let's consider how different datasets affect the posterior.

```
In [10]: #create six different datasets
         X_OBSERVED = []
         X_OBSERVED.append( [0] )
         X_OBSERVED.append( [1, 0] )
         X_OBSERVED.append( [1, 0, 0] )
         X_OBSERVED.append( [1]*5 + [0]*3 )  #5 ones and 3 zeros
         X_OBSERVED.append( [1]*23 + [0]*15 )  #23 ones and 15 zeros
         X_OBSERVED.append( [1]*60 + [0]*72 )    #60 ones and 72 zeros


         pyplot.figure(figsize=(10, 6))


         for i,x_observed in enumerate(X_OBSERVED):
             n_observations = len(x_observed)
             n_right = sum(x_observed)
             n_left = n_observations - n_right

             # create a model of the experimental variables:
             x = pymc.Uniform("x", lower=0, upper=1)  #prior distribition for x
             y = pymc.Binomial("y", n=n_observations, p=x, value=n_right, observed=True)
         #model of experimental observations

             # numerically fit the model to the data:
             model = pymc.Model([x, y])
             mcmc  = pymc.MCMC(model)
             mcmc.sample(11000, 1000)

             #compute analytical posterior:
             xx = np.linspace(0, 1, 101)
             posterior_analytical = stats.beta.pdf(xx, n_right+1, n_left+1)

             # check results:
             posterior = mcmc.trace("x")[:]  #numerical posterior
             ax = pyplot.subplot(2,3,i+1)
             ax.hist(posterior, density=True, range=(0,1), bins=21, facecolor="r")
             ax.plot(xx, posterior_analytical, 'g', lw=2, label="Analytical posterior")
             ax.set_ylim(0, 9)
             ax.text(0.1, 0.75, '%d Left\n%d Right' %(n_left, n_right), transform=ax.tra
         nsAxes)
```



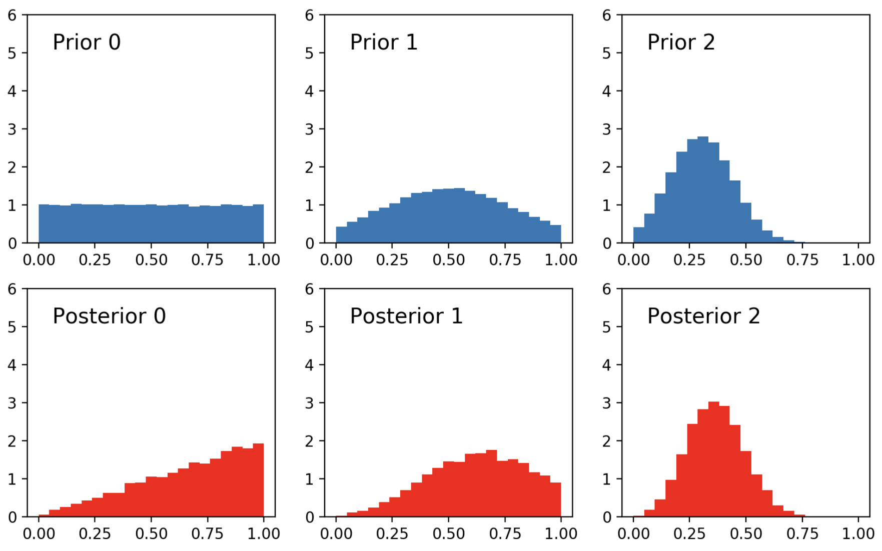
```
[-----------------100%-----------------] 11000 of 11000 complete in 0.3 sec
```

In the figure above we can see that:

- As the number of observations increases, we become increasingly confident that we are honing in on x's true value.
- We retain uncertainty, even for relatively large datasets.
- We never completely dismiss the possibility that x's true value can be anywhere between 0 and 1.
- We can compute arbitrary probabilities, based on the observed data, regarding where along the 0-to-1 range x's true value lies.

Now let's return to the issue of the prior, and how it affects the posterior. Let's first consider just a single "Right" observation and then compare the previous flat (uniform) prior to two different normal priors.

In [11]:
```
x_observed = [1]
n_observations = len(x_observed)
n_right = sum(x_observed)
n_left = n_observations - n_right

# create three different priors:
x0 = pymc.Uniform("x", lower=0, upper=1)
x1 = pymc.Normal("x", mu=0.5, tau=10)
x2 = pymc.Normal("x", mu=0.3, tau=50)

pyplot.figure(figsize=(10,6))

for i,x in enumerate([x0,x1,x2]):
    y      = pymc.Binomial("y", n=1, p=x, value=1, observed=True) #model of expe
rimental observations
    model = pymc.Model([x, y])
    mcmc  = pymc.MCMC(model)
    mcmc.sample(51000, 10000)

    prior     = [x.random() for ii in range(50000)]  #numerically sampled prior
distribution
    posterior = mcmc.trace("x")[:]  #numerical posterior
    ax = pyplot.subplot(2,3,i+1)
    ax.hist(prior, density=True, range=(0,1), bins=21)
    ax.set_ylim(0, 6)
    ax.text(0.1, 0.85, 'Prior %d' %i, transform=ax.transAxes, size=14)

    ax = pyplot.subplot(2,3,i+4)
    ax.hist(posterior, density=True, range=(0,1), bins=21, facecolor="r")
    ax.set_ylim(0, 6)
    ax.text(0.1, 0.85, 'Posterior %d' %i, transform=ax.transAxes, size=14)
```

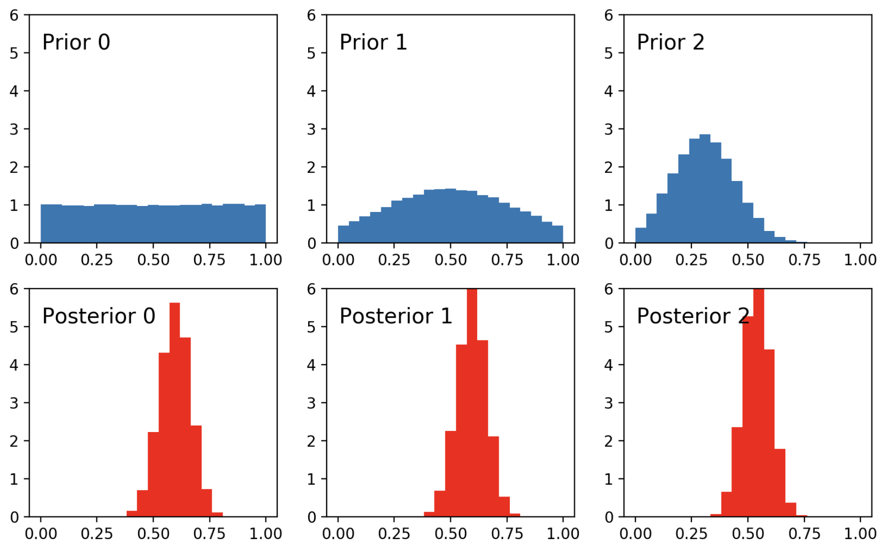


```
[-----------------100%-----------------] 51000 of 51000 complete in 1.5 sec
```

From these results we notice that single observations have a substantial effect on flat priors, but contrastingly little effect on narrow priors. In other words, the prior distribution's narrowness reflects our confidence, and this confidence is not easily shaken by a single observation.

Let's now look at what happens when we use the exact same priors, but with a larger dataset consisting of 20 Left and 30 Right observations.

In [12]:
```
x_observed = [0]*20 + [1]*30
n_observations = len(x_observed)
n_right = sum(x_observed)
n_left = n_observations - n_right

# create three different priors:
x0 = pymc.Uniform("x", lower=0, upper=1)
x1 = pymc.Normal("x", mu=0.5, tau=10)
x2 = pymc.Normal("x", mu=0.3, tau=50)

pyplot.figure(figsize=(10,6))

for i,x in enumerate([x0,x1,x2]):
    y     = pymc.Binomial("y", n=n_observations, p=x, value=n_right, observed=T
rue) #model of experimental observations
    model = pymc.Model([x, y])
    mcmc  = pymc.MCMC(model)
    mcmc.sample(51000, 10000)

    prior     = [x.random() for ii in range(50000)]  #numerically sampled prior
distribution
    posterior = mcmc.trace("x")[:]  #numerical posterior
    ax = pyplot.subplot(2,3,i+1)
    ax.hist(prior, density=True, range=(0,1), bins=21)
    ax.set_ylim(0, 6)
    ax.text(0.05, 0.85, 'Prior %d' %i, transform=ax.transAxes, size=14)

    ax = pyplot.subplot(2,3,i+4)
    ax.hist(posterior, density=True, range=(0,1), bins=21, facecolor="r")
    ax.set_ylim(0, 6)
    ax.text(0.05, 0.85, 'Posterior %d' %i, transform=ax.transAxes, size=14)
```



```
[-----------------100%-----------------] 51000 of 51000 complete in 2.0 sec
```

These results show that, irrespective of the prior, our beliefs converge to approximately the same posterior distribution. In other words, Bayes' rule permits a relatively large amount of data to objectively sway our prior beliefs in a data-driven manner, whatever those prior beliefs may be.

The issue of the prior-dependent posteriors is thus applicable primarily to small datasets. There are objective ways for selecting priors, but that discussion is beyond the scope of this paper. Like above the main paper considers a variety of priors and then qualitatively compares the posteriors.

REFERENCES

Darwiche A. (2009). *Modeling and reasoning with Bayesian networks*. Cambridge University Press.

Davidson-Pilon C (2015). *Probabilistic Programming & Bayesian Methods for Hackers*. Addison-Wesley Professional.

Patil A, Huard D, Fonnesbeck CJ (2010). PyMC: Bayesian stochastic modelling in Python. *Journal of Statistical Software* 35(4): 1.

APPENDIX B: Bayesian vs. least-squares estimates of population means

This Appendix compares Bayesian to least-squares (LS) estimates of population means, where the LS estimate is simply the sample mean (or arithmetic mean). This Appendix aims to demonstrate that even rudimentary LS concepts like the sample mean can be interpreted somewhat differently when regarding data from a Bayesian perspective.

Below two Bayesian approaches to population mean estimation are described, and these results are then subsequently compared to the LS mean.

Let's start by drawing a random sample from the standard normal distribution, for which the true population mean and standard deviation (SD) are 0 and 1, respectively:

In [1]:
```
%matplotlib notebook

import numpy as np
from matplotlib import pyplot
import pymc

np.random.seed(4)        #arbitrarily set the random number generator seed
J = 5                    #sample size
y = np.random.randn(J)   #random data sample
print('A random sample from the standard normal distribution:')
print('   (True population mean = 0)')
print('   (True population SD   = 1)')   #standard deviation (SD)
print(y)
```

```
A random sample from the standard normal distribution:
   (True population mean = 0)
   (True population SD   = 1)
[ 0.05056171  0.49995133 -0.99590893  0.69359851 -0.41830152]
```

Approach 1: Least-squares

The least squares (LS) solution minimizes the following function:

$$f(x) = \sum (y_i - x)^2$$

where x is the estimate of the population mean, and y_i are the individual sample observations. The value of x which minimizes $f(x)$ is the sample mean:

In [2]:
```
mean_ls = np.mean( y )  #optimum value for "x" in the equation above
print('Least-squares estimate of the population mean (i.e. sample mean):')
print( mean_ls )
```

```
Least-squares estimate of the population mean (i.e. sample mean):
-0.0340197804923
```

Note that the sample mean is not zero, even through the true population mean is zero. For the sample mean to converge precisely to the true population mean, one would generally have to draw an infinitely large sample, or sample the entire population.

Approach 2: Bayesian (known variance)

The simplest Bayesian approach is to presume that population SD (σ) is known. Let's implement the full solution in PyMC, then consider the commands afterwards.

```
In [3]: #(1) Build model
        x          = pymc.Uniform("x", -3, 3)  #prior distribution for the population m
        ean
        sigma      = 1                         #known population standard deviation (SD
        )
        tau        = 1.0 / sigma**2            #measurement precision ( 1 / SD^2 )
        data_model = pymc.Normal("data_model", x, tau, observed=True, value=y)  #model
        of the observations

        #(2) Conduct inference:
        model = pymc.Model([data_model, x])    #full model containing data model and al
        l stochastic variables
        mcmc  = pymc.MCMC(model)               #Markov Chain Monte Carlo sampler
        mcmc.sample(iter=11000, burn=1000)     #MC sampling
        post  = mcmc.trace("x")[:]             #posterior distribution for the populati
        on mean
        bmean = post.mean()                    #Bayesian estimate of the population mea
        n
        print()
        print('Bayesian estimate of the population mean (known variance):')
        print(bmean)
```

```
 [-----------------100%-----------------] 11000 of 11000 complete in 0.4 sec
Bayesian estimate of the population mean (known variance):
-0.0292631259209
```

Note that the Bayesian estimate of the population mean is closer to its true value of zero than was the LS estimate. Let's consider each step of the Bayesian estimation procedure outlined in the code above.

In order to conduct Bayesian inference regarding experimental data, a model of those data is required. The first step is to create a prior distribution for each stochastic variable:

```
In [4]: x          = pymc.Uniform("x", -3, 3)  #naive prior distribution for the popula
        tion mean
```

This prior implies that we expect the true population mean to be between -1 and +1, and that all values in this range are equally likely. As described in Appendix A, the prior can affect the final results, and we shall consider effects of priors below.

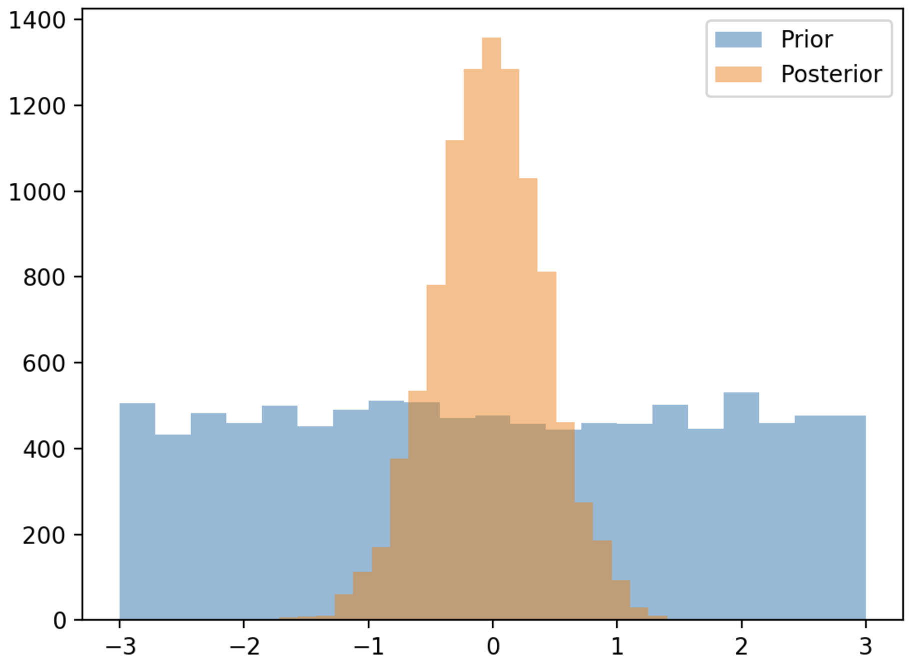
The next step is to build a model of the data:

```
In [5]: data_model = pymc.Normal("data_model", x, tau, observed=True, value=y)  #model
        of the observations
```

This command implies that we believe that our data sample comes from a normal distribution with an unknown population mean of x and a known precision of tau=1. The "observed=True" and "value=y" keyword arguments specify that the values in y were observed. In other words, based on the values y we want to find the most likely value for x, or more generally the posterior distribution for x given y.

Once we have the model we can conduct inference, and the Markov Chain Monte Carlo (MCMC)-related inference commands above follow from Appendix A so are not discussed further here. Following inference, it can be informative to visualize the prior and the posterior distributions:

```
In [6]: prior   = [x.random() for i in range(10000)]

        pyplot.figure()
        ax = pyplot.axes()
        ax.hist(prior, bins=21, alpha=0.5, label='Prior')      #prior distribution for
        x
        ax.hist(post,  bins=21, alpha=0.5, label='Posterior')  #posterior distribution
        for x
        ax.legend();
```



The observed data sample y has changed our beliefs regarding x from the depicted prior distribution into the depicted posterior distribution via Bayes' rule.

Approach 3: Bayesian (unknown variance)

In order to model unknown variance we just need to change the precision variable (tau) into a stochastic variable, then ensure that tau is included in the full pymc model like this:

In [7]:
```
#(1) Build model
x          = pymc.Uniform("x", -3, 3)       #prior distribution for the populat
ion mean
tau        = pymc.Uniform("tau", 0.01, 100) #prior distribution for precision
data_model = pymc.Normal("data_model", x, tau, observed=True, value=y)  #model
of the observations

#(2) Conduct inference:
model = pymc.Model([data_model, x, tau])   #full model containing data model an
d all stochastic variables
mcmc  = pymc.MCMC(model)                  #MCMC sampler
mcmc.sample(iter=11000, burn=1000)        #MC sampling
post  = mcmc.trace("x")[:]                #posterior distribution for the populati
on mean
bmean = post.mean()                       #Bayesian estimate of the population mea
n
print()
print('Bayesian estimate of the population mean (unknown variance):')
print(bmean)
```

```
 [-----------------100%-----------------] 11000 of 11000 complete in 0.7 sec
Bayesian estimate of the population mean (unknown variance):
-0.028969135243
```

Adding tau as a stochastic variable has not greatly affected the results.

Systematic comparison between Bayesian and LS approaches

The analyses above represent a single dataset, single sample size, and single prior distributions, and results showed that the Bayesian results were somewhat more accurate. How do Bayesian and LS approaches compare for arbitrary datasets, sample sizes, priors, etc.? To answer that question comprehenively would require millions of simulations. Here we shall consider only effects of dataset, and we'll check whether Bayesian or LS approaches perform better.

In [8]:
```
#create a function for estimating population mean using Bayesian inference:
def bayesian_mean(y, iter=11000, burn=1000):
    #build model:
    x       = pymc.Uniform("x", -3, 3)
    tau     = pymc.Uniform("tau", 0.01, 100)
    dmodel  = pymc.Normal("data_model", x, tau, observed=True, value=y)
    #conduct inference:
    model   = pymc.Model([data_model, x])
    mcmc    = pymc.MCMC(model)
    mcmc.sample(iter=iter, burn=burn, progress_bar=False)
    post    = mcmc.trace("x")[:]
    return post.mean()


np.random.seed(0)
nDatasets    = 20   #number of datasets to generate
n            = 0    #number of datasets for which the Bayesian estimate is more
accurate
for i in range(nDatasets):
    y       = np.random.randn(J)
    mean_ls = y.mean()
    mean_b  = bayesian_mean(y, iter=110000, burn=10000)
    if (abs(mean_ls) < abs(mean_b)):
        result = 'LS'
    else:
        result = 'Bayesian'
        n     += 1
    print('Dataset %d.  LS: %.5f, Bayesian: %.5f (%s more accurate)' %(i, mean_
ls, mean_b, result))
print()
print('Bayesian more accurate in %d/%d datasets.' %(n,nDatasets))
```

```
Dataset 0.  LS: 1.45028, Bayesian: 1.44847 (Bayesian more accurate)
Dataset 1.  LS: 0.15133, Bayesian: 0.15841 (LS more accurate)
Dataset 2.  LS: -0.72543, Bayesian: -0.72868 (LS more accurate)
Dataset 3.  LS: -0.22593, Bayesian: -0.22542 (Bayesian more accurate)
Dataset 4.  LS: -0.11956, Bayesian: -0.11465 (Bayesian more accurate)
Dataset 5.  LS: 1.47015, Bayesian: 1.46882 (Bayesian more accurate)
Dataset 6.  LS: -0.80645, Bayesian: -0.80766 (LS more accurate)
Dataset 7.  LS: -0.40028, Bayesian: -0.39992 (Bayesian more accurate)
Dataset 8.  LS: 0.22548, Bayesian: 0.22538 (Bayesian more accurate)
Dataset 9.  LS: -0.56589, Bayesian: -0.56771 (LS more accurate)
Dataset 10.  LS: 0.00039, Bayesian: 0.00468 (LS more accurate)
Dataset 11.  LS: 0.21691, Bayesian: 0.21854 (LS more accurate)
Dataset 12.  LS: -0.14829, Bayesian: -0.14565 (Bayesian more accurate)
Dataset 13.  LS: -0.18566, Bayesian: -0.18351 (Bayesian more accurate)
Dataset 14.  LS: -0.41073, Bayesian: -0.41028 (Bayesian more accurate)
Dataset 15.  LS: 0.22798, Bayesian: 0.22986 (LS more accurate)
Dataset 16.  LS: 0.25266, Bayesian: 0.24976 (Bayesian more accurate)
Dataset 17.  LS: -0.35683, Bayesian: -0.35616 (Bayesian more accurate)
Dataset 18.  LS: 0.20721, Bayesian: 0.20923 (LS more accurate)
Dataset 19.  LS: -0.47126, Bayesian: -0.46361 (Bayesian more accurate)

Bayesian more accurate in 12/20 datasets.
```

We can see that the Bayesian approach is occasionally, but not compellingly more accurate than the LS approach.

Summary

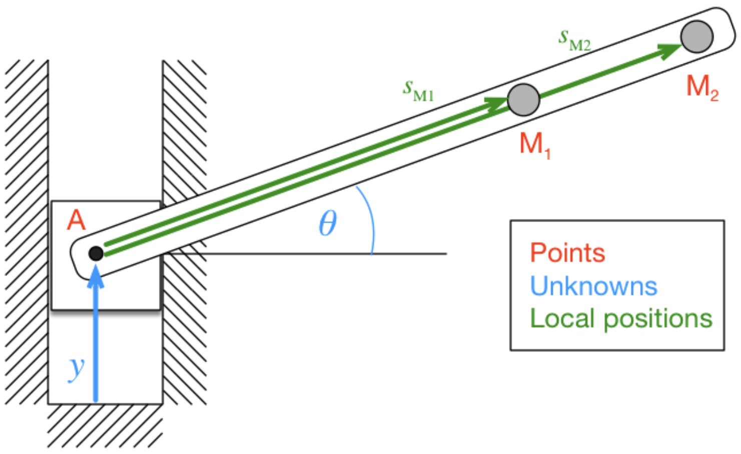
This Appendix has shown that population means can be estimated not only using the typical LS approach (i.e. the sample mean), but also using at least two different Bayesian approaches. Results show that Bayesian estimates are slightly more accurate than LS estimates for some datasets, but overall that there is not much difference between Bayesian and LS results. However, this is a very simple case of Bayesian inference, with a very simple data model. Subsequent Appendices and the main paper show that Bayesian and LS accuracies diverge as the data model becomes increasingly complex.

APPENDIX D: Bayesian and least-squares results divergence

This Appendix describes a minimal inverse kinematics model for which Bayesian and least-sqaures results diverge.

Model

The planar mechanism depicted below has two degrees of freedom (DOF): one translational and one rotational. The first (slider) segment moves to a position y and the second segment rotates about point A to an angle θ. Two markers (M1 and M2) are rigidly fixed to the rotating segment and have fixed local positions: $\boldsymbol{s}_{\mathrm{M1}} = \begin{Bmatrix} 35 \\ 0 \end{Bmatrix}$ and $\boldsymbol{s}_{\mathrm{M2}} = \begin{Bmatrix} 45 \\ 0 \end{Bmatrix}$.



The three points' true global positions are:

$$\boldsymbol{r}_{\mathrm{A}} = \begin{Bmatrix} 0 \\ y \end{Bmatrix}$$

$$\boldsymbol{r}_{\mathrm{M1}} = \boldsymbol{r}_{\mathrm{A}} + \boldsymbol{R}\,\boldsymbol{s}_{\mathrm{M1}}$$

$$\boldsymbol{r}_{\mathrm{M2}} = \boldsymbol{r}_{\mathrm{A}} + \boldsymbol{R}\,\boldsymbol{s}_{\mathrm{M2}}$$

where $\boldsymbol{R}$ is the rotation matrix:

$$\boldsymbol{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Inverse kinematics (IK) problem

Imagine that we have measured the following marker positions:

$$\boldsymbol{r}'_{\mathrm{M1}} = \begin{Bmatrix} 33.51 \\ 12.11 \end{Bmatrix}$$

$$\boldsymbol{r}'_{\mathrm{M2}} = \begin{Bmatrix} 42.63 \\ 16.18 \end{Bmatrix}$$

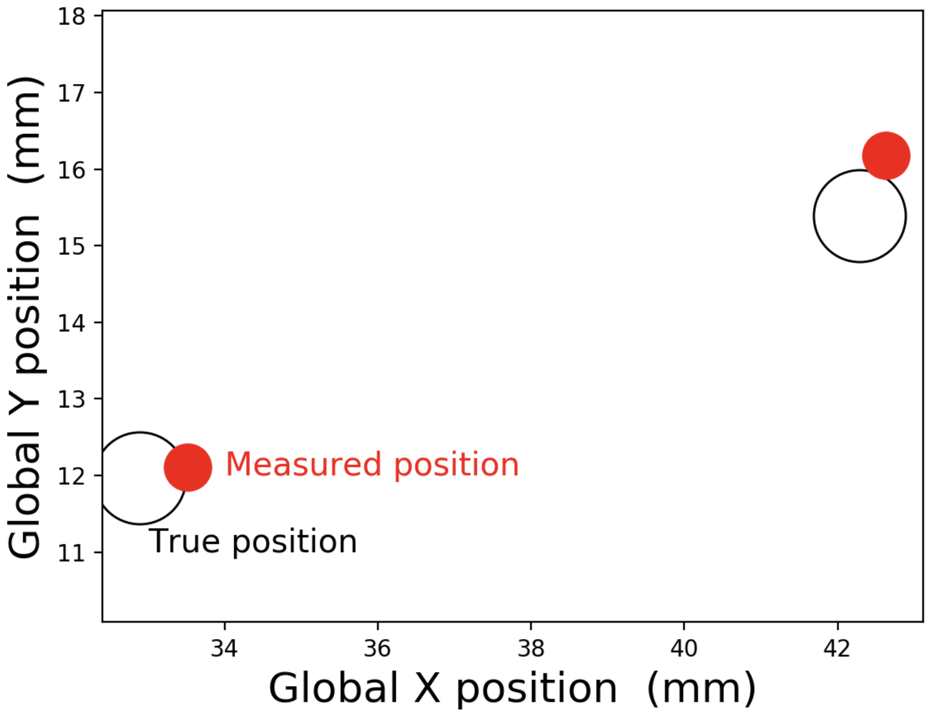
The IK problem is to estimate y and θ given these measurements.

For argument's sake let's say we also happen to know that the true values of the unknown variables are: y = 0 and θ = 20 deg. In this case the true global marker positions are:

$$\boldsymbol{r}_{\mathrm{M1}} = \begin{Bmatrix} 32.89 \\ 11.97 \end{Bmatrix}$$

In [1]:

```
%matplotlib notebook

from math import sin,cos,radians,degrees
import numpy as np
from matplotlib import pyplot

#(0) Define known values:
### local positions:
sM1        = np.array([35.0, 0.0])
sM2        = np.array([45.0, 0.0])
### measurements:
rpM1       = np.array([33.51, 12.11])
rpM2       = np.array([42.63, 16.18])
### true values: (not used in IK solutions, but useful for visualization)
y_true     = 0.0
theta_true = radians(20)



#(1) Compute global marker positions given y and theta:
def rotation_matrix(theta):
    '''Construct rotation matrix'''
    c,s    = cos(theta), sin(theta)
    R      = np.matrix(  [[c, -s], [s, c]] )
    return R

def rotate(R, r):
    '''Rotate a position vector r using rotation matrix R'''
    return np.asarray(  (R * np.mat(r).T)  ).flatten()

def get_positions(y, theta):
    '''Compute global marker positions given y and theta'''
    rA     = np.array([0, y])       #global A position
    R      = rotation_matrix(theta)
    rM1    = rA + rotate(R, sM1)    #global M1 position
    rM2    = rA + rotate(R, sM2)    #global M2 position
    return rM1,rM2

rM1,rM2    = get_positions(y_true, theta_true)



#(2) Plot:
pyplot.figure()
ax = pyplot.axes()
ax.plot(rM1[0],  rM1[1],  'o', markeredgecolor='k', markerfacecolor='w', ms=40)
ax.plot(rM2[0],  rM2[1],  'o', markeredgecolor='k', markerfacecolor='w', ms=40)
ax.plot(rpM1[0], rpM1[1], 'o', markeredgecolor='r', markerfacecolor='r', ms=20)
ax.plot(rpM2[0], rpM2[1], 'o', markeredgecolor='r', markerfacecolor='r', ms=20)
ax.text(33, 11, 'True position',     color='k', size=14)
ax.text(34, 12, 'Measured position', color='r', size=14)
ax.set_xlabel('Global X position  (mm)', size=18)
ax.set_ylabel('Global Y position  (mm)', size=18)
pyplot.axis('equal')
pyplot.show()
```



Measured position
True position
Global X position (mm)
Global Y position (mm)
34
36
38
40
42
11
12
13
14
15
16
17
18

Least-squares solution

This IK problem is relatively simple, so it could be solved using an analytical approach similar to the one presented in Appendix B. However, we'll be a bit lazy and instead solve the IK problem numerically, using the "get_positions" function that we've already written above.

Once we set values for y and θ we know the global positions of $\boldsymbol{r}_{\mathrm{M1}}$ and $\boldsymbol{r}_{\mathrm{M2}}$, so we can compute the difference between those positions and our measurements as:

$$\begin{aligned} \boldsymbol{\epsilon}_1 &= \boldsymbol{r}'_{\mathrm{M1}} - \boldsymbol{r}_{\mathrm{M1}} \\ \boldsymbol{\epsilon}_2 &= \boldsymbol{r}'_{\mathrm{M2}} - \boldsymbol{r}_{\mathrm{M2}} \end{aligned}$$

We can then express total measurement error as follows:

\begin{equation} f(y, \theta) = \big| \ \boldsymbol{\epsilon}_1 \ \big| \ ^2 \

- \ \big| \ \boldsymbol{\epsilon}_2 \ \big| \ ^2 \end{equation}

Although we have called $f(y, \theta)$ a "measurement error" function, the opposite perspective is equally apt: "guess error". That is, if we know our measurement values ($\boldsymbol{r}'_{\mathrm{M1}}$, $\boldsymbol{r}'_{\mathrm{M2}}$) and then use y and θ to guess the true marker positions ($\boldsymbol{r}_{\mathrm{M1}}$, $\boldsymbol{r}_{\mathrm{M2}}$), then $f(y, \theta)$ represents our guess error. We nevertheless use "measurement error" below because we presume there is no guess which can yield zero error.

For later purposes let's bundle our two unknown variables into a generalized "unknowns" vector $\boldsymbol{x}$ as follows:

$$\boldsymbol{x} = \begin{Bmatrix} y \\ \theta \end{Bmatrix}$$

Now our error function is:

\begin{align} f(\boldsymbol{x}) = \big| \ \boldsymbol{\epsilon}_1 \ \big| \ ^2 \

- \ \big| \ \boldsymbol{\epsilon}_2 \ \big| \ ^2 \end{align}

Let's implement this function in Python and then explore its output to make sure it's returning reasonable values.

In [2]:
```python
def measurement_error(x):
    y,theta = x
    rM1,rM2 = get_positions(y, theta)
    e1,e2   = rpM1 - rM1, rpM2 - rM2
    e1,e2   = np.linalg.norm(e1), np.linalg.norm(e2)
    f       = e1**2 + e2**2
    return f

x = [0, radians(20)]
print( measurement_error(x) )
```

```
1.14563304519
```

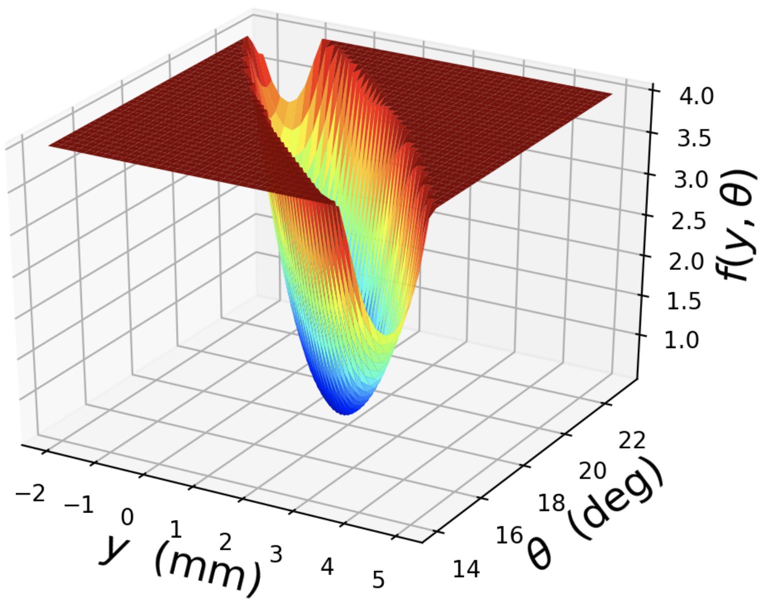
Here the total measurement error is 1.146 mm^2. By moving to a different vertical position (e.g. y=2) or a different angular position (θ=10 deg) we see that measurement error increases:

In [3]:
```python
print(   measurement_error( [2, radians(20)] )   )
print(   measurement_error( [0, radians(10)] )   )
```

```
5.43207890941
110.13801125
```

Since we can compute the error function $f(y, \theta)$ for arbitrary y and θ values we can view it as a three-dimensional surface as follows:

```
In [4]: from matplotlib import cm
        from mpl_toolkits.mplot3d import Axes3D


        ### Compute error for a range of y and theta values:
        Y        = np.linspace(-2, 5, 41)
        THETA    = np.radians( np.linspace(14, 23, 41) )
        Y,THETA = np.meshgrid(Y, THETA)
        F        = [measurement_error(x)    for x in zip(Y.flatten(), THETA.flatten())]
        F        = np.reshape(F, Y.shape)
        F[F>4]   = 4  #cap all values at 4 to more clearly see the function's minimum


        ### Plot:
        fig      = pyplot.figure()
        ax       = fig.gca(projection='3d')
        ax.plot_surface(Y, np.degrees(THETA), F, cmap=cm.jet, rstride=1, cstride=1, lin
        ewidth=0.2)
        ax.set_xlabel(r'$y$  (mm)', size=18)
        ax.set_ylabel(r'$\theta$  (deg)', size=18)
        ax.set_zlabel(r'$f(y, \theta)$', size=18)
        pyplot.show()
```



In the figure above it appears that our error function's minimum value occurs for approximately y=1.5 and θ=19, but let's find its minimum value algorithmically using **scipy.optimize.minimize** as follows:

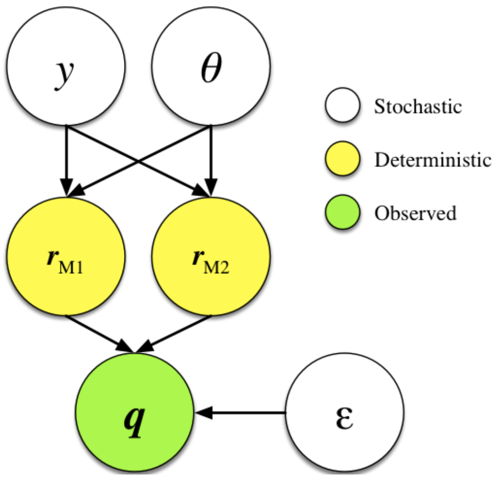
In [5]:
```
from scipy import optimize

x0       = [0, radians(20)]  #initial (y, theta) guess
results  = optimize.minimize(measurement_error, x0)

print('Least-squares estimates:')
print('   y     = %.3f' %results.x[0])
print('   theta = %.3f' %degrees(results.x[1]) )
```

```
Least-squares estimates:
   y     = 1.321
   theta = 18.700
```

Due to measurement error our final estimates for y and θ are slightly different from their true values (y=0, θ=20).

Bayesian approach

The figure below depicts a stochastic forward-kinematics model for this system. Once the numerical values of y and θ are known we know the values of our deterministic variables $\boldsymbol{r}_{\mathrm{M1}}$ and $\boldsymbol{r}_{\mathrm{M2}}$. Similarly, once the numerical value of our measurement error ϵ is known we will also know the value of our generalized measurement vector $\boldsymbol{q}$, where $\boldsymbol{q}$ contains all four measured coordinates from $\boldsymbol{r}'_{\mathrm{M1}}$ and $\boldsymbol{r}'_{\mathrm{M2}}$. Let's implement this model in PyMC.



y
θ
Stochastic
Deterministic
Observed
$\boldsymbol{r}_{\mathrm{M1}}$
$\boldsymbol{r}_{\mathrm{M2}}$
$\boldsymbol{q}$
ε

In [6]:
```
import pymc


q_observed = np.asarray([rpM1, rpM2]).flatten() #measured positions
tau        = 20  #presumed measurement precision (we'll relax this later)
y          = pymc.Normal("y", 0, 1)  #prior for y
theta      = pymc.Uniform("theta", radians(-45), radians(45))  #prior for theta


@pymc.deterministic
def observations_model(y=y, theta=theta):
    rM1,rM2 = get_positions(y, theta)
    q       = np.asarray([rM1, rM2]).flatten()
    return q
q_model    = pymc.Normal("q", observations_model, tau, value=q_observed, observe
d=True)
```

Now that we have a model of our observations we can set values for y and θ and then check what kinds of results our model produces.

```
In [7]: y.set_value(0)
        theta.set_value( radians(20) )
        print( q_model.random() )
        print( q_model.random() )
        print( q_model.random() )
```

```
[ 32.78484058  11.89083546  42.45642173  16.02977362]
[ 33.28954332  11.9619234   42.19980949  15.71036284]
[ 32.77358118  12.14726452  42.56477381  15.23909662]
```

The first two columns represent r'_{M1} and the last two columns represent r'_{M2}. We can see that these values are similar to our actual measured values, so our model appears to have been specified correctly.

Variability exists in these values because we have not set the value of our last stochastic variable (ϵ); this variability is built into the model above via the parameter "tau" which specifies our measurement precision. If we raise tau to a very high value we will see that the model's results become much less variable.

```
In [8]: tau       = 1e9
        q_model   = pymc.Normal("q", observations_model, tau, value=q_observed, observe
        d=True)
        print( q_model.random() )
        print( q_model.random() )
        print( q_model.random() )
        print( q_model.random() )
        print( q_model.random() )
```

```
[ 32.88928498  11.97068501  42.28618804  15.39089926]
[ 32.88924399  11.97076292  42.28616353  15.39089276]
[ 32.88931438  11.97073356  42.2861079   15.39092759]
[ 32.88925868  11.97070977  42.2862221   15.39091137]
[ 32.88926231  11.97067152  42.2861532   15.390917  ]
```

Now let's relax our assumptions regarding tau's true value, then use PyMC to run Markov-Chain Monte-Carlo simulations with the goal of finding posterior distributions for y and θ (and tau) which are most consistent with the observed marker positions.

```
In [9]: tau        = pymc.Normal("tau", 20, 1)  #prior for measurement precision
        y          = pymc.Normal("y", 0, 1)     #prior for y
        theta      = pymc.Uniform("theta", radians(-45), radians(45))  #prior for theta

        @pymc.deterministic
        def observations_model(y=y, theta=theta):
            rM1,rM2 = get_positions(y, theta)
            q       = np.asarray([rM1, rM2]).flatten()
            return q
        q_model   = pymc.Normal("q", observations_model, tau, value=q_observed, observe
        d=True)


        mcmc      = pymc.MCMC([q_model, y, theta, tau])
        mcmc.sample(40000, 20000)
        Y         = mcmc.trace('y')[:]
        THETA     = np.degrees( mcmc.trace('theta')[:] )
        TAU       = np.degrees( mcmc.trace('tau')[:] )

        print('\n\nBayesian estimates:')
        print('   y     = %.3f' %Y.mean() )
        print('   theta = %.3f' %THETA.mean() )
```

```
 [-----------------100%-----------------] 40000 of 40000 complete in 7.7 sec

Bayesian estimates:
   y     = 1.145
   theta = 18.923
```

Our Bayesian estimates for y and θ are somewhat closer to their true values than are our least-squares estimates. But this is just for one case. Let's now compare the two approaches' results more systematically.

Systematic approach comparison

First let's pick some new true values for y and θ and then generate ten random datasets based on those true values.

In [10]:
```
y_true      = -0.1
theta_true  = radians(5)
rM1,rM2     = get_positions(y_true, theta_true)
q_true      = np.array([rM1, rM2]).flatten()

nIterations = 10
noise_amp   = 0.5
np.random.seed(0)
Q_obs       = q_true + noise_amp * np.random.randn(nIterations, 4)

print(Q_obs)
```

```
[[ 35.74884061   3.1505296   45.31813041   4.94245502]
 [ 35.80059343   2.46181206  45.30380562   3.74632982]
 [ 34.81520501   3.15575025  44.9007832    4.54914518]
 [ 35.2473333    3.0112885   45.05069303   3.98884559]
 [ 35.61385397   2.84787186  44.98529526   3.39496055]
 [ 33.59031953   3.27726029  45.26097951   3.45092591]
 [ 36.00169175   2.22326816  44.85164067   3.7284165 ]
 [ 35.63320404   3.68513038  44.90623513   4.01108968]
 [ 34.42292156   1.96005276  44.65480534   3.90018291]
 [ 35.48195977   3.55164092  44.63509801   3.67085705]]
```

These are the ten measurements we'll test. Let's first compute the least-squares solutions for each set of measurements.

In [11]:
```
def measurement_error(x, q_obs):
    y,theta = x
    rpM1    = q_obs[:2]
    rpM2    = q_obs[2:]
    rM1,rM2 = get_positions(y, theta)
    e1,e2   = rpM1 - rM1, rpM2 - rM2
    e1,e2   = np.linalg.norm(e1), np.linalg.norm(e2)
    f       = e1**2 + e2**2
    return f

def solution_ls(q_obs):
    x0      = [y_true, theta_true]  #initial (y, theta) guess
    results = optimize.minimize(measurement_error, x0, args=(q_obs,))
    y,theta = results.x
    return y, degrees(theta)

np.set_printoptions(precision=3, suppress=True)
RESULTS_LS  = np.array([solution_ls(q_obs)  for q_obs in Q_obs])
print('Least-squares results:')
print('   y     = %s' %RESULTS_LS[:,0])
print('   theta = %s' %RESULTS_LS[:,1])
```

```
Least-squares results:
   y     = [ 0.568  0.501 -0.742  0.672  1.625  1.289 -0.372  2.901 -3.663  3.
141]
   theta = [ 4.988  3.732  6.596  4.055  2.144  2.974  4.801  1.356  9.488  0.
674]
```

Next let's get Bayesian estimates for each dataset.

In [12]:
```
def solution_bayesian(q_obs):
    x0      = [y_true, theta_true]  #initial (y, theta) guess
    tau     = pymc.Normal("tau", 1/(noise_amp**2), 1)
    y       = pymc.Normal("y", y_true, 1, value=y_true)
    theta   = pymc.Uniform("theta", radians(-45), radians(45))

    @pymc.deterministic
    def observations_model(y=y, theta=theta):
        rM1,rM2 = get_positions(y, theta)
        q       = np.asarray([rM1, rM2]).flatten()
        return q
    q_model   = pymc.Normal("q", observations_model, tau, value=q_obs, observed
=True)

    mcmc      = pymc.MCMC([q_model, y, theta, tau])
    mcmc.sample(40000, 20000, progress_bar=False)
    Y         = mcmc.trace('y')[:]
    THETA     = np.degrees( mcmc.trace('theta')[:] )

    return Y.mean(), THETA.mean()

RESULTS_B = np.zeros((nIterations,2))
for i,q_obs in enumerate(Q_obs):
    print('Iteration %d of %d...' %(i+1, nIterations))
    y,theta = solution_bayesian(Q_obs[0])
    RESULTS_B[i] = [y, theta]
    print('   y = %.3f,  theta = %.3f' %(y,theta))
```

```
Iteration 1 of 10...
   y = 0.112,  theta = 5.617
Iteration 2 of 10...
   y = 0.061,  theta = 5.694
Iteration 3 of 10...
   y = 0.044,  theta = 5.708
Iteration 4 of 10...
   y = 0.176,  theta = 5.506
Iteration 5 of 10...
   y = 0.095,  theta = 5.625
Iteration 6 of 10...
   y = 0.083,  theta = 5.662
Iteration 7 of 10...
   y = 0.151,  theta = 5.557
Iteration 8 of 10...
   y = 0.068,  theta = 5.681
Iteration 9 of 10...
   y = 0.052,  theta = 5.699
Iteration 10 of 10...
   y = 0.007,  theta = 5.772
```

Here is a summary of IK error for the two approaches:

In [13]:
```
x_true   = [y_true, degrees(theta_true)]
error_LS = RESULTS_LS - x_true
error_B  = RESULTS_B - x_true

print('Average absolute error (least-squares):')
print('   y: %.3f,  theta: %.3f' %tuple( np.abs(error_LS).mean(axis=0) ) )
print('Average absolute error (Bayesian):')
print('   y: %.3f,  theta: %.3f' %tuple( np.abs(error_B).mean(axis=0) ) )
```

```
Average absolute error (least-squares):
   y: 1.587,  theta: 2.136
Average absolute error (Bayesian):
   y: 0.185,  theta: 0.652
```

We can see that, on average, the Bayesian approach produces smaller errors for both y and θ.

However, you may have noticed that our calculations above have cheated a bit by using the true values of y and θ as the algorithmic starting points. Thus the results above suggest only that the Bayesian approach performs better than the least-squares approach when both start from the known true values of y and θ. In real IK applications the true values of y and θ are of course unknown. The main manuscript uses the true values of y and θ as starting points only for LS estimation, to maximize its chances of converging to the true solution. The main manuscript then uses the LS solution as the starting point for the Bayesian estimate, to avoid potential biases associated with starting from the true solution. In other words, it gives LS estimation a huge advantage in finding the true maximum, but it gives Bayesian estimation no such advantage.

Finally, let's check how the least-squares and Bayesian approaches compare on a case-by-case basis.

In [14]:
```
error_difference = np.abs(error_LS) - np.abs(error_B)
print(error_difference)
```

```
[[ 0.457 -0.605]
 [ 0.44   0.574]
 [ 0.498  0.888]
 [ 0.496  0.439]
 [ 1.53   2.231]
 [ 1.206  1.364]
 [ 0.021 -0.358]
 [ 2.833  2.963]
 [ 3.412  3.789]
 [ 3.134  3.554]]
```

Positive values indicate that the least-squares solution had greater error and negative values indicate the opposite. This shows that the Bayesian estimates for y were better in all ten cases, and that the Bayesian estimates for θ were better in 8 of 10 cases.

Summary

This Appendix shows that Bayesian and least-squares (LS) inverse kinematics (IK) estimates can diverge even for simple planar rotations. The results suggest that Bayesian IK performs better in general than LS-IK. Generally, *probabilistically* matching noisy data to a stochastic model via Bayesian techniques outperforms LS estimates derived from noisy data.

APPENDIX E: Pose estimate errors for two- and three-link kinematic chains

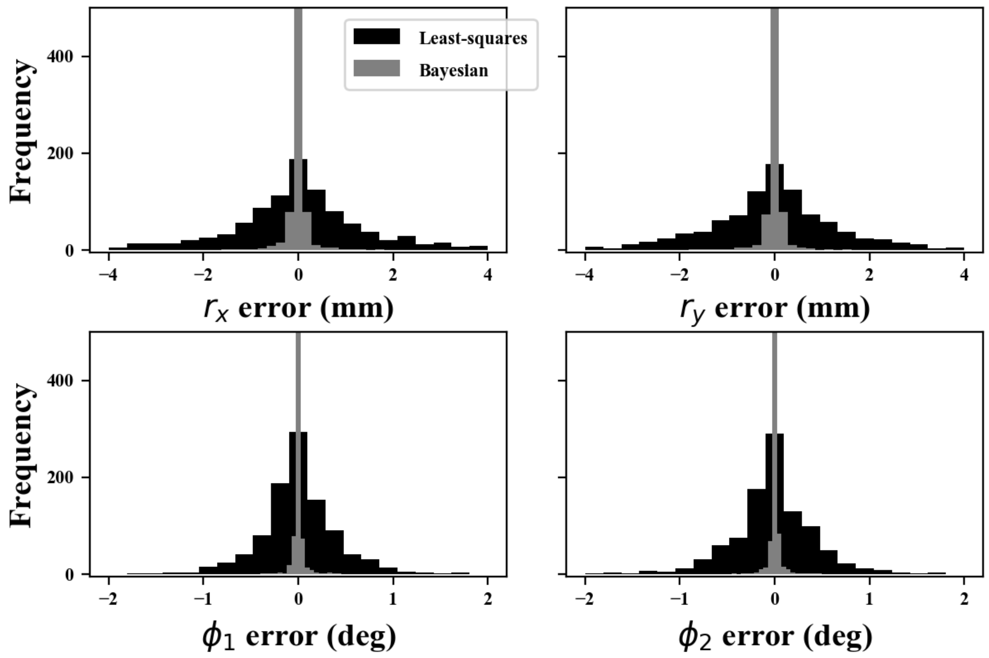


Figure E1. Two-link kinematic chain pose estimate errors.

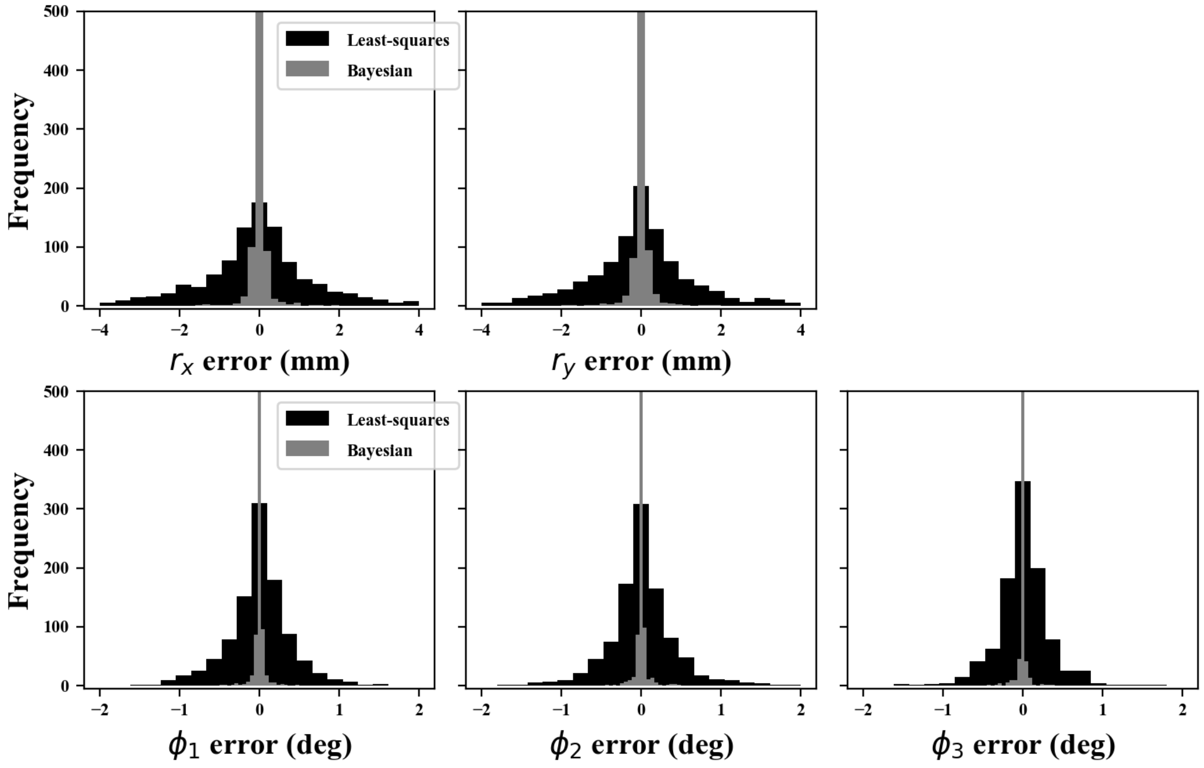


Figure E2. Three-link kinematic chain pose estimate errors.